富岳コデザイン・レポート

～フラッグシップ2020プロジェクト・テクニカルレポート～

フラッグシップ2020プロジェクト
理化学研究所 計算科学研究センター

2022年3月

概要

本レポートは、文部科学省が 2014 年度〜2020 年度まで実施された、フラッグシップ 2020 プロジェクトにおける理化学研究所（以降、理研）が実施した次世代超高速電子計算機システム（以降、富岳）の開発・整備についての技術報告書である。

富岳の開発では、2020 年頃までに世界トップレベルで幅広い課題に対応できる汎用システムを実現することを目標とし、理研は 2014 年 10 月からは富士通と共に 2015 年 8 月にかけて富岳の基本設計を実施し、2016 月 1 月までに文部科学省の評価を受けた後、詳細設計およびソフトウェア調整を実施した。2019 年 3 月には富士通と製造契約を締結した。2019 年 12 月から 2020 年 5 月にかけて富岳のハードウェア製造および設置を行い、その後、システム調整を進め、2021 年 3 月に共用を開始し、プロジェクトを終了した。

本プロジェクトにおいてはシステム開発に際し、システム設計とアプリケーションの開発を協調して行う「コデザイン」（協調設計）を行った。大規模並列システムの開発においては、設備で設定された電力で最大限の性能を得るためにはシステムの電力効率を高める必要がある。システム性能を向上させるだけでなく、ターゲットとなるアプリケーションを選択・設定し、実際の計算科学アプリケーションの特性を考慮しつつシステムを設計するとともにシステムを有効に利用するためのアプリケーションの最適化を行った。

本レポートは、システム並びにアプリケーションでのコデザインの取り組みについて、まとめたものである。

執筆責任者一覧

第 1 章 開発経緯 石川 裕（理化学研究所）、佐藤 三久（理学研究所）

第 2 章 システム概要・性能 石川 裕（理化学研究所）、佐藤 三久（理学研究所）、児玉 祐悦（理化学研究所）辻 美和子（理化学研究所）

第 3 章 システムのコデザイン

- **3.1 CPU アーキテクチャ** 佐藤 三久（理学研究所）、児玉 祐悦（理化学研究所）
- **3.2 オペレーティングシステム** 石川 裕（理化学研究所）、堀 敦史（理化学研究所）、高木 将通（理化学研究所）、小倉 崇浩（理化学研究所）、Balazs Gerofi（理化学研究所）
- **3.3 通信ライブラリ** 石川 裕（理化学研究所）、高木 将通（理化学研究所）
- **3.4 ストレージ＆ファイル I/O** 石川 裕（理化学研究所）
- **3.5 プログラミング環境** 村井 均（理化学研究所）
- **3.6 数値計算ライブラリ** 今村 俊幸（理化学研究所）
- **3.7 運用系ソフトウェア** 宇野 篤也 （理化学研究所）

第 4 章ターゲットアプリケーションのコデザイン

- **4.1 アプリケーションにおけるコデザインの概要** 富田浩文（理化学研究所）、三上和徳（理化学研究所）
- **4.2 重点課題 1 GENESIS のコデザイン** 小林千草（理化学研究所）、Jaewoon Jung（理化学研究所）
- **4.3 重点課題 2 Genomon のコデザイン** 鈴木 惣一朗（理化学研究所）、伊東 聰（東京大学）
- **4.4 重点課題 3 GAMERA のコデザイン** 藤田航平 (東京大学)
- **4.5 重点課題 4 NICAM+LETKF のコデザイン** 八代 尚（国立環境研究所）
- **4.6 重点課題 5 NTChem のコデザイン** 三上 和徳（理化学研究所）、河東田道夫（理化学研究所）
- **4.7 重点課題 6 ADVENTURE のコデザイン** 南 一生（理化学研究所）、宮本 健吾（理化学研究所）
- **4.8 重点課題 7 RSDFT のコデザイン** 黒田 明義 （理化学研究所）
- **4.9 重点課題 8 FrontFlow/blue のコデザイン** 南 一生（理化学研究所）、熊畑 清（理化学研究所）
- **4.10 重点課題 9 LQCD のコデザイン** 中村 宜文 （理化学研究所）

第 5 章 新たなる付加価値：PCI を使った将来拡張の可能性 佐野健太郎 （理化学研究所）

第 6 章 主な共同研究

- **6.1 東京大学** 中島 研吾（東京大学）
- **6.2 筑波大学** 朴 泰祐（筑波大学）
- **6.3 京都大学** 中島 浩 (京都大学)

カッコ内は、プロジェクト終了時または執筆時期における所属である。

謝辞

理化学研究所、富士通、文部科学省、共同研究において、本プロジェクトに参加・貢献された、すべての方々に、ここに感謝する。

注釈

- 本報告書において、メモリ容量は 2 のべき乗表現、2 進接頭辞 (MiB, GiB 等) とし、表現していないその他の数値は 10 のべき乗表現としている。
- システムの名称は、2019 年 5 月に決定した「富岳」を用いるが、特に開発中のシステムと区別したい場合には, 開発コード名「ポスト京」を用いている箇所があることをお断りしておく。

目 次

第 1 章　開発経緯 **21**

第 2 章　システム概要・性能 **25**
2.1 スーパコンピュータ富岳・システム概要 25
2.1.1 システム諸元 25
2.1.2 メニーコアプロセッサ A64FX 25
2.1.3 ネットワークとストレージ 28
2.2 基本性能 29
2.3 ベンチマークによる評価 33
2.3.1 SPEC ベンチマーク 33
2.3.2 オープンソース HPC アプリケーションの性能 33
2.3.3 HPL(TOP500), HPCG, HPL-AI, Graph500 34
2.4 ターゲットアプリケーション性能評価結果 36

第 3 章　システムのコデザイン **38**
3.1 CPU アーキテクチャ 38
3.1.1 アプローチ 38
3.1.2 コデザイン成果 40
3.2 オペレーティングシステム 51
3.2.1 アプローチ 51
3.2.2 コデザイン成果 52
3.3 通信ライブラリ 55
3.3.1 アプローチ 55
3.3.2 コデザイン成果 57
3.4 ストレージ＆ファイル I/O 59
3.4.1 アプローチ 59
3.4.2 コデザイン成果 60
3.5 プログラミング環境 70
3.5.1 アプローチ 70
3.5.2 コデザイン成果 70
3.5.3 FDPS 77
3.5.4 ツール 77
3.6 数値計算ライブラリ 79
3.6.1 アプローチ 79
3.6.2 コデザイン成果 81
3.7 運用系ソフトウェア 86
3.7.1 アプローチ 86

3.7.2 コデザイン成果 88

第 4 章 ターゲットアプリケーションのコデザイン 93
4.1 アプリケーションにおけるコデザインの概要 93
4.1.1 コデザイン手法とその対象 93
4.1.2 重点課題とターゲットアプリケーション 94
4.1.3 アプリケーションの計算手法と主なコデザイン観点 95
4.1.4 性能評価用ターゲット問題の設定 99
4.1.5 富岳向け性能推定の基本的方針 100
4.2 重点課題 1 GENESIS のコデザイン 101
4.2.1 アプリケーションの特徴とコデザインの方針 101
4.2.2 短距離相互作用計算での演算待ちの減少 102
4.2.3 短距離相互作用計算での浮動小数点ロード L2 アクセスの減少 102
4.2.4 短距離相互作用計算でのペアリストを使用しないアルゴリズムの開発 . . . 103
4.2.5 短距離相互作用計算での lookup table について 103
4.2.6 長距離非結合相互作用の高速化 104
4.2.7 PairList のメモリ減少について 104
4.2.8 Constraint のインダイレクトアクセス 104
4.2.9 コデザインによるシステムへの実装要請項目 105
4.2.10 短距離相互作用計算でのプリフェッチ最適化 105
4.2.11 ソフトウェア調整 (1) での長距離非結合相互作用の高速化 105
4.2.12 袖通信高速化 106
4.2.13 ボトルネック部分外を含む全区間のでの高速化 106
4.2.14 富岳向け性能推定 106
4.2.15 富岳で達成された性能倍率 106
4.3 重点課題 2 Genomon のコデザイン 107
4.3.1 アプリケーションの特徴とコデザインの方針 107
4.3.2 GFK を対象にしたコデザイン 109
4.3.3 Genomon2 におけるワークフロー制御とジョブスケジューリング 111
4.3.4 Genomon2 におけるファイル I/O 調査 113
4.3.5 アライメントプログラム BWA 117
4.3.6 変異検出・検証・フィルタリング計算に対する最適化 122
4.3.7 ファイルシステムの負荷削減 123
4.3.8 アライメント後処理計算の並列化 123
4.3.9 ワークフローレベルでの高速化 124
4.3.10 コデザインによるシステムへの実装要請項目 126
4.3.11 富岳向け性能推定 127
4.3.12 富岳上でのチューニングと性能評価 132
4.4 重点課題 3 GAMERA のコデザイン 138
4.4.1 アプリケーションの特徴とコデザインの方針 138
4.4.2 コデザイン前の EBE カーネル 138
4.4.3 EBE における SIMD 演算器の利用 139
4.4.4 EBE におけるカラリングによるリストアクセスの削減 139

4.4.5 EBE におけるループ分割によるスピルの削減 140
4.4.6 コデザインによるシステムへの実装要請項目 141
4.4.7 富岳向け性能推定 142
4.4.8 富岳で達成された性能倍率 142
4.4.9 さらなる高速化に向けた検討：時間並列アルゴリズム 143
4.5 重点課題 4 NICAM+LETKF のコデザイン 150
4.5.1 アプリケーションの特徴とコデザインの方針 150
4.5.2 NICAM におけるコデザイン 151
4.5.3 LETKF におけるコデザイン 161
4.5.4 NICAM+LETKF ワークフローにおけるコデザイン 163
4.5.5 コデザインによるシステムへの実装要請項目 163
4.5.6 富岳向け性能推定 168
4.5.7 富岳で達成された性能倍率 168
4.5.8 主筆者・執筆協力者 169
4.6 重点課題 5 NTChem のコデザイン 170
4.6.1 アプリケーションの特徴とコデザインの方針 170
4.6.2 RI-MP2 計算の超並列実装 170
4.6.3 RI-MP2 計算のメモリ分割実装 (1) 175
4.6.4 RI-MP2 計算のメモリ分割実装 (2) 176
4.6.5 RI-MP2 計算の単精度・倍精度の混合精度化アルゴリズムの検討 178
4.6.6 RI-MP2 計算における密行列データ圧縮によるネットワーク通信量削減の検討180
4.6.7 RI-MP2 計算の単精度・倍精度の混合精度化アルゴリズムのプロトタイプ実装および性能評価 181
4.6.8 RI-MP2 計算の単精度・倍精度の混合精度化アルゴリズムの数値検証 . . . 183
4.6.9 RI-MP2 計算における 3 中心積分計算の高速化 185
4.6.10 入力ファイル読み込みの高速化と選択切り替え 187
4.6.11 ターゲットジョブのプロセス構成再精査 187
4.6.12 計算と通信のオーバーラップの明示的な促進 188
4.6.13 PMlib 性能モニターライブラリの組み込み 188
4.6.14 試作機での小規模データ実行状況の確認 189
4.6.15 コデザインによるシステムへの実装要請項目 189
4.6.16 富岳向け性能推定 191
4.6.17 富岳で達成された性能倍率 199
4.6.18 執筆者 199
4.7 重点課題 6 ADVENTURE のコデザイン 201
4.7.1 アプリケーションの特徴とコデザインの方針 201
4.7.2 DomainFEM 区間の主要ルーチン MatVec_Product の高速化 203
4.7.3 コースグリッド修正区間の主要ルーチン CoarseMatVec の高速化 205
4.7.4 コデザインによるシステムへの実装要請項目 207
4.7.5 富岳向け性能推定 208
4.7.6 富岳での性能測定 211
4.7.7 執筆者 217
4.8 重点課題 7 RSDFT のコデザイン 219

4.8.1 アプリケーションの特徴とコデザインの方針 219
4.8.2 主要演算 (DGEMM)・通信性能 (集団通信) パラメータの最適化 219
4.8.3 メモリ分散アルゴリズムの検討 225
4.8.4 コデザインによるアーキテクチャへの要請 268
4.8.5 富岳向け性能推定 273
4.8.6 富岳で達成された性能倍率 283
4.9 重点課題 8 FrontFlow/blue のコデザイン 284
4.9.1 アプリケーションの特徴とコデザインの方針 284
4.9.2 勾配計算ルーチン (GRAD3X) 主要ループの SIMD 化・ソフトウェアパイプライン化、ロードストア数削減を目的としたアルゴリズム変更 285
4.9.3 発散計算ルーチン (FLD3X2) 主要ループの SIMD 適用、ストリーム数調整、セクタキャッシュの利用 288
4.9.4 運動方程式ルーチン (VEL3D1) 主要ループの高速化の検討 289
4.9.5 疎行列ベクトル積ルーチン (CALAX) 主要ループ 290
4.9.6 配列のコピーの除去 290
4.9.7 配列のゼロクリアの除去 291
4.9.8 非スレッド並列ルーチン (CRSCVA, CLRCRS, NODLEX, DGNSCL) のスレッド並列化 291
4.9.9 if を含むループのリストベクトル化 292
4.9.10 型変換ループの除去 292
4.9.11 その他改善検討 293
4.9.12 通信およびファイル I/O に関して 293
4.9.13 入力ファイル生成について 294
4.9.14 コデザインによるシステムへの実装要請項目 294
4.9.15 富岳向け性能推定 295
4.9.16 富岳での性能測定 300
4.9.17 執筆者 307
4.10 重点課題 9 LQCD のコデザイン 308
4.10.1 アプリケーションの特徴とコデザインの方針 308
4.10.2 QCD クォークソルバーの最適化 308
4.10.3 5 次元型フェルミオン行列の逆行列反復解法コード 337
4.10.4 核力計算コードの改善検討 337
4.10.5 コデザインによるシステムへの実装要請項目 338
4.10.6 京でのベースライン測定 339
4.10.7 富岳向け性能推定 340
4.10.8 富岳を用いた実測 341
4.10.9 主筆者・執筆協力者 344

第 5 章 新たなる付加価値：PCI を使った将来拡張の可能性 345
5.1 はじめに 345
5.2 拡張方式 345
5.3 拡張部アーキテクチャと試作システム 346
5.3.1 拡張部アーキテクチャの検討 346

5.3.2 試作システムの開発と実機試験 348
5.4 アプリケーションの検討 349
5.5 共同研究に向けた準備 351
5.6 おわりに 351

第 6 章 主な共同研究 **358**
6.1 東京大学 358
6.1.1 概要 358
6.1.2 プロセッサアーキテクチャ・電力制御技術 360
6.1.3 数値計算ライブラリ 372
6.1.4 システムソフトウェア 380
6.1.5 今後の展望 382
6.2 筑波大学 387
6.2.1 概要 387
6.2.2 タスク並列モデルに基づくマルチコア・アーキテクチャのプログラミングと性能評価に関する研究 387
6.2.3 FPGA と GPU を混載した複合型演算加速システムにおける OpenACC プログラミング環境に関する研究 391
6.2.4 FPGA 間高性能通信システムに関する研究 396
6.3 京都大学 400
6.3.1 概要 400
6.3.2 富士通製コンパイラの評価 400
6.3.3 まとめ 419

図 目 次

1.1 開発スケジュール 21

2.1 A64FX プロセッサのブロック図 26
2.2 A64FX プロセッサのチップパッケージとダイ写真 27
2.3 SPEC ベンチマークの結果 33
2.4 オープンソース HPC アプリケーションの性能 (Xeon との相対性能) 34

3.1 性能推定ツールの使い方 39
3.2 O3 資源量を変えたときのカーネルの性能とダイサイズへの影響 (Size and performance are shown in %, relative to "Base Set") 46
3.3 Stream ベンチマークにおけるエコモードとコアリテンションの効果 49
3.4 DGEMM ベンチマークにおけるブーストモードとコアリテンションの効果 50
3.5 McKernel 51
3.6 ノード内テンポラリファイルシステム例 62
3.7 ノード内テンポラリファイルシステム例 64
3.8 グローバルファイルシステム例 65
3.9 共通ファイル読み込み高速化例 68
3.10 MPI-IO 向け LLIO 69
3.11 富岳における Impact3D の性能向上と京コンピュータとの比較 71
3.12 富岳における CCS QCD の性能向上と京コンピュータとの比較 72
3.13 富岳における NTChem-MINI の性能向上と京コンピュータとの比較 72
3.14 `tasklet`, `taskletwait`, `tasklets` 指示文の文法 74
3.15 `tasklet gmove` と `tasklet reflect` 指示文の文法 75
3.16 `tasklet` と `tasklet gmove` 指示文の例 75
3.17 2.0GHz モードのシングル CMG における [DSH]GEMM 性能（Gflop/s) 82
3.18 A64FX のシングルコアにおける 1d-FFT カーネル性能ベンチマーク結果 83
3.19 富岳における EigenExa2.6c の総計算時間 (経過時間の単位: 秒) 84
3.20 N= 131,072 の場合の 3 つのカーネルルーチンの時間内訳 84
3.21 Preliminary benchmark results of Batched BLAS kernels on an AFX64 with a single CMG binding 85
3.22 2.5PD-GEMM の実装方法 86
3.23 Strong scaling performance of 2D-compatible 2.5D-PDGEMM on the supercomputer Fugaku (on the pre-evaluation environment). 87

4.1 GFK ワークフロー全体のパイプライン実行 110

4.2 Genomon2 DNA 解析パイプラインのワークフロー（赤枠内が富岳ターゲット問題としての評価対象） 112
4.3 京、1 スレッドでの BWA-MEM 実行時間（秒） 119
4.4 FX100、1 スレッドでの BWA-MEM 実行時間（秒） 119
4.5 BWA SIMD 化部分（KT_FOR2 区間）実行時間（秒）
「SVE*」は富岳試作機、「NO SIMD」「HPC-ACE*] は FX100(1.975GHz) 上で実行。全て 8 スレッド実行。 121
4.6 BWA アライメント計算部分 (KT_FOR1 + KT_FOR2 区間) 実行時間（秒）
「SVE*」は富岳試作機上で 12 スレッド、「NO SIMD」「HPC-ACE*] は FX100(1.975GHz) 上で 8 スレッド実行。 121
4.7 Genomon2 ワークフロー: 詳細設計（３）終了時 124
4.8 Genomon2 ワークフロー: 改良 1 125
4.9 Genomon2 ワークフロー: 改良 2 126
4.10 Task3A「split_files」最も時間のかかった VGE ジョブの実行時間内訳（秒） . . . 133
4.11 Task6A「identify_mutations(i.b.)」Task8A「merge_bam(m.b.)」最も時間のかかった VGE ジョブの実行時間内訳（秒） 134
4.12 平均消費電力 (左図) と最大消費電力 (右図) のノード種別の平均: CN=IO ノード非兼任な計算ノード (83 個)、BIO=BIO ノード (6 個)、SIO=SIO ノード (6 個)、GIO=GIO ノード (1 個)、ALL=全計算ノード (96 個)。棒クラフ長と数値はノード間平均値、エラーバーはノード間での最小値〜最大値を表す。 135
4.13 100 塩基対データに対する Genomon2 主要タスク実行時間（秒）: 「京」はソフトウェア調整（１）での性能推定時のコード、「ポスト京 (LLIO 無)」「ポスト京 (LLIO 有)」では最終版コードを使用。 137
4.14 コデザイン前の EBE カーネル 139
4.15 SIMD 演算器を利用した EBE カーネル 140
4.16 EBE カーネルにおけるカラリングによるリストアクセスの削減 146
4.17 一般的なカラリング手法 146
4.18 コデザインにおいて開発したカラリング手法 147
4.19 GAMERA におけるインバランス係数の推定方法 147
4.20 GAMERA の富岳におけるウィークスケーリング性能の測定結果 148
4.21 時間並列積分による EBE カーネルにおける non-SIMD 計算の削減 149
4.22 NICAM3.5km メッシュ傾圧波理想実験における倍精度ー単精度実験の差 156
4.23 富岳での NICAM のウィークスケーリング性能計測結果 162
4.24 富岳での LETKF の性能計測結果 164
4.25 富岳での LETKF(14km メッシュ) の計測時間内訳 165
4.26 RI-MP2 計算の 1 次元並列化 (従来法) と 2 次元 MPI 階層並列化 (新手法) 171
4.29 3 中心積分データ通信の P2P 通信法からリング通信法への変更 171
4.27 RI-MP2 計算の 2 次元 MPI 階層並列化時の行列データ並列化 172
4.28 RI-MP2 計算の 2 次元 MPI 階層並列化時の行列ブロックデータの通信パターン . . 172
4.30 3 中心積分データ通信の P2P 通信法からリング通信法への変化よる通信時間削減効果173
4.31 2048 ノード 2 次元 MPI 階層並列化時の演算時間と通信時間の内訳 174
4.32 行列のコレスキー分解のメモリ分割並列実装 176
4.33 上三角行列逆行列計算のメモリ分割並列実装 177

4.34 密行列-密行列積演算の混合精度計算アルゴリズム 179
4.35 コロネン 2 量体 $(C_{24}H_{12})_2$ の RI-MP2/def2-SVP 計算における 3 中心積分行列の要素分布 180
4.36 富岳性能推定用想定問題 204
4.37 MulMatVec のチューニングの概要 212
4.38 MulMatVec の CPU 解析レポートの結果 212
4.39 MatVec_Rect_Quad のチューニングの概要 (1) 214
4.40 MatVec_Rect_Quad のチューニングの概要 (2) 214
4.41 富岳性能測定結果と性能推定結果 215
4.42 富岳の対「京」性能倍率 217
4.43 (旧) 代表プロセス方式の考え方 221
4.44 RSDFT のバンド方向のデータ転送 222
4.45 (新) 代表プロセス方式の考え方 224
4.46 (新) 代表プロセス方式の Reduce の処理 224
4.47 (改新) 代表プロセス方式の Reduce の処理 225
4.48 LU による直交化おける実行時間の内訳の変化 229
4.49 LU 分解における上三角行列の並列分散方法、左図：RSDFT 内部の並列分散方法、右図：ScaLAPACK での並列分散方法 230
4.50 TGS/チューニング性能（空間並列数変化/No.1-1,1-5） 233
4.51 TGS/通信時間（空間並列数変化） 233
4.52 TGS/チューニング性能（ブロックサイズ変化/No.2-2,2-5） 234
4.53 TGS/チューニング性能（バンド並列数変化/No.3-3,3-5） 235
4.54 FGS/チューニング性能（バンド並列数変化） 237
4.55 FGS/チューニング性能（メモリ分散による性能の変化） 237
4.56 FGS/対角優先処理シーケンス (ランク 0) 238
4.57 SD/メモリ分散版の性能（空間並列数変化/No.1-1、1-5） 239
4.58 SD メモリ分散版の性能（バンド並列数変化/No.3-3,3-5） 240
4.59 EigenExa のスケーラビリティ。（左図：区間ごと、右図：演算と通信） 241
4.60 EigenExa のバージョンによる性能の違い。 242
4.61 EigenExa のプロセス配置による性能の違い。 242
4.62 SD-EigenExa 連携の性能。左図：RSDFT の分割数を変化。右図：EigenExa の分割数を変化。 244
4.63 trans_blkcy2cy のロードインバランス 244
4.64 今まで評価を行っていた Tofu 専用アルゴリズムの通信概要：左図：二次元 Trinaryx3、右図：一次元 Simple Spread 245
4.65 一次元 Bintree3D アルゴリズムの通信の概要 245
4.66 dc_regist の処理イメージ 246
4.67 CG の処理別経過時間（空間並列数変化）左図：メモリ分散無し、右図：メモリ分散有り 247
4.68 CG の処理別経過時間（バンド並列数変化）左図：メモリ分散無し、右図：メモリ分散有り 247
4.69 CG の処理別経過時間（バンドまとめ数変化）左図：メモリ分散無し、右図：メモリ分散有り 248

4.70 Excel ベースの CPMD メモリ使用量簡易シミュレータ 251
4.71 CPMD のターゲット問題 (SiO_2) とその動作パラメータ 252
4.72 CPMD のターゲット問題 (SiC-SiO_2 界面問題) 252
4.73 ROTORB のメモリ分散による実行時間の変化。左：MLBK=72、右：MLBK=180。253
4.74 ROTORB のバンド分割数に関する weak scaling 測定結果 254
4.75 ROTOORB のグリッド分割数に関する strong scaling 測定結果 254
4.76 CPMD のスケーラビリティの測定パターン (Si 4,096 原子) 255
4.77 CPMD のスケーラビリティ測定結果 (Si 4,096 原子) 255
4.78 京を用いたアプリ全体のメモリ消費量と実行時間の測定結果 (SiO_2) 257
4.79 詳細設計 (4) での実機検証との DGEMM 性能比較。赤枠内の性能が異なる。 . . . 259
4.80 富岳共用前評価環境 I(左図),II(右図) での SCF 実行時間のぶれ 260
4.81 富岳共用前評価環境での小規模フルアプリケーションのスケーラビリティ 260
4.82 ターゲット問題の半分の原子数でのメモリ使用量プロファイル結果。allocate 処理からの見積もりと/proc からの実測を比較した。 265
4.83 EigenExa のプロセスマッピングによる通信時間の変化。 266
4.84 1 秒精度のラック電力を用いた、RSDFT 1SCF の消費電力の変化。 268
4.85 ターゲット問題を測定した時の 1 分精度の全ラック電力の消費電力の変化。 268
4.86 TGS の従来方式（対角優先）と新規方式（対角移動）での処理順序。 276
4.87 一般的な GS 手法、TGS 従来方式（対角優先)、TGS 新規方式（対角移動）のプロセスごとの処理順序の例。 276
4.88 GRAD3X の処理内容とストアの様子 286
4.89 GRAD3X のループ構造 286
4.90 ループ構造変更前後のストアの様子 287
4.91 FLD3X2 の改変 289
4.92 GRAD3X のチューニング (1) 300
4.93 GRAD3X のチューニング前性能 301
4.94 GRAD3X のチューニング後性能 301
4.95 GRAD3X のチューニング (2) 前 302
4.96 GRAD3X のチューニング (2) 後 302
4.97 GRAD3X のチューニング (2) 前の性能 303
4.98 GRAD3X のチューニング (2) 後の性能 303
4.99 FIELD3X のチューニング 304
4.100FIELD3X のチューニング前性能 304
4.101FIELD3X のチューニング後性能 305
4.102富岳の FFB 主要カーネルのメモリバンド幅値 (ノード) 305
4.103富岳の FFB 主要カーネルの性能値 (ノード) 306
4.104富岳の FFB の並列性能 306
4.105富岳の対「京」性能倍率 307
4.106__mult_clvs の最適化前のコード 313
4.107__mult_clvs の最適化後のコード 314
4.108LQCD の X 方向のステンシル差分計算の概念図（最適化前） 315
4.109LQCD の X 方向のステンシル差分計算の概念図（最適化後） 315
4.110LQCD の命令レベルの並列性を高めるための最適化前後のソースコード 316

4.111LQCD の混合精度ソルバーの残差を反復回数の関数としてプロットした図。 . . . 319
4.112ダブルバッファリングの状態遷移を表す図。左端は、富士通 RDMA を用いた時の利用関数。 . . . 323
4.113LQCD のウィークスケーリング（QCDJDD 解法 1 反復)。158976 ノードは富岳の全ノード数である。147456 ノードの縦線は、LQCD のターゲット問題を実行するノード数を示すためのものである。 . . . 343

5.1 ポスト京ラックと PCI が利用可能なシャーシ . . . 346
5.2 FPGA の構造 . . . 347
5.3 ポスト京拡張部のアーキテクチャ . . . 347
5.4 FPGA クラスタ試作システム ESSPER とポスト京との接続 . . . 348
5.5 実験システムのシステムスタック . . . 349

6.1 評価における全体的なワークフロー . . . 360
6.2 計測電力とモデルによる予測値の比較 . . . 361
6.3 浮動小数点演算に要するエネルギーの算出用コード . . . 363
6.4 評価のスコープ . . . 365
6.5 SIMD 幅及び FP パイプライン数のスケーリング . . . 365
6.6 STREAM/DGEMM 実行時の電力比較及びスケーリングによる電力削減効果 . . . 366
6.7 面積比較及びスケーリングによる面積削減効果 . . . 366
6.8 SIMD1 要素の操作に要するエネルギー (倍精度) . . . 367
6.9 電力バジェットに対するコア数のスケーリング効果 . . . 368
6.10 面積バジェットに対するコア数のスケーリング効果 . . . 368
6.11 SIMD 幅，FP パイプライン幅のスケーリングに応じた電力の変化 . . . 369
6.12 SIMD 幅，FP パイプライン幅のスケーリングに応じた面積の変化 . . . 369
6.13 コア設定毎の面積対性能，電力対性能の比較 . . . 370
6.14 コア設定毎の面積対性能，電力対性能の比較 (Ideal) . . . 370
6.15 TSQR の実行時間と消費エネルギー . . . 371
6.16 TSQR の平均消費電力 . . . 372
6.17 TSQR における資源利用率 . . . 373
6.18 不均質多孔質媒体中の三次元地下水流れ (a) 透水係数分布，(b) 流線分布 . . . 374
6.19 並列多重格子法と Coarse Grid Solver（参考文献 [28]) . . . 375
6.20 様々な疎行列格納法（a）CRS,（b）ELL,（c）Sliced ELL,（d）SELL-C-σ（参考文献 [29] [30]) . . . 376
6.21 各疎行列格納法における前進代入部分の記述 . . . 377
6.22 OFP 1 ノードによる MGCG 法計算時間，HB 4×16 . . . 378
6.23 SCS-b 適用時の MGCG 法計算時間 . . . 379
6.24 MGCG 法部分の SCS-a (:a)，SCS-b (:b) による *Sliced ELL* に対する改善率 . . . 380
6.25 三次元ポアソン方程式ソルバーの解析対象：差分格子の各メッシュは直方体（辺長さは ΔX,ΔY,ΔZ），X,Y,Z 各方向のメッシュ数は NX,NY,NZ . . . 380
6.26 混合精度演算（前進代入部分）(a) 倍精度＋単精度前処理 (D-S), (b) 倍精度＋半精度前処理 (D-H), (c) 単精度＋半精度前処理 (S-H), 緑字：倍精度，赤字：単精度，青字：半精度 . . . 381

6.27 Bottom 点（図 6.25 の原点）での（D-D）の結果との相対誤差，D-S，D-H は解が得られる範囲では誤差は無い 382
6.28 各ケースにおける反復回数（D-D の反復回数で無次元化） 383
6.29 前進代入のサンプルコード 384
6.30 Base 実装の Chunk size に対する計算時間 384
6.31 Opt1 実装の Chunk size に対する計算時間 384
6.32 Opt2 実装の Chunk size に対する計算時間 384
6.33 IC 前処理部の計算時間の比較 385
6.34 ICCG 法の全体の計算時間の比較 385
6.35 IHK/McKernel 概要・システムコール移譲機能概略 参考文献 [32] 385
6.36 Performance Improvement of MGCG solver with SCS-b by IHK/McKernel . . . 386
6.37 Xeon Broadwell CPU を用いた評価 388
6.38 タスク実行の流れ 389
6.39 Xeon Skylake CPU を用いた評価 390
6.40 Xeon Phi KNL を用いた評価 390
6.41 EPYC Naples を用いた評価 391
6.42 ThunderX2 を用いた評価 391
6.43 A64FX を用いた評価 392
6.44 プログラミング環境の概要 392
6.45 MHOAT の処理の流れ 393
6.46 拡張ディレクティブの使用例 394
6.47 追加する機能のイメージ 394
6.48 行数の比較 395
6.49 文字数の比較 396
6.50 CIRCUS の通信性能評価 397
6.51 ART 法の ray の動き 398
6.52 CIRCUS を使った ART の実装 398
6.53 ART on FPGA の性能評価 399
6.54 `particle_push()` のカーネルループ 401
6.55 F-1912 が生成する `particle_push` ループの概念的コード 402
6.56 `ppush()` のカーネルループ 1 404
6.57 `ppush()` のカーネルループ 2 411
6.58 `cscat()` のカーネルループ 414

表 目 次

2.1 諸元 . . . 25
2.2 A64FX プロセッサの仕様 . . . 26
2.3 ストレージの仕様 . . . 28
2.4 基本性能 . . . 29
2.5 コンパイラ性能 . . . 30
2.6 OpenMP 性能 . . . 30
2.7 数学ライブラリ性能 . . . 30
2.8 MPI 通信ライブラリ性能 . . . 31
2.9 OS およびファイル IO およびシステム管理性能 . . . 32
2.10 評価したオープンソース HPC アプリケーションの概要とパラメータ . . . 35
2.11 富岳全体を用いたベンチマークの性能 . . . 35
2.12 ターゲットアプリケーションの対京性能倍率と消費電力 . . . 36
2.13 ターゲットアプリケーションの電力制御各モードでの比較 . . . 37

3.1 ダイとパッケージ、マルチチップモジュールのコスト . . . 42
3.2 64 コアの場合、4 CMG と 8 CMG の構成のカーネルの性能 . . . 42
3.3 L1 キャッシュのラインサイズを変化させた時のカーネルの性能 (Normalized to the Estimated Performance of 256 Bytes) . . . 44
3.4 L1/L2 キャッシュサイズとウェイ数を変化させたときのカーネルの性能 (Normalized to the Estimated Performance of L1:64KiB4Way and L2:6MiB24Way) . . . 44
3.5 DGEMM と Stream によるパワーノブの評価 . . . 47
3.6 ターゲットアプリの電力モードによる電力予測 (性能と電力はノーマルモード (N) を 1.0 とした相対値) . . . 48
3.7 メモリ領域ごとのラージページ利用可否と利用可能サイズ . . . 52
3.8 対応開始時 (2019/6) と最終結果 (2021/2) のビルド可能率比較 . . . 53
3.9 ファイル共有関係・生存期間とファイルシステム . . . 60
3.10 LLIO が提供するデバイス名 . . . 61
3.11 階層ストレージマウント例 . . . 61
3.12 Lustre 2.10 に含まれる富士通拡張機能 . . . 66
3.13 富士通拡張機能 . . . 66
3.14 読み込み方式によるキャッシュおよびデータ転送先の違い . . . 67
3.15 プログラミング環境に関する要望と対応 (抜粋) . . . 70
3.16 サポートするプログラミング言語の規格/仕様 . . . 76
3.17 本体システム起動時間の改善結果 . . . 89
3.18 本体システム全体のパッチ適用時間 . . . 89
3.19 統計情報への追加項目結果 . . . 91

4.1 重点課題とターゲットアプリケーション 95
4.2 計算・通信の種類によるアプリケーションの分類 98
4.3 重点課題ターゲットアプリケーションの問題設定 99
4.4 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (GENESIS) . . 105
4.5 GENESIS の対京性能倍率と消費電力 106
4.6 GENESIS の電力制御各モードでの比較 106
4.7 Genomon2 入出力ファイル一覧（現在の問題規模に対する 1 検体解析） 114
4.8 Genomon2 入出力ファイル一覧（富岳ターゲット問題に対する 1 検体解析） . . . 115
4.9 第 1 階層ストレージへの理想的なファイル配置時のストレージ消費量 116
4.10 第 1 階層ストレージ消費量 (ファイル配置変更後) 117
4.11 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (Genomon) . . 127
4.12 Genomon(GFK) の性能評価結果 (詳細設計（２）旧問題規模) 128
4.13 Genomon2 アライメントプログラム BWA の性能評価結果 (詳細設計（２）) . . . 129
4.14 Genomon2 予備調査により得られた予想実行性能と使用ノード数の関係 130
4.15 Genomon2 性能推定 130
4.16 Task3A「split_files」最も時間のかかった VGE ジョブの実行時間内訳（秒） . . . 133
4.17 Task6A「identify_mutations」Task8A「merge_bam」最も時間のかかった VGE ジョブの実行時間内訳（秒） 134
4.18 富岳での 100 塩基長データ解析時の計算性能 135
4.19 富岳での実測値から推定したターゲット問題計算性能 136
4.20 Genomon の対京性能倍率と消費電力 136
4.21 Genomon の電力制御各モードでの比較 136
4.22 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (GAMERA) . 141
4.23 GAMERA の京・富岳におけるインバランス係数の推定結果 142
4.24 GAMERA の対京性能倍率と消費電力 143
4.25 GAMERA の電力制御各モードでの比較 143
4.26 GAMERA の富岳における性能測定に用いたモデルセット。A-ポスト京はターゲット問題、A-京はターゲット問題を京全系で計測した際のモデル分割を示す。 . . . 144
4.27 NICAM 力学カーネルの倍精度・単精度での性能 152
4.28 最適化後の NICAM 力学カーネルの性質のまとめ 154
4.29 NICAM カーネル区間のアプリケーション全体に対する代表性 157
4.30 NICAM の実行時間見積もりとその内訳 159
4.31 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (NICAM+LETKF) 167
4.32 NICAM+LETKF の対京性能倍率と消費電力 169
4.33 NICAM+LETKF の電力制御各モードでの比較 169
4.34 ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ の RI-MP2/cc-pVTZ 計算 (240 原子、6432 原子軌道) の「京」での計算時間、並列性能 173
4.35 ナノグラフェン 2 量体 $(C_{150}H_{30})_2$ の RI-MP2/cc-pVTZ 計算 (360 原子、9840 原子軌道) の「京」での計算時間、並列性能ならびに実行性能 174
4.36 メモリ分割版と非分割版の「京」での実行性能：ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ の RI-MP2/cc-pVTZ 計算 (240 原子、6432 原子軌道)、「京」1536 ノード使用 . . . 178
4.37 コロネン 2 量体 $(C_{24}H_{12})_2$ の RI-MP2/def2-SVP 計算における 3 中心積分行列の要素分布 180

4.38 RI-MP2 倍精度版コードおよび混合精度版コードを用いた際の計算精度、実行時間、性能倍率 (Intel Xeon Haswell 環境、RI-MP2/cc-pVTZ 計算) 181
4.39 RI-MP2 倍精度コードおよび混合精度コードを用いた際の演算・通信区間の実行時間と性能倍率 (バッキーキャッチャー $C_{60}@C_{60}H_{28}$ RI-MP2/cc-pVTZ 計算) 182
4.40 計算精度モデルと採用する行列積のタイプの対応 182
4.41 混合精度モデル (SP123a, SP123b, SP23, SP3) における倍精度モデル DP からの数値誤差と計算加速度 183
4.42 データセットを用いた RI-MP2 分子間相互作用エネルギー計算における倍精度モデル (DP) と各混合精度モデルとの平均誤差絶対値 (kcal/mol) 184
4.43 ターゲット問題の RI-MP2 相関エネルギー (a.u.) と倍精度 (DP) との誤差 (kcal/mol) 185
4.44 3 中心積分計算の高速化 187
4.45 ターゲットジョブのプロセス構成再精査 (全て SP3 計算精度モデル) 188
4.46 ターゲットジョブの通信統計情報 188
4.47 富岳で利用可能な PMlib レポート機能 189
4.48 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (NTChem) . . 191
4.49 NTChem/RI-MP2 ベースライン版コードの演算区間と主な処理の概略 191
4.50 NTChem/RI-MP2 ベースライン版コードの通信区間と主な処理の概略 193
4.51 切り出しカーネルコード 194
4.52 カーネルコードによる京ベースライン推定と実測された結果 194
4.53 通信区間の通信パターンと富岳の推定値 196
4.54 NTChem/RI-MP2 コデザイン最新版のターゲット問題通信区間の通信パターン . . 197
4.55 NTChem の富岳推定計算時間 (通常モード、電力制御なし、プロセス数 82928) . . 197
4.56 4 中心分子積分計算における行列積のサイズ及び回数の測定値（プロセス番号 0） 199
4.57 NTChem の対京性能倍率と消費電力 199
4.58 NTChem の電力制御各モードでの比較 200
4.59 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (ADVENTURE) 207
4.60 京の 32768 ノード（32768 プロセス）での実行時間 208
4.61 富岳の 4096 ノード（16384 プロセス）での実行時間（通常モード、エコモード無効） 209
4.62 性能推定時の区間と性能測定時の区間対応 216
4.63 RSDFT の性能改善 220
4.64 RSDFT の通信性能改善 1 222
4.65 RSDFT の通信性能改善 2 222
4.66 RSDFT の通信性能改善 3、4 223
4.67 RSDFT の通信性能改善 5 225
4.68 RSDFT 新アルゴリズムの性能（2015 年度） 226
4.69 RSDFT 新アルゴリズムの検討状況 (2017 年度) 227
4.70 LUGS の性能（プロセス範囲の変化） 228
4.71 RSDFT メモリ分散の試作・評価の状況 (2016 年度) 231
4.72 TGS/メモリ分散版の性能測定ケース 231
4.73 TGS/オリジナル版とメモリ分散版の比較 232
4.74 TGS/チューニングの性能測定ケース（空間並列数変化） 232
4.75 TGS/チューニング性能（空間並列数変化/No.1-5） 233
4.76 TGS/チューニングの性能測定ケース（ブロックサイズ変化） 233

4.77 TGS/チューニング性能（ブロックサイズ変化/No.2-2） 234
4.78 TGS/チューニングの性能測定ケース（バンド並列数変化） 234
4.79 TGS/チューニング性能（バンド並列数変化/No.3-5） 235
4.80 TGS/並列効果（バンド並列数変化） 235
4.81 FGS/評価パラメータ（バンド並列数変化） 236
4.82 SD メモリ分散版の性能測定ケース（空間並列数変化） 237
4.83 SD/メモリ分散版の性能（ブロックサイズ変化） 239
4.84 SD メモリ分散版の性能測定ケース（バンド並列数変化） 240
4.85 SD メモリ分散版/並列効果（バンド並列数変化） 241
4.86 Subspace-diag と ROTORB の処理の比較 249
4.87 ROTORB の実行時間の測定結果：単位 [s] 250
4.88 ROTORB のメモリ分散前後の測定パターン 253
4.89 ROTORB のバンド分割数に関する weak scaling 測定パターン 254
4.90 ROTORB のグリッド分割数に関する strong scaling 測定パターン 254
4.91 京と富岳のメモリ使用量の比較、青罫は比較的同規模での計算に相当する。 263
4.92 代表プロセス方式における見積もりと実測の通信性能の比較。 267
4.93 アプリケーション検討会で議論されたシステムへの実装要請一覧表 (RSDFT) . . . 271
4.94 RSDFT の性能予測結果 (基本設計時) 274
4.95 RSDFT の性能予測結果 (詳細設計 (1)：パラメータ修正・コデザイン検討課題 (3) の 1) 反映後) 274
4.96 RSDFT の性能予測結果 (詳細設計 (2)：2016 年度ハードウェアパラメータ見直し後)274
4.97 RSDFT の性能予測結果 (詳細設計 (3) 開発期間 I：2017 年度前半性能評価精緻化による予測) 277
4.98 EigenExa の逆変換 (TRDBAK) ブロックで使用する代表的な DGEMM パラメータ 278
4.99 EigenExa の逆変換 (TRDBAK) ブロックの試作版 DGEMM による性能 279
4.100RSDFT の性能予測結果 (詳細設計 (3) 開発期間 II：2017 年度後半性能評価による予測) 279
4.101DGEMM の試作機 CPU 測定性能 280
4.102RSDFT の性能予測結果 (詳細設計 (4)：2018 年度性能評価による予測) 280
4.103プロセスあたりのグリッド数 16, 17 での DGEMM カーネルの実行時間の比較。 . 281
4.104バンド分割を不連続な 3 分割から CMG 内の 4 分割に変更した時の性能の変化。 . 282
4.105RSDFT の性能予測結果 (ソフトウェア調整 (2)：2020 年度実機による最終測定性能)282
4.106RSDFT の対京性能倍率と消費電力 283
4.107RSDFT の電力制御各モードでの比較 283
4.108アプリケーション検討会で議論されたシステムへの実装要請一覧表 (FFB) 295
4.109FrontFlow/blue 京ベースライン実行時間 10 万タイムステップ 298
4.110FrontFlow/blue 富岳実行時間と電力性能 10 万タイムステップ (エコモード無効) . 299
4.111FrontFlow/blue 富岳実行時間と電力性能 10 万タイムステップ (エコモード有効) . 300
4.112FrontFlow/blue 富岳の対京性能向上比 300
4.113LQCD 測定区間一覧 308
4.114LQCD 測定区間一覧（ポスト京試作機用） 309
4.115LQCD のダブルバッファリングにおける、各ステップでの使用関数 322
4.116LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$) 329

4.117LQCD の TNI 割り当てと通信量 (プロセスあたり格子サイズ 32 × 6 × 6 × 2) . . . 329
4.118LQCD の隣接通信のリンクあたり通信量 (プロセスあたり格子サイズ 32 × 6 × 6 × 2)330
4.119LQCD の隣接通信の TNI あたり通信量 (プロセスあたり格子サイズ 32 × 6 × 6 × 2) 331
4.120LQCD の軸と Tofu の軸の対応 (検証用、プロセスあたり格子サイズ 32 × 6 × 6 × 2) 331
4.121LQCD の TNI 割り当てと通信量 (検証用、プロセスあたり格子サイズ 32 × 6 × 6 × 2)332
4.122LQCD の隣接通信のリンクあたり通信量 (検証用、プロセスあたり格子サイズ 32 × 6 × 6 × 2) 332
4.123LQCD の隣接通信の TNI あたり通信量 (検証用、プロセスあたり格子サイズ 32 × 6 × 6 × 2) 333
4.124LQCD の軸と Tofu の軸の対応 (検証用規模縮小版、プロセスあたり格子サイズ 32 × 6 × 6 × 2) 333
4.125LQCD の隣接通信の通信量の実測値（プロセスあたり格子サイズ 32 × 6 × 6 × 2） 333
4.126LQCD の隣接通信の実行時間の実測値（プロセスあたり格子サイズ 32 × 6 × 6 × 2）334
4.127LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ 32 × 6 × 4 × 3) 334
4.128LQCD の TNI 割り当てと通信量 (プロセスあたり格子サイズ 32 × 6 × 4 × 3) . . . 334
4.129LQCD の隣接通信のリンクあたり通信量 (プロセスあたり格子サイズ 32 × 6 × 4 × 3)335
4.130LQCD の隣接通信の TNI あたり通信量 (プロセスあたり格子サイズ 32 × 6 × 4 × 3) 335
4.131LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ 32 × 6 × 4 × 3、隣接通信 36 ラック検証用) 336
4.132LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ 32 × 6 × 4 × 3、隣接通信 3 ラック検証用) 336
4.133LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ 32 × 6 × 4 × 3、隣接通信 432 ノード検証用) 336
4.134LQCD の軸と Tofu の軸の対応（1MPI プロセスで 4CMG） 336
4.135LQCD の隣接通信区間における各リンクの転送量（1MPI プロセスで 4CMG） . . 337
4.136LQCD の隣接通信区間における各 TNI の転送量 337
4.137アプリケーション検討会で議論されたシステムへの実装要請一覧表 (LQCD) 339
4.138LQCD の京でのベースライン測定結果 340
4.139LQCD の富岳における通信区間の推定実行時間。単位はミリ秒。 341
4.140LQCD の富岳試作機を用いた性能測定結果。 341
4.141LQCD の単体性能（QCDJDD 解法 500 反復)。 342
4.142LQCD の単体性能（QCDJDD 解法 500 反復中の ddd_in_s_ 区間)。 342
4.143IMB–MPI1 を用いた Allreduce ベンチマークテスト結果。48 × 12 × 48 ノードを用いて、Tofu ハードウェアによる 3 要素までの MPI_Allreduce の高速処理機構が「あり」、「なし」でそれぞれ測定。繰り返し回数は 10000 回で、最小 (min)、最大 (max)、平均 (avg) 時間をそれぞれ示す。byte は MPI_Allreduce1 回あたりのメッセージ長で、count は要素数、MPI_Allreduce の data type は MPI_FLOAT である。343
4.144LQCD の経過時間内訳（QCDJDD 解法 1 反復)。 344
4.145LQCD の対京性能倍率と消費電力 344
4.146LQCD の電力制御各モードでの比較 344

5.1 FPGA クラスタにより付加価値が得られそうな計算問題の候補 350

6.1 FX700 システム構成 362

6.2 推定されたメモリ電力パラメタ 363
6.3 推定された倍精度浮動小数点演算にかかる各種エネルギー 364
6.4 FP リソースのスケーリング評価に用いる設定 366
6.5 混合精度演算の効果 378
6.6 各データ型、実装での最適な Chunk size C 384
6.7 Cygnus の実行環境 395
6.8 `ppush()` カーネルループ 1 の命令数比較 406
6.9 `ppush()` カーネルループ 1 のレジスタ割付 407
6.10 `ppush()` カーネルループ 1 のスカラレジスタ割付 408
6.11 `ppush()` カーネルループ 1 の `eex` 等の計算での FMA 命令 408
6.12 `ppush()` カーネルループ 1 の remainder loop の冗長 spill-in/out 410
6.13 `ppush()` カーネルループ 2 の命令数比較 412
6.14 `ppush()` カーネルループ 2 のベクトルレジスタ割付 413
6.15 `ppush()` カーネルループ 2 の remainder loop の冗長 spill-in 414
6.16 `cscat()` カーネルループの命令数比較 415
6.17 `cscat()` カーネルループのベクトルレジスタ割付 416
6.18 `cscat()` カーネルループの remainder loop の冗長 spill-in 416
6.19 `jv[][][][]` 更新のための命令数 418
6.20 問題点と改善状況 418

第1章　開発経緯

富岳の開発は、プロジェクト開始4年前の2010年、東京大学、筑波大学、東京工業大学、京都大学とともに、草の根的に始めた「戦略的高性能計算システム開発に関するワークショップ (SDHPC：Strategic Direction/Development of HPC)」までさかのぼることができる。当時、2009年秋の事業仕分け後、文部科学省が「京」を中核とする各大学情報基盤センター等のスーパーコンピュータを「革新的ハイパフォーマンス・コンピューティング・インフラ（HPCI)」と位置づけ、多様な利用者のニーズに応える計算環境の構築を開始した時期である。「京」の開発に目途が立ち、また筑波大学、東京大学、京都大学が共同研究の結果策定したT2Kオープンスーパーコンピュータ仕様に基づくマシンが稼働していた。一方、国際連携では、IESP（International Exascale Software Project）[7]が2008年に立ち上がり、エクサスケールマシンに向けた技術課題やロードマップの議論が開始された。

SDHPCで画期的だったのは，以下の二点であった。

- アプリケーション・システムソフトウェア・ハードウェアの若手研究者および若手開発者を一堂に集めて議論できたこと。
- 所属組織の立場ではなく、個人の立場で議論できたこと。

その後，文部科学省HPCI計画推進委員会 [15] 傘下に「今後のHPC技術の研究開発のあり方検討ワーキンググループ」が立ち上がり、その下にアプリケーション作業部会およびコンピュータアーキテクチャ・コンパイラ・システムソフトウェア作業部会が設置され [17]、SDHPCは後者の作業部会のアクティビティと統合した。これら2つの作業部会により「今後のHPC技術開発に関する報告書」として、「計算科学研究ロードマップ白書」および「HPCI技術ロードマップ白書」がまとめられ、2012年HPCI計画推進委員会に提出された。

2012年、文部科学省「将来のHPCIシステムのあり方の調査研究」（以降，調査研究と呼ぶ）が公募され、以下の4課題が2012年7月から2014年3月の2年間、調査研究が実施された。

- 「レイテンシコアの高度化・高効率化による将来のHPCIシステムに関する調査研究」 (代表：東京大学情報基盤センター　石川裕)
- 「演算加速機構を持つ将来のHPCIシステムに関する調査研究」 (代表：筑波大学計算科学研究センター　佐藤 三久)

図 1.1: 開発スケジュール

- 「高メモリバンド幅アプリケーションに適した将来の HPCI システムのあり方の調査研究」(東北大学サイバーサイエンスセンター 小林 広明)
- 「アプリケーション分野からみた将来の HPCI システムのあり方の調査研究」(計算科学研究機構 (現、計算科学研究センター) 富田 浩文)

上記調査研究が進む中、理化学研究所は 2013 年に以下の三つのシステム設計方針に基づき、計算ノードに汎用 CPU（「レイテンシコアの高度化・高効率化による将来の HPCI システムに関する調査研究」 の成果）と加速部から (「演算加速機構を持つ将来の HPCI システムに関する調査研究」) 構成されるシステムを富岳」として提案した。

1. サイエンスドリブンで設計を進める。すなわち,「今後の HPC 技術開発に関する報告書」にある計算科学ロードマップ（第 2 版）に基づき，社会的・科学的課題解決に必要な計算資源を提供することを目的として設計を進める。
2. サステイナブルなシステムであること。「京」の後継機として「京」の資産を受け継ぎ，そして将来の計算機システム発展動向を見据えたシステムとする。
3. TCO（Total Cost of Ownership）を考えたシステムであること。低消費電力，ソフトウェア高移植性，高耐故障性を有するシステムを設計する。

平成 25 年 11 月総合科学技術会議（現、総合科学技術・イノベーション会議）において消費電力 30〜40 MW で理論演算性能 1 エクサフロップス級を有し以下の目標を掲げた [51]。

1. 「京」の 100 倍のアプリケーション実効性能。
2. 2020 年から運用可能で優れた性能電力比と幅広いアプリケーション実行環境。

本目標を達成するために、アプリケーション開発者とハードウェア開発者とともにアプリケーションおよびハードウェアのコデザイン（協調設計）を行うこと、とした。

プロジェクトが正式に開始された 2014 年早々、理化学研究所は加速部も含めた開発・製造費用に関して精査し、文部科学省「将来の HPCI システムのあり方の調査研究」に諮った。その結果,加速部の開発は技術自体の実現可能性は十分に見込まれるが、システムとして競争力を持つためには広い需要を得る必要があると評価され、本開発プロジェクト内での開発を断念した。代替案として、加速部として GPU を採用することも検討したが、必要とされる性能を満たす GPU の登場時期が不明確であり、ポスト「京」開発を遅延なく遂行するためには、GPU の採用は適切ではないと判断した。

総事業費の範囲内で幅広いアプリケーションを高い実効性能で利用できるシステムとする必要があることから、演算加速部を採用せずに、当該資源を汎用部の拡大に用いるという選択をした。しかし、消費電力 30〜40 MW で理論演算性能 1 エクサフロップス級を有するハードウェアを実現することは困難であった。調査研究では 8 タイプのアーキテクチャを評価していたが、その一つを基にすれば当初の理論性能目標を達成できる可能性があった。しかし、このアーキテクチャは Linpack で 1 エクサフロップス性能が見込めても、プロジェクト目標である幅広いアプリケーション実行環境を提供するという点で受け入れられず、このアーキテクチャを採用しなかった。

社会的・科学的課題解決に必要な計算資源を提供するという観点で支障をきたさず、アプリケーションレベルでの実効性能として最大で「京」の 100 倍を目標とし、汎用部のみで構成されるシステムを文部科学省「次期フラッグシップシステムに係るシステム検討ワーキンググループ」に提

案した。この提案の議論は「次期フラッグシップシステムに係るシステム検討ワーキンググループ報告書」としてまとめられた [52]。

理化学研究所と富士通との研究開発契約は、一般公募を経て 2014 年 10 月から開始された。そして、文部科学省が選定した九つの重点課題実施機関とともに、ハードウェアとソフトウェアのコデザインを進めた。各重点課題実施機関からアプリケーションを選定してもらい、それらのターゲットアプリケーションの実行効率を高めるべく、ハードウェアおよびソフトウェアの設計を進めた。ハードウェア・システムソフトウェア系の開発に 13 の WG（ワーキンググループ）を、アプリケーション系に九つの WG を設置し、設計が進んでいった。開発初期からアプリケーション開発者とともにハードウェアおよびシステムソフトウェアの設計・開発を密に進められたことにより、システム開発者はアプリケーションの特徴を認識し、アプリケーション開発者はハードウェアやシステムソフトウェアの構造および性能限界を認識することができた [50]。また、アプリケーション開発者は早期のソフトウェアチューニングが可能となった。

2015 年 9 月文科省 HPCI 計画推進委員会とシステム検討 WG 合同で基本設計の評価が開始され 2016 年 1 月文部科学省報告書において、「予算等の様々な制約条件がある中で、課題解決型であり国際競争力のある、世界最高水準の汎用性のあるスーパーコンピュータの実現という開発目的に向けた設計がなされており、概ね妥当である。」とされ、また、システムの特色として、消費電力性能、計算能力、ユーザの利便・使い勝手の良さ、画期的な成果の創出を備えた 2020 年代において世界の他のシステムに対して総合力で卓抜するものであること、が示された。その後、2019 年まで、詳細設計を進めた。

本プロジェクトの研究開発計画は，調査研究期間である 2012 年頃に調査した半導体製造技術の将来動向を基に策定したものであった。この当時，ポスト「京」の CPU は 10 nm 微細加工技術を利用することで開発を進めていた。詳細設計が進む中、2016 年には 10 nm 微細加工技術の雲行きが怪しくなり，この技術では我々が必要とする性能は達成できないことが判明した。選択肢は 2 つあり，このまま開発期限を変更せず目標性能を下げて進めるか，次の微細加工技術 7 nm を使った開発に遅延させるか，のいずれかであった。様々な議論があったが，開発期間を 1 年から 2 年延伸するとともに新たなる付加価値を付けることにした。

2016 年 5 月、理化学研究所は付加価値の一つとして、半精度浮動小数点演算の実装可能性を富士通に打診した。富士通は開発工程の更なる遅延を危惧し当初否定されたが、Arm V8 SVE（Scalable Vector Extension）最終仕様に半精度浮動小数点演算命令が定義されたため、富士通も A64FX に本命令を実装した。AI 応用分野では半精度浮動小数点演算が使われており、今 AI 応用分野にも「富岳」が利用されようとしている。

2017 年 11 月、文部科学省富岳 に係るシステム検討 WG のコスト及び性能の評価に係る報告書において、詳細設計に基づくポスト「京」システム及びアプリケーションの全体として，所与のコスト制約の中で，プロジェクトの性能目標である最大で「京」の 100 倍の実効性能及び 30〜40 MW以下の消費電力を達成する見込みであり，世界最高水準の汎用的な計算機システムの実現に向けた開発が着実に進展していることから，コスト・性能評価の結果はおおむね妥当と評価された。

2018 年夏、富士通沼津工場において試作機が完成し基本性能の測定・評価がおこなれた。同年 9 月、文科省 HPCI 計画推進委員会 (第 37 回)、ポスト「京」に係るシステム検討 WG（平成 30 年度第 2 回）合同会議において、8 月末までの初版 CPU 評価に関して、機能試験は問題なく、負荷試験でバグを確認したものの次版 CPU で修正済みであり、製造計画には影響を与えないとの評価結果を得た。

2018 年 製造移行前の評価として中間評価が文部科学省により実施され、同年 9 月 HPCI 計画推進委員会「ポスト「京」システム開発に係る中間評価票」において、「ポスト「京」システム設計

結果に基づきポスト「京」の製造・設置を着実に推進することが適当」と評価を受けた。

2019 年 1 月 最終的に動作周波数としてノーマルモード 2.0GHz、ブーストモード 2.2GHz、全体のノード数を 15,8976 ノードとするシステム構成が文部科学省システム検討 WG で承認され、製造に着手した。2019 年 5 月 23 日には名称「富岳」を公開した。

2019 年 12 月 2 日、かほく市の富士通 IT プロダクツで組み立てられた「富岳」の出荷が開始され、約 6ヶ月をかけて全ての筐体が R-CCS に設置された。

その後、システム調整期間においては、2020 年 1 月から 3 月にかけて理研に設置された実機の一部を使った開発環境が理研側開発者およびゴードン・ベル賞チャレンジユーザに対して提供された。2020 年 4 月からは、スーパーコンピュータ「富岳」成果創出加速プログラムを設置し、一部のユーザに対して共用前評価環境として提供された。また、同じく 4 月から「富岳」が新型コロナウイルス対策を目的とした優先的な試行的利用が開始され、大きな社会的なインパクトのある成果を得た。2016 年 6 月には、「富岳」が 4 つのスパコンランキング（TOP500、HPCG、HPL-AI、Graph500）で世界第 1 位を獲得、2022 年 11 月まで、このランキングを維持している。

2021 年 3 月からは、共用が開始され、プロジェクトを終了した。

第2章　システム概要・性能

2.1　スーパコンピュータ富岳・システム概要

2.1.1　システム諸元

スーパコンピュータ富岳は、Fujitsu A64FX メニーコアプロセッサをノードとし、これを Tofu-D interconnect で結合した大規模分散メモリシステムである。システム諸元を表 2.1 に示す。

表 2.1: 諸元

総理論倍精度 浮動小数点演算性能	537 PF (ブーストモード) 488 PF (通常モード)
計算ノード数	158976 (24 x 23 x 24 x 12)
ネットワークトポロジ	6 次元 mesh/torus (Tofu-D interconnect)
総メモリ容量	4.8 PiB
ストレージ容量（第 1 階層）	15.8 PB
ストレージ容量（第 2 階層）	150 PB
ノードプロセッサ	Fujitsu A64FX
微細加工技術	7 nm FinFET
プロセッサ動作周波数	2.2 GHz(ブーストモード), 2.0 GHz(通常モード)
ノード当たりプロセッサ数	1
ノード理論 DP 演算性能	3.3 TF (ブーストモード) 3.0 TF (通常モード)

2.1.2　メニーコアプロセッサ A64FX

表 2.2 に、富岳のプロセッサである、富士通 A64FX の仕様の概要を示す。

プロセッサチップは、7nm FinFET の半導体技術を使ったメニーコア・プロセッサである。図 2.1 と図 2.2 にプロセッサのブロック図とチップパッケージとダイ写真を示す。12 コアづつグループに分かれており、これを CMG (Core Memory Group) と呼ぶ。同一 CMG にあるコアはコア間で共有される L2 キャッシュと HBM(high bandwidth memory) を接続するメモリコントローラに接続されている。CMG は、チップ内のネットワークで接続されており、CMG 間においても共有メモリは可能である。プログラミングモデルとしては、CMG あたりに OpenMP のプロセスを配置し、MPI で通信する MPI+OpenMP、もしくは 4CMG の全部のコアを OpenMP でプログラムする共有メモリプログラムも可能となっている。

表 2.2: A64FX プロセッサの仕様

Component	Specification
Architecture	Armv8.2-A SVE (512 bits SIMD)
Core	48 cores for compute and 2/4 for OS activities
	Normal Mode: Freq: 2.0 GHz, DP: 3.072 TF, SP: 6.144 TF, HP: 12.288 TF
	Boost Mode: Freq: 2.2 GHz, DP: 3.3792TF, SP: 6.7584 TF, HP: 13.5168 TF
L1 Cache	64 KiB, 4 way, 256 GB/s(load), 128 GB/s (store) @2.0GHz
L2 Cache	8 MiB/CMG, Total 32MiB/16way, BW for Core: 128 GB/s (load), 64 GB/s (store) @ 2.0GHz
Memory	HBM2 32 GiB, BW for Chip 1024 GB/s
Interconnect	TofuD: 28 Gbps x 2 lane x 10 port, injection BW: 6.8GB/s x 6
I/O	PCIe Gen3 x 16 lane
Silicon Technology	7nm FinFET, CoWoS (Chip on Wafer on Substrate) [3] for HBM2

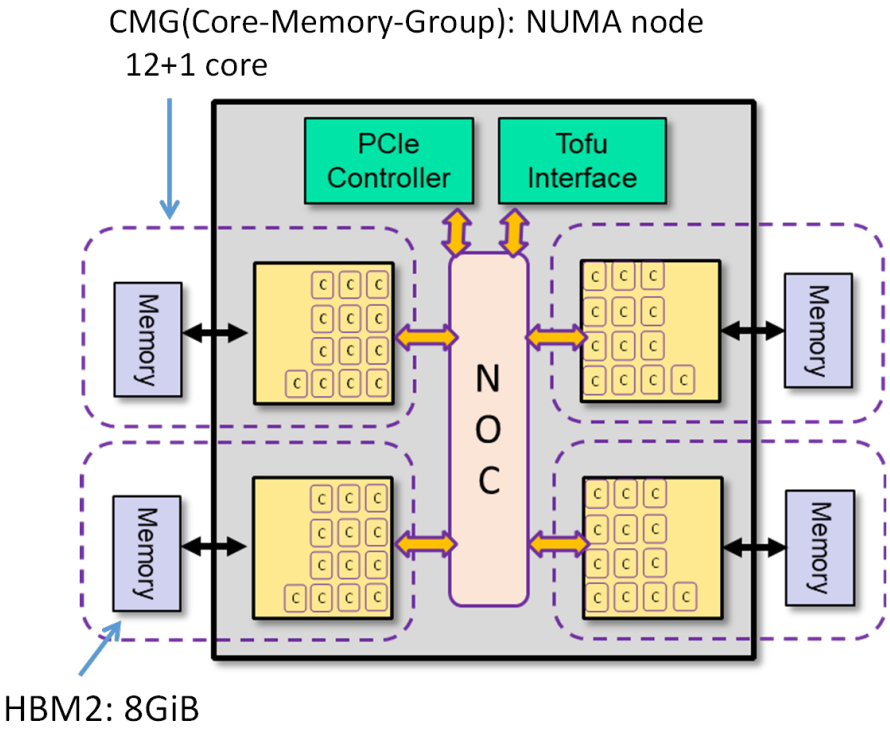


図 2.1: A64FX プロセッサのブロック図

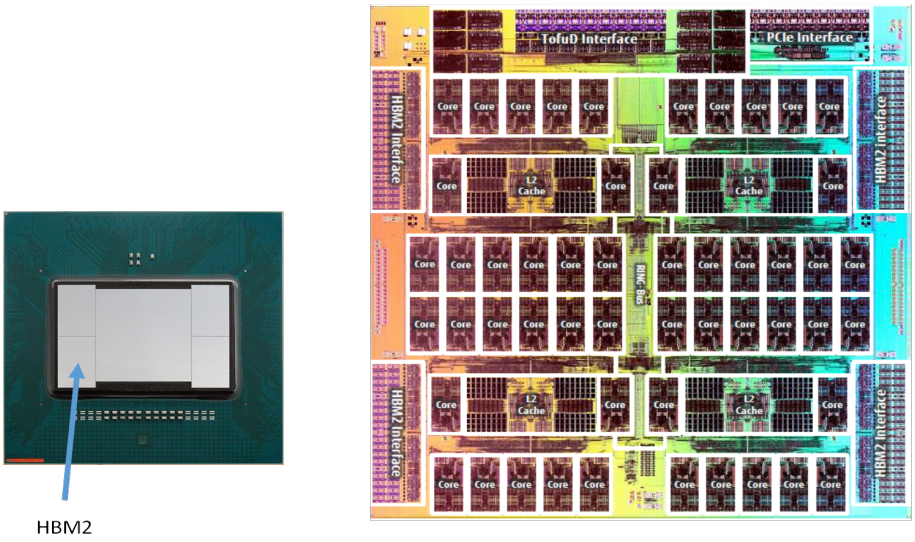


図 2.2: A64FX プロセッサのチップパッケージとダイ写真

CMG には 12 コアに加えて 1 つのアシスタントコアとよばれるコアがあり、これはオペレーティングシステムのデーモンプロセスや IO 処理専用コアとして用いられることを想定している。なお、アシスタントコアは少なくとも 2 つの CMG で利用できることとしている。

標準の周波数は、2.0GHz で、2.2GHz に上げることができる（ブーストモード)。

命令セットは、Arm 社の 64 ビットアーキテクチャの命令セット Armv8.2-A を採用している。さらに、Arm の高性能計算のための SIMD 演算命令の仕様である SVE (Scalable Vector Extension)[3] をサポートする。SVE による SIMD(Single Instruction Multiple Data) 幅は 512 ビットで、倍精度浮動小数点演算の理論性能は、標準周波数の 2.0GHz の場合、コアあたり 64GF である。

SVE は、以下の特徴を持つ。

1. ベクトル長非依存プログラミング (Vector-length agnostic programming) が可能。ハードウエアでサポートされている SIMD 演算命令のベクトル長は、VL という名前のレジスタに格納されており、各命令はこれを参照して動作するように設計されている。したがって、バイナリレベルで異なるベクトル長のハードウエアで動作するプログラミングが可能である。

2. SVE のほとんどの SIMD 演算命令は、ベクトルの各要素の演算を制御するプレディケートマスクを持つことができる。プレディケートマスクを生成する豊富なマスク生成命令があり、これを駆使することによってコードをコンパクトにすることができる。たとえば、SIMD 演算命令を使ったループの最初と最後の処理を簡単にすることができる。

3. アドレスをベクトル要素として持つオペランドとして、複数のメモリアドレスの要素を読み書きする、Gather ロード命令、Scatter ストア命令をサポート。

4. ベクトル要素のデータ型は、FP64(倍精度浮動小数点)、FP32(単精度浮動小数点) のほかに、FP16(半精度浮動小数点) や INT8（8 ビット整数）などのデータタイプをサポートする。

SVE は、FP16（半精度浮動小数点）や INT8 のデータタイプのベクトル演算もサポートしており、AI の中心技術であるディープラーニングの計算を高速化することが期待される。

表 2.3: ストレージの仕様

		Minimum Throughput	Measured Throughput
1st Storage	write	49 MB/s /node	125 MB/s /node
	read	113 MB/s /node	293 MB/s /node
2nd Storage	write	200 GB/s /volume	211 GB/s /volume
	read		220 GB/s /volume

Note: The 2nd storage is formed from 6 volumes.

このチップの大きな特徴は、HBM による非常に高いメモリバンド幅である。HBM は、シリコン基盤上に、積層メモリと CPU チップを Si 貫通電極（TSV: through-silicon via）により接続するメモリ技術であり、HBM の最新規格の HBM2 のメモリを使い、１つの積層メモリあたり 256GB/s のメモリバンド幅を持つ。4 つの積層メモリが接続されており、全体では 32GiB の容量、1TB/s のメモリバンド幅を持つ。計算科学の多くのアプリケーションは、頻繁にメモリにアクセスする必要があるプログラムであり、この HBM による非常に高いメモリバンド幅は多くのアプリケーションに有効である。演算性能とメモリアクセス性能の比である、B/F 値は約 0.33 となっている。

2.1.3 ネットワークとストレージ

大規模な並列アプリケーションにおいて、京との互換性を維持するためにネットワークは、京のネットワークである Tofu ネットワークを改良した Tofu Interconnect D と呼ばれるネットワークを用いる。図 2.1 に示すように、ネットワーク・インタフェースや IO のための PCIe は同じチップに統合されている。8 byte の通信によるレーテンシは、0.49〜0.54 usec, 1MiB 通信時のスループットは、6.35 GB/s である。6 つの TNI（Tofu Network Interface。個々の性能は 6.8 GB/s、6 つあわせて 40.8 GB/s）が Tofu ネットワーク・ルーターに接続されており、ルーターから 1 ポート 2 レーン× 10 ポート（計 20 レーン）に分割される。

第一階層ストレージは、16 の計算ノードごとにある、SIO ノードに接続された SSD により構成されている。この階層は、ノード内テンポラリ領域、共有テンポラリ領域、第 2 階層ストレージのキャッシュ領域に使われている。第二階層ストレージは、グローバルファイルシステムであり、富士通によって開発された FEFS であり、Lustre ベースのファイルシステムとなっている。第二階層ストレージは、InfiniBand で結合されたハードディスクからなる。仕様については、表 2.3 に示す。

それぞれのノードのオペレーティングシステムは、Linux である。システムのデーモンプロセスは、アシスタントコアと呼ばれる、計算コアとは別の 2 つまたは 4 つのコアで実行される。IO ノードでは IO のために負荷が高くなるために、4 つのアシスタントコアを持つプロセッサとなっている。

2.2 基本性能

ここでは、各要素、コンパイラ、言語システム、ライブラリについての基本性能を示す。なお、設計の検収条件とした性能と検収時に確認された性能を示す。

表 2.4: 基本性能

			必須目標	検収時測定性能
全系	Linpack	効率	80.833%	82.86 %
		ノーマル時性能	394 PF	404.69 PF
		ノーマル時電力性能	13.5 GF/W	15.418 GF/W
		ノーマル時電力	29.3 MW	26.248 MW
CPU (48 core)	DGEMM	効率	87.4%	88.13%
	Stream Triad	バンド幅	654 GB/s	831.43 GB/s
Tofu	pingpong	片道遅延	796 nsec	495 nsec
	1hop	遅延	187 nsec	71 nsec
パワーノブ	周波数変更 (1.6GHz/2.0GHz)	遷移時間	130 usec	32.48 usec
		電力効果	87%	82%
	命令実行数制限 (2命令/4命令)	遷移時間	8 usec	0.06 usec
		電力効果	97%	92%
	FLA Only (エコモード off)	遷移時間	8 usec	0.06 usec
		電力効果	90%	82 %
	FLA Only (エコモード on)	遷移時間	5 msec	2.69 msec
		電力効果	76 %	69 %
	HBMアクセス制限	遷移時間	8 usec	4.52 usec
		電力効果	100 %	100%
SIMD幅変更	ベクトル長レジスタ変更 (256bit/512bit)	遷移時間	8 usec	0.06 usec
		電力効果	91%	78 %
電力モード	node retention	遷移時間	5 msec	2.69 msec
		電力	68 W	47.7 W
	ブーストモード遷移時間	遷移時間	1.64 msec	1.30 msec

表 2.5: コンパイラ性能

		必須目標	検収時測定性能
基礎カーネルベンチマーク (single core)	y(i) = Const; for i = 1; n	9.33 GF	14.70 GF
	y(i) = x(i); for i = 1; n	6.67 GF	9.74 GF
	y(i) = x1(i) + x2(i); for i = 1; n	5.18 GF	7.32 GF
	s = s + x1(i) * x2(i); for i = 1; n	12.97 GF	22.50 GF
	y(i) = x1(i) + y(i - 1); for i = 1; n	0.16 GF	0.22 GF
	y(i) = x1(i) * x2(i); for i = 1; 4 * n; 4	0.59 GF	0.86 GF
	y(ind(i)) = x1(i); for i = 1; n	1.13 GF	1.46 GF
	y(i) = x1(ind(i)); for i = 1; n	1.73 GF	3.36 GF
	cy(i) = cx1(i) + cx2(i); for i = 1; n	5.18 GF	7.21 GF
	cy(i) = cx1(i) * cx2(i); for i = 1; n	8.64 GF	21.1 GF
	if (int(i))y(i) = y(i) + c0 * x1(i); for i = 1; n	2.36 GF	4.78 GF

表 2.6: OpenMP 性能

	必須目標	検収時測定性能
DO overhead / 12 thread	0.14 usec	0.12 usec
DO overhead / 48 thread	2.05 usec	1.11 usec
PARALLEL DO overhead / 12 thread	0.65 usec	0.40 usec
PARALLEL DO overhead / 48 thread	5.06 usec	3.54 usec
BARRIER overhead / 12 thread	0.12 usec	0.11 usec
BARRIER overhead / 48 thread	2.05 usec	1.14 usec
REDUCTION overhead / 12 thread	1.29 usec	0.95 usec
REDUCTION overhead / 48 thread	8.43 usec	4.85 usec
TASK overhead / 12 thread	6.39 usec	2.98 usec
TASK overhead / 48 thread	22.29 usec	8.93 usec
TASKWAIT overhead / 12 thread	5.62 usec	3.16 usec
TASKWAIT overhead / 48 thread	12.92 usec	8.96 usec

表 2.7: 数学ライブラリ性能

	必須目標	検収時測定性能	備考
DGEMV(Level 2 BLAS)	3.69 %	6.52 %	CPU 動作周波数2.0GHz。1CMG/1 ノード実行時の演算効率
1次元FFT（倍精度複素数）	8.8 GF	12.70 GF	CPU 動作周波数2.0GHz。1コア演算性能

表 2.8: MPI 通信ライブラリ性能

		必須目標	検収時測定性能
Ping Pong 片道遅延	8 Byte Z 軸隣接ノード間	1.25 usec	1.05 usec
	32 Byte Z 軸2 ホップノード間	2.84 usec	2.48 usec
	8 Byte ノード内(CMG 間)	0.93 usec	0.86 usec
Ping Pong スループット	Z 軸隣接ノード間	4.61 GB/sec	7.60 GB/sec
	ノード内(CMG 間)	5.26 GB/sec	9.25 GB/sec
Put/Win_flush ソフトウェア部分遅延	8 Byte Z 軸隣接ノード間	0.89 usec	0.44 usec
	8 Byte ノード内(CMG 間)	1.15 usec	0.37 usec
Put/Win_flush スループット	Z 軸隣接ノード間	4.61 GB/sec	6.34 GB/sec
	ノード内(CMG 間)	4.61 GB/sec	6.45 GB/sec
Get/Win_flush ソフトウェア部分遅延	8 Byte Z 軸隣接ノード間	1.71 usec	1.41 usec
	8 Byte ノード内(CMG 間)	1.76 usec	0.72 usec
Get/Win_flush スループット	Z 軸隣接ノード間	4.61 GB/sec	6.34 GB/sec
	ノード内(CMG 間)	4.61 GB/sec	5.94 GB/sec
Allreduce	全系ノードw/ ppn=4 8 Byte	42.5 usec	12.45 usec
Bcast	全系ノードw/ ppn=4 4 MiB	4796 usec	1184.33 usec
Allreduce	全系ノードw/ ppn=4 8 KiB	1236 usec	768.20 usec
Allgather	全系ノードw/ ppn=4 256 Byte	94.4 msec	25.58 msec
Alltoall	1ノードw/ ppn=4 256 KiB	172 usec	59.43 usec
ライブラリ消費メモリ容量	1,536ノードw/ ppn=4 省メモリ型通信モード時	0.11 GiB/proc	0.093 GiB/proc
	全系ノードw/ ppn=4 省メモリ型通信モード時	1.8 GiB/proc	0.50 GiB/proc

表 2.9: OS およびファイル IO およびシステム管理性能

		必須目標	検収時測定性能
OSジッタ	Linux Only 平均ノイズ率	$5x10^{-5}$以下	$2.47x10^{-6}$
	Linux Only 最大ノイズ長	42.5usec以下	22.58 usec
OS使用メモリ	Linux Only	2.4GiB以下	2.23 GiB
ファイルI/Oスループット: total		write 7.8TB/s read 18TB/s	write 18.63 TB/s read 45.18 TB/s
ファイルI/Oスループット:per compute node		write 49MB/s read 113MB/s	write 117.25 MB/s read 284.26 MB/s
ファイルI/O メタデータアクセス性能	第2階層ストレージのキャッシュ (per volume)	ファイル作成：32 Kops ファイル削除：48 Kops	ファイル作成：58.02 Kops ファイル削除：50.02 Kops
	共有テンポラリ領域 (per volume)	ファイル作成：32 Kops ファイル削除：48 Kops	ファイル作成：59.15 Kops ファイル削除：50.52 Kops
	ノード内テンポラリ領域 (per compute node)	ファイル作成：0.8 Kops ファイル削除：3.6 Kops	ファイル作成：10.60 Kops ファイル削除：11.01 Kops
第1 階層 第2 階層データ転送スループット (per volume)		125GB/s	178.84 GB/s
第2階層ストレージ (per volume)	ファイルI/O スループット性能	200GB/s	write: 211.49 GB/s Read:219.98 GB/s
	ファイルI/O メタデータアクセス性能	ファイル作成：32Kops ファイル削除：48Kops	ファイル作成：51.74 Kops ファイル削除：76.61 Kops
ノード単体OS 起動時間		2分以内	1分49秒
ノード単体OS 再起動時		3分以内	1分48秒
システム起動時間		2時間以内	1時間57分33秒
本体システムの全BoBへのパッチ適用時間[注1]		1時間30分以内	48分40秒
計算資源提供率		80%	87 %

(A) SPEC CPU 2017 (int speed/base)

	Lang	Threads	A64FX	Xeon
600.perlbench_s	C	1	1.20	6.20
602.gcc_s	C	1	2.63	9.57
605.mcf_s	C	1	3.42	11.2
620.omnetpp_s	C++	1	1.26	7.31
623.xalancbmk_s	C++	1	1.61	9.46
625.x264_s	C	1	2.06	11.6
631.deepsjeng_s	C++	1	1.37	5.17
641.leela_s	C++	1	1.26	4.36
648.exchange2_s	F90	1	1.42	13.2
657.xz_s	C/OpenMP	48	8.52	23.5
SPECspeed®2017_int_base			1.98	9.07

(B) SPEC OMP 2012 (speed/base)

	Lang	Threads	A64FX	Xeon
350.md	F	48	2.63	62.6
351.bwaves	F	48	15.5	11.2
352.nab	C	48	3.00	12.9
357.bt331	F	48	5.82	16.0
358.botsalgn	C	48	5.22	10.5
359.botsspar	C	48	3.07	6.83
360.ilbdc	F	48	7.69	8.25
362.fma3d	F	48	4.28	11.3
363.swim	F	48	53.1	8.38
367.imagick	C	48	12.2	13.6
370.mgrid331	F	48	32.6	7.46
371.applu331	F	48	8.88	14.4
372.smithwa	C	48	12.8	11.8
376.kdtree	C++	48	3.22	9.24
base(g-mean)			7.77	12.0

図 2.3: SPEC ベンチマークの結果

2.3 ベンチマークによる評価

2.3.1 SPEC ベンチマーク

SPEC ベンチマークは、一般的な CPU 性能を評価するためのベンチマークセットである。図 2.3 に、SPEC CPU(int) と SPEC OMP の結果を示す。評価のために、Speed (一つのタスクでの評価)/Base(すべてのベンチマークプログラムに共通のコンパイルオプションを適用) を示した。A64FX は、2.0GHz(ノーマルモード)、Xeon は、SPEC CPU(int) については、Platinum 8168(Skylake), 2.7GHz, 24cores x 2 chip, turbo on, SPEC OMP については、Platinum 8280(Cascade Lake), 2.7GHz, 28cores x 1chip, hyperthread on (56threads), turbo on. を用いた。

SPEC CPU(int) の評価は、657.xz_s を除いて 1 スレッド、すなわち 1 コアでの評価である。Xeon に比べて、1/4 程度の性能となっている。SPEC OMP は、48 スレッドの比較で、全体としては、Xeon の 65%であるが、363.swim と 370.mgrid に見られるように、メモリバンド幅が必要なプログラムに関しては、HBM の高メモリバンド幅による大幅な性能向上が見られる。しかし、350.md は、A64FX は極端に性能が悪いが、これは手続きのインライン化やループのアンローリングなどの手動の最適化を行うことによる、性能が改善されることがわかっている。この他の SPEC ベンチマークの結果についての詳細は [33] を参照。

2.3.2 オープンソース HPC アプリケーションの性能

すでに、いくつかのオープンソースのアプリケーションが移植・評価されている。図 2.4 は、表 2.10 に示したアプリケーションについて、A64FX(2.2GHz ブーストモード) と Xeon (Xeon Platinum 8268 (CascadeLake, 2.90 GHz, 24 cores/socket、2 ソケット) と比較したものである。実行時間と平均電力を、Xeon に対する相対値で示す。これらのアプリでは、Xeon と同等な性能を半分の電力で達成していることがわかる。

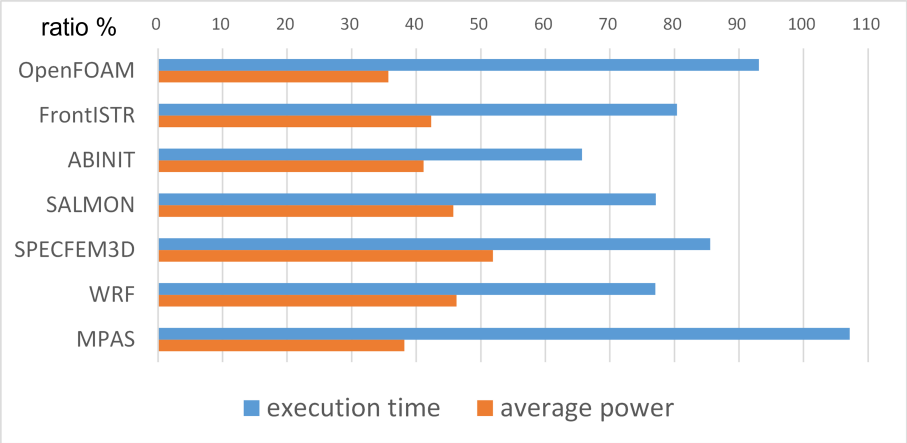


図 2.4: オープンソース HPC アプリケーションの性能 (Xeon との相対性能)

2.3.3 HPL(TOP500), HPCG, HPL-AI, Graph500

表 2.11 に、富岳全体を使った HPL(TOP500), HPCG, HPL-AI, Graph500 の各ベンチマークの値と効率を示す。この値は、2020 年 11 月に公表した値である。参考までに、システムの一部が動作し始めた 2020 年の 6 月の値も示す。

2020 年の 6 月から、2021 年 11 月の 4 期連続で、これらのベンチマークで 1 位を達成した。

表 2.10: 評価したオープンソース HPC アプリケーションの概要とパラメータ

Applications	Field	Ver.	Size	Model	Solver	MPI/ OpenMP	Measured Section
OpenFOAM [23]	CFD	1812	14 million meshes	Motor Bike	PCG	FlatMPI (48 procs)	Time integration loop
FrontISTR [5]	Structure Analysis	5	0.4 million meshes	Hinge	SSOR+CG	FlatMPI (48 procs)	Analysis section
ABINIT [48]	Material	8.10.2	26244 FFT meshes	tmbt_3	GW/spectral method	FlatMPI (48 procs)	Analysis section
SALMON [11]	Ab-initio Light-Matter	1.2.1	1.0 million meshes	exercise_07_classic EM_lr	FDTD	FlatMPI (48 procs)	Analysis section
SPECFEM 3D [12]	Seismic	7.0.2	64 x 64	s362ani	SEM	Hybrid (4 procs x 12 threads)	Iteration loop
WRF [14]	weather	3.8.1	425 x 300 x 35	Conus 12km		Hybrid (4 procs x 12 threads)	Domain integration loop
MPAS [13]	weather	6.2	40962 meshes	The Jablonowski and Williamson baroclinic wave		Hybrid (24 procs x 2 threads)	Time integration loop

表 2.11: 富岳全体を用いたベンチマークの性能

Benchmark	Measured	Peek Perf	Efficiency	(June 2020)
HPL	442.01 PF	537.21 PF	82.3%	(415.53 PF)
HPCG	16.00 PF	537.21 PF	3.0%	(13.4PF)
HPL-AI	2.00 EF	2.14 EF	93.2%	(1.42EF)
Graph500	102.95 Tteps			(70.98)

2.4 ターゲットアプリケーション性能評価結果

富岳において達成された重点課題ターゲットアプリケーションの対京性能倍率と消費電力の値を表 2.12 に示す。

測定値は全て周波数ブーストモードかつエコ無効モードで実行されたものである。詳細設計期間まではシステムが許容する最大消費電力の範囲内で計算を行うために、アプリ毎に周波数 2 モード（通常／ブースト）およびエコ 2 モード（エコ有効／無効）の組み合わせを変えて実行する必要があると評価していたが、完成した富岳においては、重点課題ターゲットアプリケーションの実行に必要な電力が設計期間における推定値よりも全般的に低い結果となり、全ての重点課題ターゲットアプリケーションを最も性能が高い周波数ブーストモードかつエコ無効モードで実行しても電力の制限範囲内に余裕をもって収まることが確認された。

エコモードとはスタンバイ時の電力を削減するモードと浮動小数点パイプライン 2 本のうち 1 本のみを使用するモードとを組み合わせて実現する。周波数通常モードとは計算ノード CPU の動作周波数が 2.0GHz である状態を、周波数ブーストモードとは計算ノード CPU の動作周波数が 2.2GHz である状態を示す。ただし I/O ノードを兼用する計算ノードにおいては CPU の動作周波数は常に 2.2GHz であることに留意する必要がある。

表 2.13 に電力制御各モードを適用した場合の対京性能倍率と平均消費電力・最大消費電力の値を一覧として示す。

表 2.12: ターゲットアプリケーションの対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
GENESIS	131 倍	22 MW
Genomon	23 倍	20 MW
GAMERA	63 倍	21 MW
NICAM+LETKF	127 倍	22 MW
NTChem	70 倍	26 MW
ADVENTURE	63 倍	28 MW
RSDFT	38 倍	30 MW
FFB	51 倍	29 MW
LQCD	38 倍	20 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

アプリケーション毎に設定された計算問題（ターゲット問題）の内容は 4.1 節において表 4.3 で示される。

表 2.13: ターゲットアプリケーションの電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
GENESIS	131 倍		119 倍		126 倍		115 倍	
	22 MW	22 MW	20 MW	20 MW	16 MW	16 MW	14 MW	14 MW
Genomon	23 倍		23 倍		23 倍		22 倍	
	18 MW	20 MW	16 MW	17 MW	13 MW	15 MW	12 MW	13 MW
GAMERA	63 倍		58 倍		56 倍		52 倍	
	22 MW	23 MW	20 MW	21 MW	17 MW	17 MW	16 MW	16 MW
NICAM+LETKF	127 倍		116 倍		121 倍		115 倍	
	24 MW	25 MW	22 MW	23 MW	19 MW	20 MW	18 MW	20 MW
NTChem	70 倍		67 倍		49 倍		45 倍	
	24 MW	26 MW	21 MW	23 MW	18 MW	19 MW	16 MW	17 MW
ADVENTURE	63 倍		60 倍		60 倍		57 倍	
	22 MW	28 MW	24 MW	25 MW	17 MW	22 MW	20 MW	22 MW
RSDFT	38 倍		35 倍		28 倍		25 倍	
	25 MW	28 MW	22 MW	25 MW	20 MW	20 MW	17 MW	18 MW
FFB	51 倍		51 倍		52 倍		50 倍	
	28 MW	29 MW	26 MW	26 MW	23 MW	23 MW	21 MW	21 MW
LQCD	38 倍		36 倍		36 倍		33 倍	
	21 MW	26 MW	19 MW	26 MW	16 MW	26 MW	15 MW	26 MW

第3章　システムのコデザイン

3.1　CPU アーキテクチャ

3.1.1　アプローチ

3.1.1.1　CPU アーキテクチャのコデザインに用いたツール

コデザインのために以下のツールを用いた。

- **パフォーマンス推定ツール:**　このツールは、富士通のスーパーコンピュータ FX100 のプロファイルデータを入力として、与えられたアーキテクチャパラメータによるシステムの性能推定を行うツールである。この性能推定は、富士通のマイクロアーキテクチャによりモデル化されたアーキテクチャモデルを用いて行う。このツールはまた、このアーキテクチャモデルに基づいて消費電力も推定できる。

- **富士通社内プロセッサシミュレータ:**　開発の当初においては、FX100(SPARC 命令セット) のトレースベースのサイクルレベル・シミュレータを拡張したシミュレータとコンパイラを用いて性能評価を行った。のちに、このシミュレータをベースに SVE を含む Arm 命令をサポートし、Arm 用コンパイラを用いて評価を行った。このシミュレータはほぼ 100kHz 相当の周波数で動作する。

- **ハードウエアエミュレータ:**　開発するプロセッサのためのハードウエアエミュレータを開発し、これを主に論理検証と実行時間と消費電力の正確な評価に用いた。このエミュレータはほぼ 1MHz 周波数相当で動作する。このエミュレータはコデザインの過程の最後工程で回路設計がほぼ固まった後に実装された。

- **富岳用の Gem5 シミュレータ:**　汎用プロセッサシミュレータである Gem5 を基に 富岳プロセッサ向けのシミュレータも理研により開発された。このシミュレータはコデザインの中で主にアーキテクチャの検証およびソフトウエア・チューニング検討のために用いられた。

次節に述べるように、基本的なアーキテクチャは本プロジェクトが開始される前に行われたフィジビリティ研究においてメニーコアプロセッサをベースとするシステムにすることが決められた。メニーコアプロセッサを設計するためには、多くのアーキテクチャパラメータや要素について決める必要があり、広い選択肢の設計空間での探索を行った。

基本的な問題の一つは、富岳で実行が想定されている科学技術アプリケーションの規模である。我々が選択したターゲットアプリケーションでさえも数千行のコードであり、複雑なアルゴリズムのデータを用いて記述されている。プロセッサシミュレータは、サイクルレベルで非常に正確な実行時間を得ることができる反面、実行は非常に遅く時間がかかり、ノード間の MPI 通信なしの単一プロセッサでの実行に限定される。

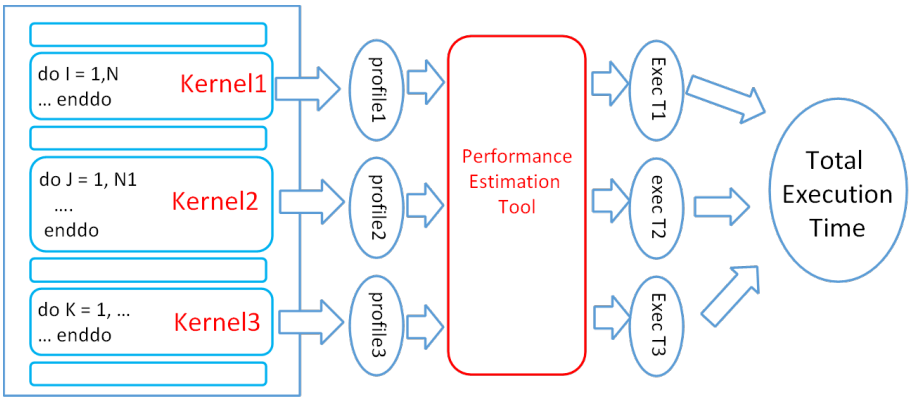


図 3.1: 性能推定ツールの使い方

一方、我々の用いた性能推定ツールは、FX100 の実際のハードウエアでの実行から得られた実行プロファイルを用いた性能解析が可能であるために、大規模な並列プログラムの性能推定に非常に有用であった。FX100 のハードウエアには、メモリアクセスの read/write のビジーサイクルや、L1/L2 キャッシュのビジーサイクル、浮動小数点の演算ユニットのビジーサイクル、命令コミットのサイクルなど、多くの性能カウンターがある。機能ブロックのビジーサイクルを変化させることにより、新しいハードウエアパラメータに対する性能推定が可能になる。実行時間の内訳を、プロセッサのマイクロアーキテクチャのパイプライン構成を考慮して、ぞれぞれの機能ブロックのビジーサイクルを集計することにより計算できる。例えば、L2 キャッシュアクセスがボトルネックだとする。L2 のキャッシュバンド幅が 2 倍になった時に、L2 キャッシュアクセスのビジーサイクルが半分になり、パイプラインの中で、L2 キャッシュアクセスの時間が他のサイクルで隠蔽されない場合に実行時間が削減されることになる。実行時間はスループットによる簡単な数式で求めるようになっているために、カーネルループのような一様な動作をするリージョンに適用するものである。そうでない場合には、推測の精度は低下することになる。さらに、O3 (out-of-order) 資源の性能に対するインパクトを考慮することも難しい。

3.1.1.2 カーネルの抽出

性能解析の最初のステップではそれぞれのターゲットアプリケーションのカーネルループを見つけ、実行プロファイルの取得のためのライブラリ呼び出しのコードを挿入する。図 3.1 に示した通り、アプリケーションの全体の実行時間は、各カーネルの実行時間の総和で求める。デザイン空間の探索においては、アーキテクチャのパラメータを変化させつつ、このプロセスを繰り返すことになる。

重要なカーネルについては、独立したプログラムとして抽出しておく。これらのカーネルは、さらに詳細な解析のためにサイクルレベルのシミュレータで実行できる。これにより、新しい命令セットや O3 の資源量を変化させたときの影響の解析を行うことができる。また、これらのカーネルは論理検証のためのプロセッサエミュレータでも用いられた。本稿で示す、いくつかのカーネルについて以下に示す。

- Kernel (Adventure): Adventure の前処理のためのブロック行列行列演算から抽出されたカーネル。

- Kernel (FFB): FFB の 3 次元疎行列ベクトル演算から抽出されたカーネル。
- Kernel A (NICAM): NICAM の 2 次元水平発散演算子から抽出されたカーネル。
- Kernel B (NICAM): NICAM の 2 次元水平発散演算子から抽出されたチューニングされたカーネル。
- Kernel C (NICAM): NICAM の 2 次元水平拡散演算子から抽出されたカーネル。
- Kernel D (NICAM): NICAM の 3 次元発散減衰演算子から取得したカーネル。
- Kernel (Seism3D): Seism3D の sdiffy3 から取得したカーネル。

また、HPL や Dgemm、Stream ベンチマークなどの簡単なカーネルに関しては解析的なモデルも使用した。このような解析モデルはハードウエアのパラメタを用いた数式で計算できるために、場合によっては有用な方法となった。

インターコネクトに関するコデザインは、通信パタンを抽出し、LogP モデルなどの解析的なモデルにより通信性能を推定した。

3.1.2 コデザイン成果

本章では、コデザインにおける 富岳の様々なアーキテクチャ・パラメータをどのように決定してきたかについて述べる。アーキテクチャのコデザインの大部分は、2014 年から 2015 年の基本設計中に行われた。製造・設置が 2019 年に予定されていたため、コデザインは、シリコンの微細加工技術、メモリ技術、SerDes、IO インターフェス規格など、2019 年頃の製造時に利用可能になる技術を前提に行う必要があった。

フィジビリティ研究プロジェクトの結果を基に、当初の案 (2014 年) の計画は、汎用 CPU と演算加速機構を組み合わせてエクサフロップスの性能を目指すものであった。しかし、予算や適用範囲の問題により、演算加速機構の部分は取りやめとなり、結果として、基本アーキテクチャは、東京大学で行われたフィジビリティ研究で検討された汎用メニーコアプロセッサを基本とした大規模システムに決定した。フィジビリティ研究で検討されたプロセッサアーキテクチャは、64 コアに 512 ビット幅の SIMD 演算器をそれぞれのコアに備えたメニーコアプロセッサであった。報告書では、さらに幅の広い演算器や行列積専用演算器などの案についても検討されたが、適用範囲が限定されていることや実際のアプリでの効率が上がらないなどの理由で採用されなかったことが報告されている。

3.1.2.1 命令セットアーキテクチャと SIMD 命令

命令セットアーキテクチャの選択はアーキテクチャ設計において重要な決定事項である。「京」で用いられていた SPARC 命令セットの代わりに、富士通から Armv8 命令と SVE と呼ばれる Arm SIMD 命令セットをポスト京の命令セットとして提案があり、それを了承した。富士通は SVE の設計についてリード・パートナーとして、これまで Arm に協力してきており、プロセッサアーキテクチャの設計にこの成果を盛り込むことになった。プロセッサは、富士通のこれまでのマイクロアーキテクチャをバックエンドに用いた、カスタムの設計である。

Arm 命令セットアーキテクチャはこれまでモバイルのプロセッサはもちろん、最近では HPC に対しても多くの開発者並びにユーザーに受け入れられている。例えば、Cavium Thunder X2 は、

サーバーと HPC 向けに設計されたプロセッサで米国サンディア国立研究所の Astra や英国ブリストル大学の Isambard など、いくつかのスーパーコンピュータシステムに用いられている。HPC ではインテルの x86 アーキテクチャが支配的なプロセッサではあるが、Arm プロセッサは、HPC コミュニティにおいては可能性と多様性に向けてオープンなアーキテクチャとして期待されている。

SVE は、拡張 SIMD の命令セットである。SVE の最も重要な特徴は、128 ビットから 2048 ビットの長さのベクトル長を一つの命令セットでサポートしているところにある。この機能によって、Vector Length Agnostic (VLA) プログラミング、すなわち、ベクトル長に依存しないプログラミングを実現している。富岳おいては、フィジビリティ研究での結果から、2 つの 512bit 幅の SIMD の演算器をサポートすることにした。開発期間において、検証の一つとして gem5 シミュレータと McPAT と呼ばれる消費電力の推定ツールを用いて、評価を行った。その結果、この SIMD の構成は、後で述べる現在の O3 資源においては、性能と消費エネルギーのバランスが良い構成であることが示されている。なお、この結果として設計された A64FX プロセッサは SVE をサポートした最初のプロセッサである。

3.1.2.2 プロセッサチップの構成

メニーコアプロセッサアーキテクチャの基本構成については、富士通から提案された。それぞれのコアは L1 キャッシュをもち、いくつかのコアからなるクラスタのコアが L2 とメモリコントローラを共有する。このコアのクラスタを Core Memory Group (CMG) と呼び、プロセッサチップの上では、いくつかの CMG はチップ上のネットワーク (NOC: network-on-chip) で接続される。

Intel や AMD などの高性能プロセッサでは、L1 と L2 キャッシュがコアにあり、L3 がラストレベルキャッシュとしてコアに共有されるという構成であるが、本プロセッサではコアのダイサイズを削減するために、コアにおいては L1 のみの構成になっている。A64FX のチップの場合は、L1 キャッシュのダイサイズの占有面積はコア内で 10%以上となっている。もしも、さらにキャッシュの階層を増やすとなるとそれぞれのコアで同じ面積のダイサイズを必要とすることになり、チップ写真からは全体のダイサイズは 4%以上増えることになることが推測される。このインパクトは大きく、コストの観点から受け入れることができない。

基本構成を基に、以下のパラメタを決定することにした。

- CMG のコア数
- チップ内の CMG の数
- CMG 内の共有 L2 キャッシュとコアの接続方式
- L1 と L2 キャッシュのウェイ数、サイズ、スループット
- CMG 間の NOC のトポロジー
- チップのダイサイズ
- ノード内のチップ数

シリコン微細加工技術について、利用可能であるとしたテクノロジは 7nm FinFET である。チップのダイサイズがコストという点では最も支配的なファクタとなる。チップのコストはほぼサイズに比例するが、あるサイズをこえるとコストが急激に増加することが知られている。さらに、チップの欠陥率はチップの大きさが大きくなるにつれて大きくなる。一つの構成として、小さいチッ

表 3.1: ダイとパッケージ、マルチチップモジュールのコスト

#Cores x #PKG(p)/MCM(M)	#Dies	Die area	Die cost	PKG/MCM cost	Total
64 x 1p	1	1.00	0.82	0.18	1.00
32 x 2p	2	0.65	0.80	0.26	1.06
16 x 4p	4	0.38	0.73	0.30	1.03
32 x 2M	2	0.64	0.77	0.31	1.08
16 x 4M	4	0.37	0.71	0.34	1.05

表 3.2: 64 コアの場合、4 CMG と 8 CMG の構成のカーネルの性能

kernel	4CMG 1proc/CMG 16threads/proc	4CMG 2proc/CMG 8threads/proc	8CMG 1proc/CMG 8threads/proc
Kernel (Adventure)	100%	112%	112%
Kernel (FFB)	100%	105%	110%
KernelA (NICAM)	100%	100%	100%
KernelC (NICAM)	100%	90%	90%
kernelD (NICAM)	100%	113%	122%

NOTE: The performance in this table is the average of small kernels extracted in the different ways from the kernel shown Table 3.4, 3.3

プをマルチチップモジュール (MCM) を用いて結合することも検討した。最近では、AMD はチップレットテクノロジを用いて成功している。このアプローチの利点は、小さなチップは比較的、欠陥率が小さくコストを下げることができる点にある。しかし、今回のケースにおいては、MCM のコストは高いことが明らかになっており、インターコネクトや I O といったチップを別に製造しなくてはならなく、コスト高になるという結論に至った。ダイ、パッケージ、MCM を用いて、1 つのパッケージに 64 コアを実装する場合について、基本設計時に見積もった相対コストを表 3.1 に示す。また、MCM の場合はチップ間の接続も消費電力を増加させる要因になる。

以上の検討から、単一のダイに、いくつかの CMG とインターコネクトのためのネットワークインタフェース、IO のための PCIe を NOC でチップ上で接続する構成を採用することにした。製造時のデータによれば、ダイのサイズはほぼ $400mm^2$ であり、7nm FinFET のプロセスを使うチップとしては適当なサイズに収まった。

京コンピュータとある程度の互換性を維持するために、プログラミングモデルとしては OpenMP-MPI のハイブリッドプログラミングとし、それぞれの MPI プロセスを一つの CMG を使って OpenMP プログラムを実行することを想定した。同じ CMG 内のコアで L2 を共有するために、L2 とコアの間はクロスバーネットワークで接続することにした。

基本設計の初期においては、コア数は 64 とし、8 コア/CMG x 8 CMGs と 16 コア/CMG x 4 CMGs の構成について、ターゲットアプリケーションから抽出したカーネルを用いて、評価した。その結果を、表 3.2 に示す。性能評価について、それぞれのケースについて、L2 キャッシュの合計のサイズを一定にして評価を行った。性能推定ツールで評価した結果からは、以下のことが分かった。

- いくつかのカーネルでは、8 コア/CMG の場合、L2 のキャッシュが小さくてすむために相対的にレーテンシが小さくなり、16 コア/CMG の構成よりも、性能が高くなった。
- いくつかのカーネルでは、8 コア/CMG の場合、L2 のキャッシュが小さくキャッシュヒット率が小さくなり、性能が低下してしまうケースがある。

CMG のコア数が 8 以下の場合、L2 のキャッシュサイズが小さすぎることになる。CMG 内のコア数については HPL に対しては解析的なモデルで検討した。

8 コア/CMG の方が 16 コア/CMG よりも良いという結論であった。のちに、TSMC の 7nm FinFET テクノロジを用いることになり、最終的に 12 コア/CMG で 4CMG の合計 48 コアの構成を選択した。次節で述べるように、4CMG に対しては 4HBM ユニットをメインメモリとして用いることにした。

CMG 間のメモリ一貫性プロトコルをサポートするためにリングトポロジーネットワークに似たオンチップネットワークを設計した。これを用いることにより、複数の CMG を用いて共有メモリプログラムを実行できるようになる。

3.1.2.3 メモリ技術の選択

CPU チップの浮動小数点演算のピーク性能は数 TFLOPS に達すると想定されていたため、DDR4 のメモリバンド幅はこの演算ピーク性能に対しては不十分であった。したがって、メモリバンド幅と浮動小数点演算性能をバランスさせるために、HBM や HMC などの高性能メモリ技術を検討した。

HMC は、スタックメモリチップと CPU チップを高速シリアルリンクで接続する技術である。富士通はすでに FX100 に HMC1.0 を採用した実績があった。 HMC2.0 によるリンク速度はリンクあたり（方向あたり）60 GB/s で、最大 4 つのリンクを使うことができる。シリアルリンクを駆動するための消費電力は比較的大きいことがわかっている。

HBM は、スタックメモリチップと CPU チップをシリコンインターポーザ上の TSV を介して接続する技術である。HBM2 のバンド幅はモジュールあたり 256 GB/s と非常に高い。 HBM2 の容量は最大 8GiB であるが、シリコンインターポーザーが必要なためコストが高くなるという問題がある。

2019 年頃に利用可能なメモリ技術として、読み取りと書き込みの両方のメモリバンド幅が高く、電力効率が高いことを重視し、HBM2 を採用することとした。追加の DDR メモリについては、コストを下げるためにつけないこととした。前節で説明したように、CMG に接続されている HBM2 モジュールの数は 4 つで、メインメモリの容量は 32GiB となる。特定のアプリケーションでは小さいように見えるが、京コンピューターに向けて様々なスケーラブルなアプリケーションが開発されてきた。このようなスケーラブルなアプリケーションでは、使用するノードの数を増やすことで問題のサイズを大きくすることができる。

3.1.2.4 キャッシュの構造

キャッシュアーキテクチャを設計するための眶は、多くのアプリケーションに対して高いヒット率を提供し、データをメモリからフルバンド幅で供給する時にボトルネックを解消することである。ダイ上の領域のサイズと消費電力の制約の下でキャッシュパフォーマンスを最適化するために、ラインサイズ、ウェイ数、容量などのパラメータの組み合わせを検討した。

表 3.3: L1 キャッシュのラインサイズを変化させた時のカーネルの性能 (Normalized to the Estimated Performance of 256 Bytes)

Kernel (Application)	128 Byte	256 Byte
Kernel (Adventure)	99 %	100 %
Kernel (FFB)	78 %	100 %
Kernel A (NICAM)	86 %	100 %
Kernel B (NICAM)	56 %	100 %

表 3.4: L1/L2 キャッシュサイズとウェイ数を変化させたときのカーネルの性能 (Normalized to the Estimated Performance of L1:64KiB4Way and L2:6MiB24Way)

	L1	64K4W	64K8W	64K4W
Kernel (Application)	L2	6MiB24W	6MiB24W	8MiB16W
Kernel (Adventure)		100 %	100 %	100 %
Kernel (FFB)		100 %	101 %	101 %
Kernel B (NICAM)		100 %	**94** %	**110** %
Kernel C (NICAM)		100 %	101 %	101 %
Kernel D (NICAM)		100 %	**98** %	101 %
Kernel (Seism3D)		100 %	100 %	99 %

データの容量を増やして、ラインサイズが同じである場合、データのためのダイ面積はもちろんタグのためのダイ面積も増える。タグとデータのキャッシュのダイ面積は、使用可能な RAM マクロの種類によっても異なる。ウェイの数が増えると、データのサイズが同じであっても、データパスの複雑さと制御ロジックの量が増えることになる。さらに、L1 キャッシュのウェイ数とラインサイズを変更すると、ダイ面積に影響することが分かった。ラインサイズを 256 バイトから 128 バイトに変更すると、ダイの面積は 5 % 増加する。

一つの CMG をシミュレートするシミュレーターで、ターゲットアプリケーションから抽出したいくつかのカーネルを実行することにより、キャッシュ構成が性能に与える影響を調査した。表 3.3 に示すように、256 バイトから 128 バイトに変更すると、キャッシュミスの数が増加し、一部のカーネルで性能の低下が見られた。細粒度のメモリアクセスを行うアプリケーションには、キャッシュサイズが小さくてもよいという利点があるが、256 バイトのラインサイズでよい性能が得られる HPC アプリケーションも多い。キャッシュラインサイズを 256 バイト以上に変更しても、特に大きな利点は見つからなかった。その上、キャッシュラインが広くなると、必要なデータパスも広くなるため、ダイ面積が大きくなり、レイテンシが長くなる。表 3.4 に、L1 と L2 のキャッシュサイズとウェイ数を変更した場合の性能を示す。 4 ウェイから 8 ウェイに変更しても、性能の大幅な向上は見られなく、消費電力への影響は小さいことがわかる。その結果、ラインサイズを 256 バイトとして、64KiB の 4 ウェイ L1 キャッシュ、および 8MiB の 16 ウェイ L2 キャッシュを選択することにした。

set associative のキャッシュ内のデータにアクセスするための電力を節約できるように設計した。すなわち、各ウェイからのデータの読み込みとタグ検索を同時に行うことにより、遅延を減らす

ことができるが、タグが一致しないデータは使用されないことになるため、電力を浪費する可能性がある。データアクセスはタグの一致後に実行されるように設計した。待ち時間は長くなるが、スループット重視の HPC アプリケーションの場合、性能への影響は小さくなる。ベクトルアクセス用の L1 キャッシュと L2 キャッシュにこの設計を適用したことにより、HPL の電力量が 10 % 削減され、性能がほとんど低下しないことが分かった。

3.1.2.5 アウトオブオーダー (O3) 資源

マイクロアーキテクチャは、富士通が設計・開発したアウトオブオーダーアーキテクチャである。O3 資源のパラメタには以下のものがある。

- リザベーションステーションのエントリ数
- リオーダーバッファ(ROBs) の数
- リネームレジスタ（汎用レジスタと浮動小数点数/SIMD レジスタ）の数
- ロード/ストアキューのエントリ数

まず、O3 リソースの比率を決定するために、プロセッサシミュレータでいくつかのカーネルを実行して、これらのパラメタの組み合わせを調査した。これによって、求められた比率を保ちながら、性能とダイサイズへの影響についてトレードオフを評価し、O3 リソース量を決定した。図 3.2 に、O3 のリソース量を変化させた場合のダイサイズの変化、評価を行った各カーネルの相対性能とそれらの平均を示す。O3 リソース量が大きいほど性能が向上する可能性があるものの、O3 リソースを増やすとダイサイズに対する O3 リソースが占める面積が大きくなることがわかる。

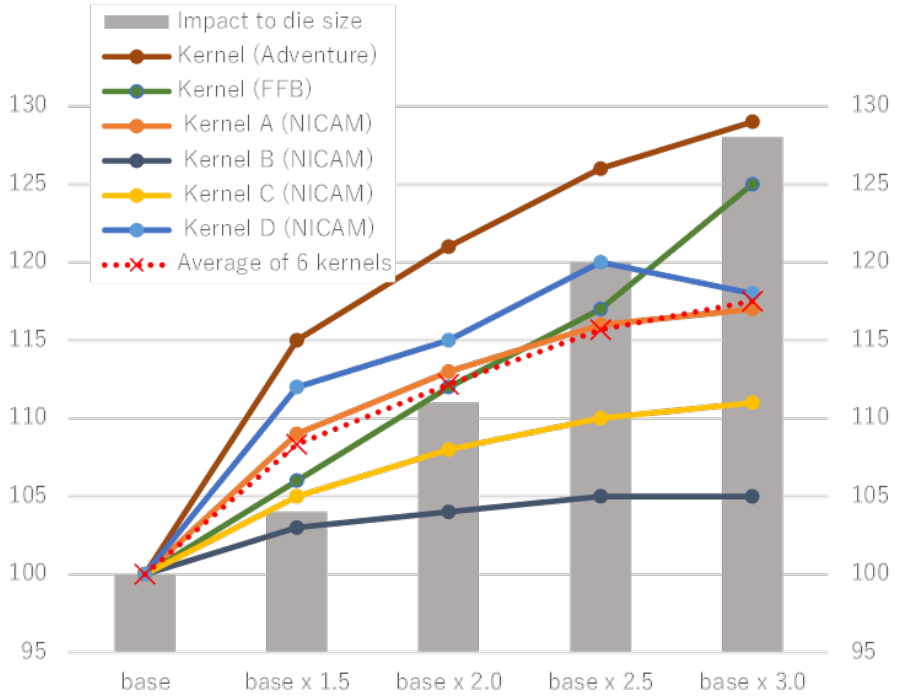
O3 リソースに関する決定は、コアの設計で最も難しい問題であった。以下に理由を示す。

- 基本設計の段階では、パラメーターの妥当なセットを評価するには、コンパイラの最適化技術がまだ未成熟であった。
- 富士通は O3 のプロセッサの設計経験があったものの、その命令セットは独自の拡張機能を備えた SPARC の命令セットで、Armv8 とは大きく異なっていた。

図 3.2 に示した “base x 2.0” と “base x 2.5”が候補となったが、ダイサイズへのインパクトを考慮して、“base x 2.0”を選択することとなった。最終的な O3 パラメタについては、A64FX のマイクロアーキテクチャのドキュメントに記載されているように決定したが、ダイサイズとのバランスに配慮して決めたものである。

3.1.2.6 ターゲットアプリケーションからのフィードバック

コデザインの取り組みを行うなかで、ターゲットアプリケーションの性能を向上させるためにアーキテクチャを改善するための問題点がアプリケーション側からフィードバックされ、それに対処する機能の設計を行った。このフィードバックにより実装された拡張機能は次の機能である。



Base Set: #RS=40, #ROB=64, renaming regs #FP=48, #GP=32,
#Fetch Port=20, #Store Port=12, #write Buffer=4

図 3.2: O3 資源量を変えたときのカーネルの性能とダイサイズへの影響 (Size and performance are shown in %, relative to "Base Set")

- **Combined gather 機能：** gather 命令による間接メモリアクセスの場合、2 つの隣接するレジスタにロードする要素が 128 バイトに整列されたブロック内にある場合、キャッシュアクセスへの要求は 1 つの要求に結合される。これにより、gather 命令のデータスループットが最大 2 倍となる。間接ロード命令を用いるカーネルの性能が、「カーネル（FFB）」で 36 %、「カーネル D（NICAM）」で 20 % 向上することが分かった。

- **L1 キャッシュ置き換え時のビジーサイクル数削減:** いくつかのアプリケーションでは、L2 キャッシュ上のデータの再利用性が高く、L1 キャッシュが頻繁に置き換えられる時、L1 キャッシュの置き換えに要する時間で L1 キャッシュのビジー率が大きくなり、性能のボトルネックとなった。

  L1 キャッシュの置き換え時、L1 キャッシュ上のデータが更新されていない場合に、キャッシュタグの更新のみを行うことにして、eviction 時のパケット数を半分にした。

3.1.2.7 ノード間インターコネクト

大規模なアプリケーションでの京との性能互換性のために、6 次元トーラスネットワークである Tofu ネットワークトポロジを選択することとした。

製造時に利用できるテクノロジであり、十分に成熟している技術の中から、インターコネクトのリンクには 28 Gbps SerDes を選択することとした。リンク速度はリンクあたり 6.8 GB/s である

表 3.5: DGEMM と Stream によるパワーノブの評価

パワーノブ	デフォルト値	設定値	DGEMM		Stream	
			性能	電力	性能	電力
周波数	2.0GHz	1.6GHz	80%	82%	98%	89%
命令発行	4 命令	2 命令	59%	95%	100%	98%
FPU	2	1	52%	84%	100%	100%
EXU	2	1	96%	100%	100%	100%
メモリ	100%	50%	100%	100%	62%	84%

が、ネットワークとチップ内のメモリ間の DMA エンジンである Tofu ネットワークインターフェイス（TNI）の数を 6 として、ノードあたりの injection バンド幅は 40.8 GB/s とした。

ターゲットアプリケーションから通信パターンを抽出し、解析的なモデルによって通信パフォーマンスを推定した。多くのターゲットアプリケーションにおいて、多く使われるのは隣接通信パターンであり、近接のノードとの通信が主になる。この場合、injection バンド幅が 40.8 GB/s の Tofu ネットワークで十分であると結論になった。

いくつかのアプリケーションでは全対全通信が用いられており、別に追加の専用全対全ネットワークについてそのメリットと実現可能性について検討を行ったが、コストに見合わないという結論から、採択には至らなかった。代わりに、ターゲットアプリの QCD アプリケーションの性能向上を目的に、3 つの倍精度浮動小数点数の総和演算を、Tofu インターコネクトに実装することとした。

新しいバージョンのインターコネクトは、TofuD とよばれ、TofuD の詳細については、IEEE Cluster 2018 の論文 [18] に記述されている。

3.1.2.8 電力制御のコデザイン

エクサスケールシステムを構築するための重要なチャレンジの一つは、消費電力を削減することである。我々は電力削減のためにいくつかの電力制御機構を検討した。我々の電力制御に対する基本的なポリシは、性能を下げずに余分な電力を削減することにより、電力効率を高めることである。このポリシを満たすために、我々はパワーノブと呼ぶ電力制御機構を採用した。これをアプリケーションの特性に応じてオン・オフすることにより、アプリケーションごとに電力効率のチューニングを行う事を想定している。

最初は以下に示すパワーノブを検討した。クロック周波数、SIMD 幅、浮動小数点演算ユニット数、整数演算ユニット数、L2 キャッシュのウェイ数、同時発行命令数、リザベーションステーションやリネームレジスタなどのアウトオブオーダリソースサイズ、メモリスロットリング。どのパワーノブを実装するかを決定するために、最初は性能電力予測ツールを用いて消費電力の見積もりを行った。その後、サイクルベースのシミュレータを用いてより正確な見積もりを行った。電力の見積もりには、計算性能がボトルネックとなるワークロードとして DGEMM カーネルを、メモリ性能がボトルネックとなるワークロードとして Stream ベンチマークを用いた。それに加えて、ターゲットアプリから抽出したカーネルプログラムによる評価も行った。評価ではあまり効果のなかったアウトオブオーダリソースの制御を不採用とし、命令発行数のパワーノブのみを実装することとした。また、キャッシュ設計の節で説明している通り、A64FX のキャッシュ機構は低電力を目指した設計となっており、ウェイ数の制御は電力削減の効果が小さいため不採用とした。

表 3.5 に DGEMM と Stream による各パワーノブの評価結果を示す。例えば、周波数を 2.0 GHz から 1.8 GHz に低くすると、DGEMM の性能は 20 %低下し、電力は 18 %削減される。一方、2

表 3.6: ターゲットアプリの電力モードによる電力予測 (性能と電力はノーマルモード (N) を 1.0 とした相対値)

アプリ	電力モード	ブースト		エコ		ブーストエコ	
		性能	電力	性能	電力	性能	電力
GENESIS	ブースト	1.09	1.20	1.00	0.80	1.09	0.96
Genomon	ブースト	1.10	1.17	-	-	-	-
GAMERA	ブーストエコ	1.06	(1.14)	1.00	0.81	1.06	0.89
NICAM+LETKF	ブーストエコ	1.07	(1.18)	0.97	0.79	1.04	0.91
NTChem	ブースト	1.08	1.21	0.57	0.69	0.62	0.83
ADVENTURE	ノーマル	1.07	(1.21)	0.90	0.85	0.98	1.00
RSDFT	ブースト	1.06	1.20	0.71	0.80	0.77	0.90
FFB	ブーストエコ	1.10	(1.17)	1.00	0.80	1.10	0.94
LQCD	ブーストエコ	1.05	1.17	1.00	0.74	1.05	0.83

つある FPU の 1 つだけを利用するパワーノブでは、DGEMM の性能が 48 %に低下するが、電力は 16 %しか削減されない。

メモリスロットリングは、メモリアクセスの少ない、計算中心のアプリケーションでは、電力削減に有効であることが期待され、初期に評価を行った HMC では効果が確認された。しかし、HBM ではメモリ電力はアクセス頻度に依存し、アクセス頻度が低い時には電力も小さくなっているため、その効果は小さかった。

メモリ律速のアプリケーションでは、演算ユニットの利用率が高くなく、演算ユニット数の制御が電力削減に有効であることを期待していた。しかし、表 3.5 に示す通り、Stream で FPU を 1 つに制御するパワーノブを適用しても、性能も電力も変化が確認できなかった。性能が同じであることは予想通りであるが、電力削減も確認できなかったため、原因の調査を行った。電力が削減できなかった理由は、電力変動が起きても安定動作を行うための機構が働き、演算ユニットが動かないときにも、一定の電力が消費されるようになっていたためであった。これは、最大電力が高い HPC 向けのプロセッサを安定して動作させるためには必要な機構である。この電力安定化と電力削減とを両立させるために、エコモードと呼ぶ新しい電力モードを採用した。これは、FPU を 1 つにするパワーノブを適用するとともに、低くなる新たな最大電力に合わせて電力安定化の制御を行うモードである。このエコモードの評価については、以下の実機による電力評価で示す。

電力効率を向上させるために、クロック周波数を維持したまま、できるだけ低い電圧で動作させることが重要である。一方で、たとえ消費電力は上がったとしても、性能を少しでも上げたいという要望も存在する。その要望に応えるため、クロック周波数を 2.2 GHz に上げたブーストモードを採用した。しかし、周波数を上げるためには電源電圧を上げる必要があり、電力効率はノーマルモードよりも低下してしまう。また、クロック周波数は全コアで共通のため、一部のコアだけをブーストモードにすることは出来ず、全コアがノーマルモードか、ブーストモードかのいずれかとなる。

エコモードとブーストモードは独立して設定することができるため、次の 4 つの電力モードが選択可能である：ノーマル、ブースト、ノーマルエコ、ブーストエコ。ターゲットアプリについて、性能電力予測ツールを用いて各電力モードの性能と消費電力の評価を行い、想定している最大電力内で性能を最大にできる電力モードを推定した。表 3.6 にその結果を示す。各値は、ノーマルモードの性能と電力をベースとした相対値である。括弧付きの値は最大電力を超えている値を示す。ブーストエコではノーマルモードに比べて性能を高めつつ、電力削減を実現できる場合があり、9 つのターゲットアプリケーションのうち、4 つでブーストエコが選択されている。

パワーノブはサンディア電力 API [10] に基づくライブラリを用いてユーザプログラムから制御

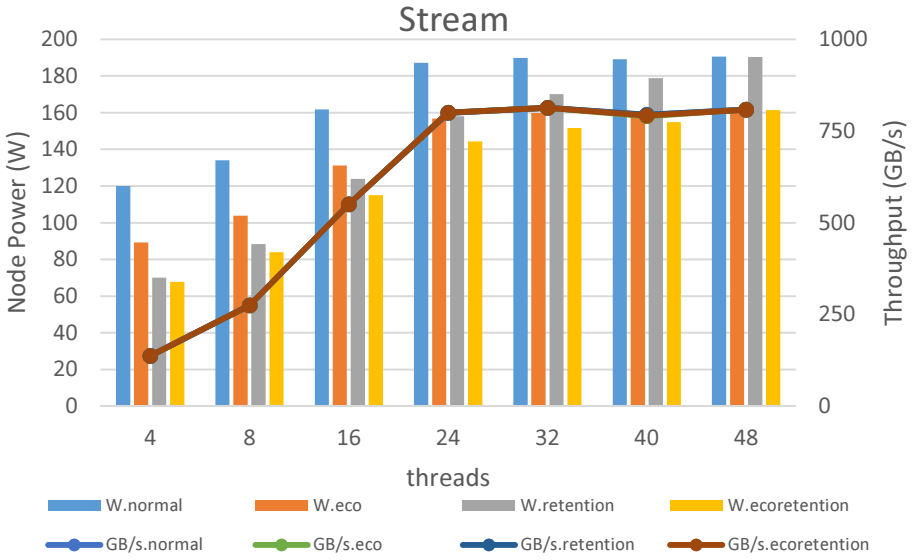


図 3.3: Stream ベンチマークにおけるエコモードとコアリテンションの効果

することが可能である。本 API は現在は電力 API コニュニティー仕様 [9] として開発が進められている。さらに、本 API を用いて、実測したノード電力やパフォーマンスカウンタから推定したノード電力を得ることができる。

大規模システムではノード数が非常に多いため、ジョブが割り当てられていないアイドル状態のノードの消費電力もシステム全体では数メガワットに達することもある。そのため、アイドル状態の電力も削減方法を検討する必要である。アイドル状態のコアの電力を削減するために、コアリテンションと呼ぶ新しい状態を定義した。この状態では電力安定化のための電力を削減することなどにより、アイドル状態の電力を削減している。さらに、ノード全体で、ノードリテンションという状態を定義した。これは 1 つのアシスタントコアを除く全てのコアがコアリテンションの状態にあり、ノード電力を約半減できる。コアリテンションの遷移はコアごとに行える。アプリケーションの実行に使われないコアが存在する場合には、コアリテンションにより電力削減を期待できる。

図 3.3 は Stream ベンチマークを、スレッド数を変化させて実行した結果である。ノーマル、エコ、リテンション、エコリテンションの 4 つの電力モードについて比較を行った。ノーマルおよびリテンションでは 2 つの FPU を利用するのに対して、エコおよびエコリテンションでは 1 つの FPU のみを利用する。ノーマルおよびエコでは、スレッドが実行されないコアは通常のアイドル状態であるのに対し、リテンションおよびエコリテンションではコアリテンション状態に遷移する。

図 3.3 では各モードのメモリスループットを折れ線グラフで示しているが、全てのモードでスループットはほぼ同じであり、折れ線グラフはほぼ重なっている。ここでは、スレッドは各 CMG にインターリーブして割り当てるスレッドスケジューリングを用いている。スループットは 24 スレッドまでスレッド数に比例して増加し、約 800 GB/s で飽和している。これは理論ピークスループットの約 80%に相当する。

図 3.3 では各モードの実測ノード電力を棒グラフで示している。スレッド数が 4 の場合、ノーマルのノード電力は約 120 W であるのに対して、エコの電力はそれより 25%低い約 90 W である。リテンションでは 4 コアのみがアクティブであり、残りの 44 コアはコアリテンション状態にあるため、ノード電力はノーマルより 40%低い約 70 W である。スレッド数が増加するにつれて、コ

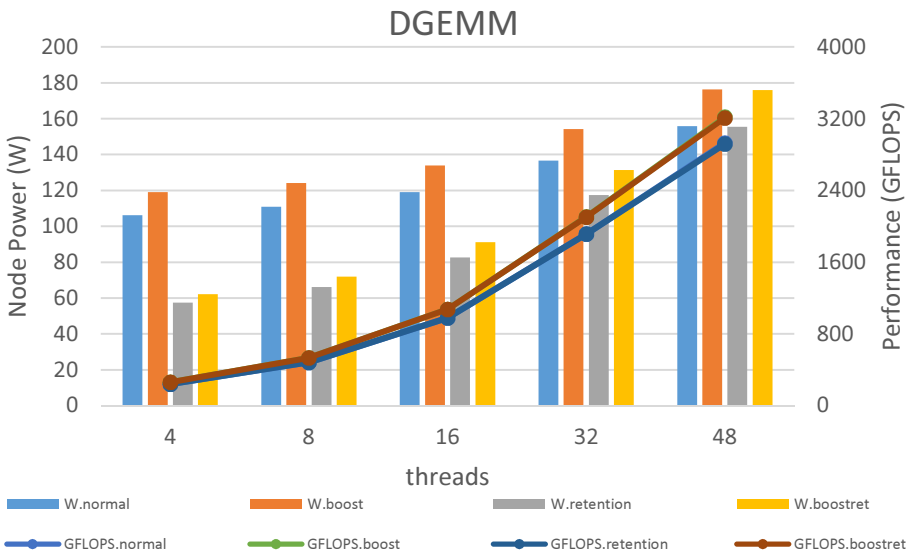


図 3.4: DGEMM ベンチマークにおけるブーストモードとコアリテンションの効果

アリテンション状態のコア数が減るため、コアリテンションによる電力削減効果は減っていく。24 スレッドでスループットが飽和したときには、ノーマルの電力は約 190 W であるのに対し、エコリテンションのノード電力は 23%低い 145 W である。

図 3.4 は DGEMM ベンチマークを、スレッド数を変化させて実行した結果である。ノーマル、ブースト、リテンション、ブーストリテンションの 4 つの電力モードについて比較を行った。ノーマルおよびリテンションでは 2.0 GHz で動作するのにに対して、ブーストおよびブーストリテンションでは 2.2 GHz で動作する。ノーマルおよびブーストでは、スレッドが実行されないコアは通常のアイドル状態であるのに対し、リテンションおよびブーストリテンションではコアリテンション状態に遷移する。

図 3.4 では各モードの演算性能 (GFLOPS) を折れ線グラフで示しているが、リテンションの有無で性能はほぼ同じであり、折れ線グラフは 2 つにみえる。ブーストの性能は周波数の増加と同じく約 10%ノーマルより高い。性能はスレッド数に比例して増加している。ブーストの 48 スレッド時の性能は約 3200 GFLOPS で、これは理論ピーク性能の約 95%に相当する。

図 3.4 では各モードの実測ノード電力を棒グラフで示している。スレッド数が 4 の場合、ノーマルのノード電力は約 106 W であるのに対して、ブーストの電力はそれより 12%高い約 119 W である。リテンションでは 4 コアのみがアクティブであり、残りの 44 コアはコアリテンション状態にあるため、ノード電力はノーマルより 46%低い約 58 W である。スレッド数が増加するにつれて、コアリテンション状態のコア数が減るため、コアリテンションによる電力削減効果は減っていく。32 スレッドでは、ノーマルの電力は約 137 W であるのに対し、ブーストリテンションのノード電力は 4%低い 132 W であるが、性能は 10%高い。このように、実行に使用されないコアがあるような場合には、ブーストモードを使いつつ、使用しないコアをコアリテンションとすることにより、電力を下げつつ性能を向上させる可能性がある。

電力モードのより詳細な解析については [34] を参照のこと。

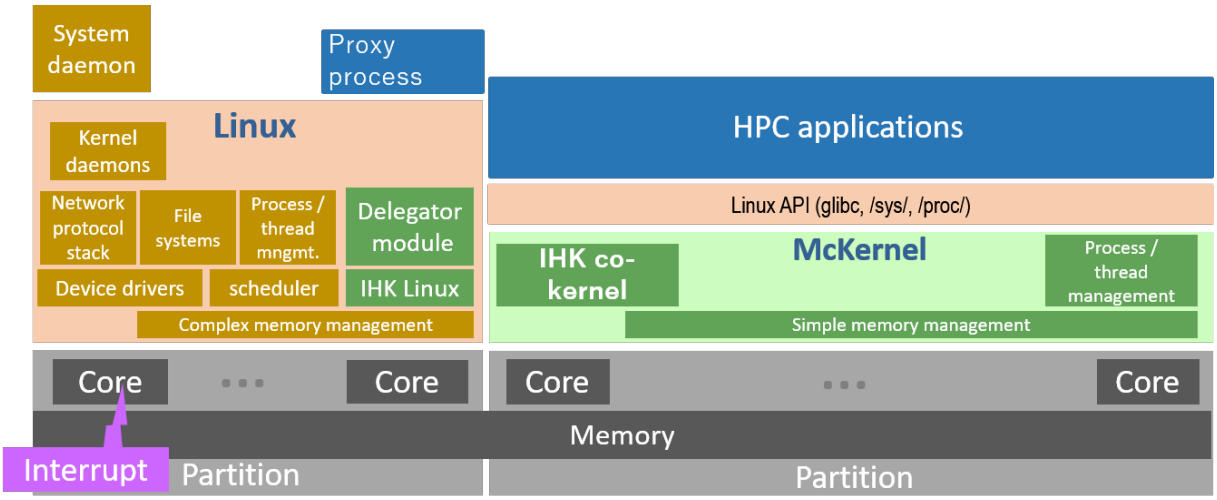


図 3.5: McKernel

3.2 オペレーティングシステム

オペレーティングシステム (OS) は計算機上でのアプリケーション実行・終了をサポートし、アプリケーションがハードウェア資源を利用するのをサポートするためのソフトウェアである。基本設計時、「京」における課題、新しいハードウェアに対応すべき機能を整理し、詳細設計ならびにソフトウェア調整期間を通して実装・調整を行った。

3.2.1 アプローチ

3.2.1.1 「京」における課題

1. Linux Distribution と継続的最新 Linux カーネル提供
   「京」の OS は Linux カーネルを採用していたが、Intel 系 CPU とは異なり、SPARC CPU 向け Linux Distrition (プログラミング環境やアプリケーションを使えるようにしたパッケージ) が存在しなかった。このため、研究者が Linux 上で利用しているプログラミングツールが必ずしも「京」では利用できなかった。また、Linux カーネルは日進月歩進化している。「京」の Linux カーネルは独自改良していたため、最新 Linux カーネル機能を取り込むことが難しかった。

2. 使い勝手の向上
   Linux Distribution は主にプログラミング環境でありスーパーコンピュータで利用できるアプリケーションは支援していない。アプリケーションがオープンソースで提供されていてもスーパーコンピュータ固有の設定の違いにより容易にインストールできるとは限らない。また、Linux カーネルは必ずしも HPC アプリケーションの実行に適した OS 機能を提供しているとは限らない。

3. 計算機科学者向け利用環境
   「京」のユーザの殆どは計算科学者であり、 計算機科学分野のユーザは限られていた。これ

は上記に述べたプログラミングツールの不足や最新 Linux カーネルに対応していないということだけでなく、新しい Linux 機能を試す実験環境が提供されていなかったことによる。

3.2.1.2 新ハードウェア対応

1. OS ノイズ削減
   「京」の 2 倍の計算ノード数、12 倍の CPU コア数となる 富岳において、「京」以上に OS ノイズ削減対策が必要である。

2. ラージページ対応
   Arm アーキテクチャのページテーブルサイズは、4KiB、16KiB、2MiB、32MiB、64MiB、512MiB、1GiB と柔軟にサイズを決められる（組み合わせ制限あり）。ターゲットアプリケーション実行時の TLB キャッシュミス率とメモリ利用率（ラージページを用いるとページサイズ未満の、利用されず、割り当て済みゆえに再利用もされない領域が増加するため）のバランスを考慮してノーマルページサイズならびにラージページサイズを決める必要がある。

3. メニーコア
   富岳の CPU は 48 コアを有するメニーコア型アーキテクチャである。ノード内での並列処理を効率よく実行するために、プロセスのように変数を共有せず、スレッドのように同一アドレス空間内で動作するようなノード内実行環境の実現が望まれている。

表 3.7: メモリ領域ごとのラージページ利用可否と利用可能サイズ

<table>
<tr><th colspan="4" rowspan="2">メモリ領域</th><th rowspan="2">McKernel拡張
THP</th><th colspan="2">hugeTLBfs</th></tr>
<tr><th>Linux</th><th>McKernel</th></tr>
<tr><td colspan="4">.text</td><td colspan="3">-</td></tr>
<tr><td colspan="4">.data</td><td>2MiB – 16GiB</td><td colspan="2">2MiB</td></tr>
<tr><td colspan="4">.bss</td><td>2MiB – 16GiB</td><td colspan="2">2MiB</td></tr>
<tr><td colspan="4">Stack</td><td>2MiB – 16GiB</td><td colspan="2">2MiB (富士通コンパイラ使用時のみ)</td></tr>
<tr><td rowspan="2">Heap</td><td colspan="3">malloc</td><td>2MiB – 16GiB</td><td colspan="2">2MiB</td></tr>
<tr><td colspan="3">sbrk</td><td>2MiB – 16GiB</td><td colspan="2">-</td></tr>
<tr><td rowspan="11">mmap</td><td colspan="3">maloc</td><td>2MiB – 16GiB</td><td colspan="2">2MiB</td></tr>
<tr><td colspan="3">hugeTLBfs</td><td colspan="3">2MiB</td></tr>
<tr><td rowspan="2">Anonymous</td><td colspan="2">Auto</td><td>2MiB – 16GiB</td><td colspan="2">-</td></tr>
<tr><td colspan="2">Manual</td><td>2MiB – 16GiB</td><td>2MiB – 16GiB</td><td>2MiB – 16GiB</td></tr>
<tr><td colspan="3">Thread stack</td><td>2MiB – 16GiB</td><td colspan="2">2MiB</td></tr>
<tr><td colspan="3">Dynamic link library</td><td colspan="3">-</td></tr>
<tr><td rowspan="4">Shared Memory</td><td rowspan="2">System V IPC</td><td>Auto</td><td>2MiB – 16GiB</td><td colspan="2">-</td></tr>
<tr><td>Manual</td><td>2MiB – 16GiB</td><td>2MiB – 16GiB</td><td>2MiB – 16GiB</td></tr>
<tr><td colspan="2">POSIX</td><td colspan="3">-</td></tr>
<tr><td colspan="2">XPMEM</td><td>2MiB – 16GiB</td><td colspan="2">-</td></tr>
</table>

3.2.2 コデザイン成果

前節「アプローチ」で述べた課題に対する取り組み結果を示す。

3.2.2.1 「京」における課題

1. Linux Distribution と継続的最新 Linux カーネル提供
   富士通は、Linux に独自修正せずに追加カーネルモジュールによって必要とするカーネル機能を実現した。Linux の富岳移植に際して発見した問題点とその修正は Linux コミュニティに報告しメインストリームに反映した。Linux Distribution として Red Hat Enterprise Linux 8 (RHEL8) を導入することとした。Linux 自体に独自拡張を行なわなかったことにより、将来 RHEL8 の最新版に速やかに対応できる。

2. 使い勝手の向上
   RHEL ならびに RHEL と互換性のあるフリー Linux Distribution である CentOS はアカデミック・産業界で広く使われている。RHEL8 の導入により今までスーパーコンピュータを利用していなかった人が利用する時の敷居を下げ、結果として使い勝手の向上に貢献する。

   スーパーコンピュータ向けオープンソースソフトウェアを容易にインストールするツールとして EasyBuild および Spack の導入を検討した。EasyBuild は欧州 CERN で開発されたツールで、フランス CEA において積極的に利用されている。Spack は Lawrence Livermore National Laboratory が中心になり開発されたツールであり、米国 ECP (Exascale Computing Project) をはじめ、米国を中心に広く使われている。検討の結果、Spack のほうが前提となるツール類の依存関係を柔軟に記述でき、また、既に登録されているオープンソース群 (5344 本 2021 年 3 月現在) が充実しているため、Spack を採用した。しかし、Spack は Arm アーキテクチャ対応が遅れていたため、理研と富士通が協力して Arm アーキテクチャ対応を進めた。表 3.8 に示す通り、2019 年 6 月時点で、x86_64 系/GCC では Spack に登録されているソフトウェアの 73%が利用できたのに対し、aarch64 系/GCC では 68%、富士通コンパイラを利用する場合では 34%しか利用できなかった。様々な対応を進めた現在では、x86_64 系/GCC では Spack に登録されているソフトウェアの 84%が利用できるのに対し、aarch64 系/GCC では 81%、富士通コンパイラでは 75%が利用できるようになっている。

表 3.8: 対応開始時 (2019/6) と最終結果 (2021/2) のビルド可能率比較

アーキテクチャ	コンパイラ	対応開始時 (2019/6)	最終結果 (2021/2)
x86_64	GCC	72.70%(2357/3242)	84.19%(4499/5344)
aarch64	GCC	67.82%(2199/3242)	81.08%(4333/5344)
aarch64	Fujitsu Compiler	34.20%(1109/3242)	74.59%(3986/5344)

HPC ワークロード向け OS 機能を提供する McKernel を開発した。McKernel は東京大学で開発された GPL ライセンスに基づくオープンソースである。富岳のデフォルト OS は Linux であり計算ノード上の全ての CPU コア上で Linux が稼働する。ユーザが McKernel を利用する場合、図 3.5 に示すとおり、McKernel が Linux カーネルと共存して各計算ノード上に配置される。「京」と違い、富岳は OS アクティビティ用の CPU コアを有する。計算用コア (48 コア) に加えて、計算兼 IO 処理ノードに 4 基、それ以外は 2 基のコアが装備されている。図において Linux カーネルはアシスタントコア上で稼働しアプリケーションが動作するコア上で McKernel が稼働する。McKernel はアプリケーションが Linux 機能を使わない限り OS ノイズは発生しない。また、性能クリティカルなシステムコールは McKernel で処理し、それ以外のシステムコールの処理を Linux に依頼することで、POSIX/Linux API をサポートする。結果としてユーザーは Linux でコンパイルした実行可能ファイルをそのまま実行でき

る。また、Linux とは異なるメモリ管理機能によりメモリ割り当て時の性能向上が得られている。

富岳ハードウェアが利用できない間、Intel メニーコアアーキテクチャXeon Phi 上での実装を進めた。富岳および Xeon Phi が搭載されている Oaskforest-PACS スーパーコンピュータを使った Linux カーネルと McKernel の評価論文を発表した [22]。

3. 計算機科学者向け利用環境
仮想計算機 KVM を提供する。これにより、富岳が提供している Linux カーネルとは異なる版の Linux を動かせると共に新しいカーネル機能の実験が可能となる。

3.2.2.2 新ハードウェア対応

1. OS ノイズ削減
「京」と違い、富岳は OS アクティビティ用の CPU コアを有する。計算用コア (48 コア) に加えて、計算兼 IO 処理ノードに 4 基、それ以外は 2 基のコアが装備されている。これらコア上で OS デーモンの実行や割り込み処理を行う。アシスタントコアの数は OS デーモンが必要とする CPU 処理能力から決定した。富士通による OS ノイズに関する評価は富士通テクニカルレビューで公開された [16]。

2. ラージページ対応
Linux および McKernel が提供するメモリ領域ごとのラージページのサイズを表 3.7 に示す。以下の環境変数設定で McKernel が提供する THP を利用できる。

```
export XOS_MMM_L_HPAGE_TYPE=thp
```

3. メニーコア
ノード内での並列処理を効率よく実行するために、Process-in-Process（PiP）を研究開発した。PiP は、これまでのマルチプロセス（例えば MPI）、マルチスレッド（例えば OpenMP）と並ぶ、新しいノード内並列実行モデルを提供する。PiP は複数の PiP タスク（これまでのプロセスに相当する）を同じアドレス空間で実行する。ここで PiP タスクは、マルチスレッドと異なり、個々の PiP タスクの静的変数はそれぞれの PiP タスクが独自に確保されている。このため、これらの変数の値の更新時に排他制御の必要がない。また、マルチプロセスの場合、それぞれのプロセスは独立したアドレス空間を持つため、プロセス間の通信オーバヘッドが大きくなってしまう欠点があったが、PiP では同じアドレス空間を共有するため、単純でオーバヘッドが少ないメモリアクセスにより PiP タスク間のデータ交換が可能となるという利点もある。

PiP は、Position Independent Executable（PIE）、Linux の `clone()` システムコールまたは POSIX の `pthread_create`、そして `dlmopen()` 関数がサポートされていれば実装できる。このため、PiP の実装は全てユーザレベルのライブラリであり、特別な OS カーネルやコンパイラは不要である。ここで `dlmopen()` 関数は `dlopen()` と異なり、新しい名前空間でライブラリやプログラムをロードすることができる。残念ながら、通常の GLIBC は `dlmopen()` で名前空間の数の上限が 16 と規定されており、これが PiP タスク数の上限となってしまい、現在のメニーコアプロセッサでは全く不十分である。このため、PiP ソフトウェアパッケー

ジでは、この制約を緩和するために、別途修正された GLIBC も提供しており、これを使うことで最大 300 までの PiP タスクが生成可能となっている。

PiP タスク群は、ページテーブルを共有しているため、コンテキスト切替時間はスレッドと同程度に高速である。マルチプロセス並列では、共有メモリ領域を設けて通信オーバヘッドを削減する方式が広く使われているが、この場合に OS カーネル内のページテーブルのメモリ消費に問題が生じる可能性がある。また、XPMEM を用いて、他のプロセスのメモリ領域を自分自身のアドレス空間にマップする方式も提案されているが、マップ生成や削除のオーバヘッドが非常に大きく、実際のマップ生成や削除の回数を減らすためにキャッシュ的な機構を導入するなどの実装上の工夫が必要である。また、他のプロセスのメモリページをマップすることから同じアドレスにマップすることは難しく、ポインタを含む領域をマップしてそのままポインタを参照することはできない。しかしながら、PiP を用いることで新たにメモリをマップする必要はないため領域生成や削除のオーバヘッドが存在しない。また、同じアドレスでアクセスできるためポインタを含む領域のアクセスにも特別な配慮の必要としない。このため必要に応じて異なる PiP タスクのデータを高速かつ柔軟に参照することが可能となっている。

公開されている PiP ソフトウェアパッケージは、PiP のタスクをデバッグ可能とする pip-gdb も含まれている。Pip-gdb では、個々の PiP タスクを gdb の `inferior` と対応させている。これら PiP のためのパッチが当たった GLIBC や GDB を一括してインストールするために PiP-pip と呼ばれるインストーラもある。また、PiP パッケージ用の spack レシピも spack のメインリポジトリに登録されている。

3.3 通信ライブラリ

3.3.1 アプローチ

3.3.1.1 「京」における課題

業界標準である MPI 通信ライブラリの仕様は MPI Forum で議論され、仕様が確定する前からオープンソースである Open MPI や MPICH に実装されることが多い。仕様が標準化されると、短い期間で新しい仕様に準拠したオープンソースが配布される。「京」の MPI 通信ライブラリはオープンソースである Open MPI をベースにしているが、その修正は多岐にわたり、Open MPI が新しい仕様に準拠したソースコードがリリースされても追従することが出来なかった。富岳においては、Open MPI や MPICH の開発コミュニティと連携を図るとともに、新しい規格が取り込まれた時に即座に対応できるように独自拡張を行わず、拡張する場合にはコミュニティにフィードバックするという戦略を採った。富士通は引き続き Open MPI を基にした通信ライブラリを開発する。理研は MPICH を基にした通信ライブラリ（MPICH-Tofu と呼ぶ）を開発する。2 つのオープンソースを基にした通信ライブラリを提供することにより、より広範な通信機能を提供できる。

「京」において問題となった動的プロセス生成 (`MPI_Comm_spawn`) 時間の短縮および以下の制約を緩和する。

- Tofu のトポロジーを意識したノード割当てができない
- ノード内のプロセス数を指定できない（元ジョブから継承）

- 環境変数 (特に Open MP 関連) を指定できない。(元ジョブから継承)
- ノード内に複数ジョブが混在できない。「京」においてはノード内で複数 MPI_COMM_WORLD を管理できないことによる。

派生データ型の pack/unpack 処理を高速化する手法として OpenMP を用いて Pack/Unpack 処理を並列化できるようにする。MPI 関数呼出し時に、OpenMP ライブラリ関数 omp_in_parallel() を使用して、parallel region 内を実行中か否かは判定可能である。また利用可能なスレッド数についても、omp_get_num_threads() や omp_get_max_threads() を使って獲得可能である。

「京」では 1 PPN (process per node) 実行形態が推奨であったため、計算ノード内のプロセス間通信性能を重要視していなかった。一方、富岳では、4 PPN 実行形態が推奨となるため、計算ノード内のプロセス間での通信性能を最適化する必要がある。基本的には共有メモリを用いて送信データを受信側プロセスの受信領域に直接コピーする 1 COPY 実装方式を採用した。カーネル機能として必要となる共有メモリ実現方式を調査・評価し共有メモリ実現方式を決めた。

3.3.1.2 ターゲットアプリケーションのニーズ

4 PPN 実行形態における Alltoall 通信時間の短縮化がターゲットアプリケーションの実行性能向上に大きく寄与することが指摘された。

京の開発時には一般ユーザに提供する通信ライブラリは MPI 通信ライブラリだけを想定していた。ミドルウェアやプログラミング環境を研究開発している計算機科学ユーザからは MPI 通信ライブラリよりも低レベルの通信ライブラリの提供ニーズがある。また、通信遅延が全体性能に著しく影響するアプリケーションでは可能な限りオーバヘッドのない通信ライブラリ利用のニーズがある。富岳では MPI 通信ライブラリよりも低レベルの通信ライブラリを提供する。

3.3.1.3 新ハードウェア対応

富岳では以下のように新機能の追加、あるいはハードウェア資源の追加がなされている。これらを有効利用して、1 対 1 通信機能改善、集団通信アルゴリズム見直し、使用メモリ量削減に取り組んだ。

1. アシスタントコア
   富岳では OS デーモンの実行や割り込み処理を行うための CPU コアとしてアシスタントコアと呼ばれるコアが装備される。アプリケーションプログラムを実行しているプログラム上でこれら OS 処理は行われない。

2. Tofu ARMW (Atomic Read Modify Write) 機能
   富岳のインターコネクト TofuD では、リモート計算ノードの主記憶上の 4 バイトまたは 8 バイトのデータに対して読み込み、演算、書き込みを不可分に行う Tofu ARMW (Atomic Read Modify Write) が装備される。

3. ネットワークインターフェイス資源
   「京」の Tofu に比べて Tofu-D では、Network Interface が 6 つ、Barrier Gate が 32x6 搭載される。

4. プロセス数増大
「京」においては約 80 K プロセス (1 ノード 1 プロセス)、富岳では、約 636 K プロセス (1 ノード 4 プロセス) 間での MPI 通信が想定される。メモリ使用量はプロセス数に比例して増大する。「京」においても MPI 通信ライブラリが使用するメモリ量を削減してきたが、富岳ではさらなるメモリ使用量削減を行う必要がある。MPI 通信ライブラリが使用するメモリは、通信に必要なネットワークアドレスなどのプロセス情報と通信バッファからなる。それぞれについて検討を行う。

3.3.2 コデザイン成果

3.3.2.1 「京」課題改善

1. MPI 通信ライブラリ機能
富士通 MPI の開発ではオープンソースである Open MPI に独自拡張を行なわずに実装を進めてきたことにより Open MPI の最新版に近い版である 4.0.1 を採用することが出来た。本バージョンは、最新の MPI 通信ライブラリ規格である Version 3 に対応している。 また、MPI 通信ライブラリの次期バージョンである Version 4 のうち、持続的集団通信 (Persistent Collective) 機能を実現した。Open MPI に含まれる既存の `libnbc` (non-blocking collective) を拡張した持続的集団通信スケジューリング・モジュールを実装し、Open MPI 開発コミュニティに公開した。

2. 柔軟な動的プロセス生成
Open MPI はプロセス管理インターフェイスとして PMIx を利用している。PMIx は実行環境デーモンと MPI プロセスの間のインターフェイスを規定する。PMIx インターフェイスには、MPI プロセスが通信路を確立する前に必要となるリモートプロセスのネットワークアドレス等の情報やプロセス間での key-value 通信、同期機能、動的プロセス生成に必要な機能などがある。動的プロセス生成に必要な機能として、`PMIx_Publish()`, `PMIx_Unpublish()`, `PMIx_Lookup()`, `PMIx_Connect()`, `PMIx_Disconnect()`, `PMIx_Spawn()` が提供されている。また、Tofu インターコネクト上の通信やプロセス配置の最適化のヒントとなるトポロジー情報を得るためのインターフェイスとして jTofu を提供する。jTofu では以下の機能が提供される。

   - ジョブの物理形状・論理形状の問い合わせ
   - ランクと物理座標・論理座標の変換
   - 通信の際の経由座標の問い合わせ

3. Pack / Unpack 処理改善
MPI + OpenMP 並列の場合、ユーザ・プログラムが OpenMP の計算用 parallel region 外での MPI 関数への呼出し時、MPI は待機中のスレッドを使用して 派生データ型で生じる Pack/Unpack 処理を並列化可能である。富士通 MPI 通信ライブラリでは、富士通製コンパイラオプションで`-Kparallel` または`-Kopenmp` を指定し実行時に `opal_mt_memcpy` オプションを指定することにより並列化される。

4. ノード内通信強化
XPMEM プロセス間共有メモリ方式を採用しノード内プロセス間通信 1 COPY を実装し

た。プロセス間共有メモリ機能として、KMEM, CMA, XPMEM を評価した。KMEM と CMA はシステムコール内で、つまり OS カーネル内で、アドレス空間をマップしてコピーする方式をとる。一方、XPMEM はアドレス空間をマップするだけで、コピーはユーザ空間で行うため、OpenMP との組合せで複数スレッドによるコピーが可能となる。このため XPMEM を採用した。

3.3.2.2 アプリケーションドリブン

1. 集団通信
   AlltoAll 通信時のノード内通信方式として、富士通製 MPI では 3 ステップによる CMG 間通信方式を設計・実装した。

2. 低遅延通信
   富士通は MPI 通信ライブラリ実装のオープンソースの一つである OpenMPI を基に MPI 通信ライブラリを開発した。また、低遅延通信機能を提供するために Tofu ハードウェアに直接アクセスする API として uTofu を開発し、そのインターフェイスを一般公開した。いくつかの特権アクセスやノード共有資源へのアクセスを除き、uTofu はユーザレベルで実装されている。

   理研は uTofu を用いたユーザレベル通信ライブラリ utf を開発し一般公開した。utf は MPI 通信ライブラリの 1 対 1 通信機能と同様のインターフェイスを有する。utf は米国アルゴンヌ研究所を中心に開発されている MPICH 通信ライブラリを Tofu 向けに移植するために開発されたが、MPI 通信を使っているアプリケーションにおいても utf 通信を利用できる。

3.3.2.3 新ハードウェア対応

1. 1 対 1 通信機能改善
   富士通は、ノンブロッキング通信時の通信処理を進行させるためにアシスタントコア側で通信のためのスレッドを生成できるようにした。また、TofuD が有する ARMW (Atomic Read Modify Write) 機能を用いて MPI 通信の 1 対 1 通信をより少ないメモリ消費量で実施できるようにした。

2. 集団通信アルゴリズム
   富士通は、ネットワークインターフェイス資源が「京」から増えたことにより集団通信 `Allreduce`、`Bcast` アルゴリズムを見直し、これら増えた資源を有効利用するようにした。

3. 使用メモリ量削減
   MPI 通信ライブラリが使用するメモリのうちネットワークアドレスなどのプロセス情報管理部分については Open MPI および Open MPI が使用している PMIx の実装方法を選択することにより使用量を削減した。PMIx 1.2 以降、同一ノード内の実行環境デーモンと MPI プロセス間の通信に共有メモリ領域を利用する機能が追加された。また、Open MPI の実装においても MPI 初期化時にリモートプロセス全ての情報を保持するのではなく、通信プロセスの情報のみを保持する実行モード機能が追加された。これら機能を用いることによりメモリ使用量が削減された。

富士通 MPI 通信ライブラリが使用する通信バッファのメモリ量削減について述べる。「京」では、Tofu インターコネクトによる 1 対 1 通信では初回通信時に送信プロセスごとに受信バッファを割り当てている。受信バッファのサイズによって省メモリモードと高速モードの 2 つが用意されており、実行時オプションで選択できる。デフォルトでは、通信相手プロセスごとに 1 対 1 通信の初回通信時に省メモリ型通信モードのコネクションを張り、その後に 16 回通信すると高速型通信モードに移行する。

受信バッファ数は通信相手プロセス数に比例して増えていく。富岳では受信バッファを共有する機構を TofuD の ARMW 機能を用いて実現した。受信バッファを管理する変数として、64 bit 長内に producer count および consumer count を配置する。この領域は全てのプロセスからリモート操作できるようにする。送信側は TofuD の Remote Atomic Add 操作を用いて producer count を増やすことにより受信バッファ領域を確保する。

3.4 ストレージ＆ファイル I/O

3.4.1 アプローチ

ターゲットアプリケーションのファイル IO パターンを調査し、ファイル共有関係・生存期間とファイルシステムの関係をまとめた (表 3.9)。表において名前空間とはファイルがどの範囲でアクセスできるか、実装の観点では MD server（メタデータサーバ）のアクセス範囲を示している。Temp. Local FS、Temp. Shared FS、Global FS は、それぞれ、ノード内テンポラリファイルシステム、共有テンポラリファイルシステム、グローバルファイルシステムである。

本調査結果からファイル I/O に関して以下の基本設計方針を決めた。

1. 次の 3 階層ストレージを導入する。第 1 階層として半導体ディスク (以降 SSD) から構成されるストレージ、第 2 階層としてハードディスクから構成されるストレージ、第 3 階層としてテープ等から構成されるアーカイブ。第 1 階層ストレージは京におけるローカルストレージの役割と第 2 階層ストレージのキャッシュとして使われる。第 2 階層ストレージは京におけるグローバルストレージと同様である。

2. 京で採用されたステージングを廃止し、非同期 I/O 処理および第 1 階層ストレージをキャッシュ領域に使うことにより高速化を図る。

3. ファイル I/O ミドルウェアとして LLIO を開発する。LLIO は計算ノード上のメモリ領域、第 1 階層ストレージ領域を第 2 階層ストレージのキャッシュあるいは一時ファイル格納領域として使い、Tofu ネットワークが有する RDMA 機能を非同期 I/O 処理に用いることにより高速化を図る。

   なお、LLIO とは別に理化学研究所は DTF(Data Transfer Framework) と呼ぶアプリケーションレベルファイル IO を開発した。DTF は pNetCDF ファイル I/O ライブラリを元にしてアプリケーション間のデータ交換を MPI 通信ライブラリで実装したものである。

4. Lustre ファイルシステムを継続採用する。

5. 耐故障性観点からのハードウェアおよびソフトウェア構成の見直しを行う。

表 3.9: ファイル共有関係・生存期間とファイルシステム

	生成ファイル	名前空間 (MD server)	ストレージの扱い	適用ファイルシステム
プロセス内ローカル	ジョブ終了後削除	計算ノードローカル	In file: 第 1 階層キャッシュ Out file: 第 1 階層に収まらなければ第 2 階層利用	Temporary Local FS. Or Global FS
	ジョブ終了後保存	計算ノードローカル or 第 2 階層	In file: 第 1 階層キャッシュ Out file: 第 1 階層キャッシュ	Temp. local FS → Global FS. Or Global FS
MPI アプリケーション内プロセス間共有	ジョブ終了後削除	ジョブローカル	In file: 第 1 階層キャッシュ Out file: 第 1 階層に収まらなければ第 2 階層利用	Temporary Shared FS
	ジョブ終了後保存	ジョブローカル or 第 2 階層	In file: 第 1 階層キャッシュ Out file: 第 1 階層に収まらなければ第 2 階層利用	Temp. Shared FS → Global FS. Or Global FS
MPI アプリケーション間共有 (Coupler)	ジョブ終了後削除	—	—	DTF
	ジョブ終了後保存	グローバル on 第 2 階層	In file: 第 1 階層キャッシュ Out file: 第 1 階層に収まらなければ第 2 階層利用	Global FS
ジョブ群間共有 (ワークフロー)	ジョブ終了後削除	グローバル on 第 2 階層	In file: 第 1 階層キャッシュ Out file: 第 1 階層キャッシュ	Global FS
	ジョブ終了後保存	グローバル on 第 2 階層	In file: 第 1 階層キャッシュ Out file: 第 1 階層キャッシュ	Global FS

3.4.2 コデザイン成果

前節で述べたとおり、富岳では、ノード内テンポラリファイルシステム、共有テンポラリファイルシステム、グローバルファイルシステムを提供することにし、新規開発した LLIO はノード内テンポラリファイルシステム、共有テンポラリファイルシステム、グローバルファイルシステムのキャッシュ機能を提供する。

3.4.2.1 マウント

- LLIO 機構
  LLIO のファイルシステムタイプは lliofs である。通常の Linux ファイルシステム同様、マウントは mount コマンドの-t オプションを用いて lliofs ファイルシステムタイプを指定する。-o オプションで指定する整数値はプロセスに対して付与するものである。これにより、当該プロセスおよびその子プロセスはマウントされたファイルシステムをアクセスすることができる。表 3.10 に LLIO が提供する階層化ファイルシステムのデバイス名を示す。

  LLIO は計算ノード上で稼働する LLIO クライアント、第 1 階層サーバ上で稼働する LLIO サーバから構成される。計算ノード上の LLIO ファイルシステム名は lliofs である。LLIO はバッチジョブスケジューラが定義するジョブ毎にユニークな整数値であるジョブ ID を用いて、ジョブ毎に独立したファイル名前空間を提供する。計算ノード上の LLIO が必要とす

表 3.10: LLIO が提供するデバイス名

Layer	Device Name
ノード内テンポラリファイルシステム	`local`
共有テンポラリファイルシステム	`share`
グローバルファイルシステム	`global`

表 3.11: 階層ストレージマウント例

第 1 階層ストレージファイルシステム	計算ノード
/mnt/ssd/J/39222/L01	/local
/mnt/ssd/J/39222/S	/share
/mnt/ssd/J/39222/C	/gs1
/mnt/ssd/J/2224649/L01	/local
/mnt/ssd/J/2224649/S	/share
/mnt/ssd/J/2224649/C	/gs1

るジョブ ID および第 1 階層サーバアドレス情報は `lliofs` に対する `ioctl` を用いて設定される。

例えば、ジョブ ID 39222 および 2224649 に対して運用系ソフトウェアにより以下のような設定をしたとする。

- ジョブ ID　39222 および 2224649 に対して、第 1 階層ストレージ群 `SIO1`、`SIO2`、`SIO3` をノード内テンポラリファイルシステム `local` として使用する。
- ジョブ ID　39222 および 2224649 の 共有テンポラリファイルシステム `share` およびグローバルファイルシステムのキャッシュは、ノード内テンポラリファイルシステム同様、第 1 階層ストレージ群 `SIO1`、`SIO2`、`SIO3` が使用される。

上記設定後に以下の `mount` コマンドを実行したとする。なお、オプション-o はジョブ ID を設定する。

```
# mount -t lliofs -o "jobid=39222" local /local
# mount -t lliofs -o "jobid=39222" share /share
# mount -t lliofs -o "jobid=39222" global:/gs1 /gs1
# mount -t lliofs -o "jobid=2224649" local /local
# mount -t lliofs -o "jobid=2224649" share /share
# mount -t lliofs -o "jobid=2224649" global:/gs1 /gs1
```

`mount` コマンドが実行されると、`lliofs` デバイスは `ioctl` で設定された第 1 階層 LLIO サーバに対してマウント要求をする。LLIO は表 3.11 に示すようなマウントを行う。

- 富岳運用ポリシー
  富岳の運用ではジョブ管理系ソフトウェアと連携し、LLIO が提供するファイルシステムをマ

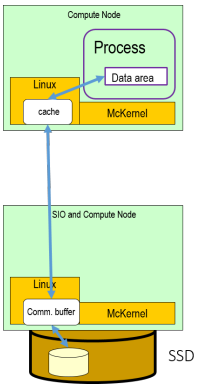


図 3.6: ノード内テンポラリファイルシステム例

ウントする。ファイルシステムのマウント先はジョブ管理系ソフトウェアが参照する LLIO 設定パラメータテーブルで設定される。

3.4.2.2 ノード内テンポラリファイルシステム

3.4.2.2.1 名前空間

計算ノード毎に閉じた名前空間を有する。すなわち、ローカルファイルシステム同様ディレクトリ管理は計算ノードに閉じ、リモート計算ノードからのアクセスはできない。

3.4.2.2.2 生存期間

- LLIO 機構
  ファイルシステムをアンマウントするとそのファイルシステムは削除される。
- 富岳運用ポリシー
  運用ソフトウェアによりジョブ起動前にノード内テンポラリファイルシステムが初期化され計算ノードにマウントする。ジョブ終了後アンマウントされる。

3.4.2.2.3 SIO 選択

計算ノードと同一の BoB に属する SIO 上にファイルシステムが生成される。

3.4.2.2.4 データ格納領域およびデータ転送

図 3.6 に示す通り計算ノード内のキャッシュと SIO が保持する SSD との間のデータ転送は SIO の通信バッファを経由して行われる。データ転送は Tofu の RDMA 機能あるいはメッセージ Send/Receive 機能を使用し、CPU 負荷を減らす。

- テンポラリ領域はジョブ毎に割り当てられ、そのサイズはユーザが実行時パラメータで設定可能である。
- 計算ノードから SIO への書き込み要求は非同期に複数の要求を発行することができる。各書き込み要求は SIO 上の SSD に書き込まれるまで完了しない。SIO は通信バッファに書き込

まれたデータを SSD に書き込んだ後、計算ノードに完了通知を出す。すなわち、計算ノードの書き込み要求の完了は SSD にデータが書き込まれたことを保証している。書き込み要求の同時発行数は SIO あたり 8 とする。

- 計算ノードから SIO への読み込み要求は非同期に複数の要求を発行することができる。各読み込み要求は計算ノードにデータが格納されるまで完了しない。SIO は SSD からデータを通信バッファに読み出した後、計算ノードのキャッシュ領域に RDMA でデータ転送する。

3.4.2.2.5 Direct I/O

ファイルオープン時に O_DIRECT オプションが指定された場合には、Posix File I/O 同様、データは計算ノード内でキャッシュせず直接 SIO の通信バッファに RDMA 転送された後 SSD に書き込まれる。

3.4.2.2.6 計算ノード内キャッシュの扱い

計算ノード内キャッシュの fill-in および flush は非同期に行う。Low/high watermark でデータ転送タイミングを制御する。計算ノード内キャッシュの大きさおよび low/high watermark はユーザが実行時パラメータで設定可能である。アプリケーションのメモリ使用量が少ない場合計算ノード内キャッシュ容量および high watermark 値を大きく設定することにより、ジョブ終了まで SIO にデータが送られずにファイルデータはメモリ内だけに留まる。

キャッシュからデータを読み込んでもキャッシュにデータを残すかどうか、flush されたキャッシュデータを残すかどうか、はユーザが実行時パラメータを設定することにより制御できる。

ファイルオープン時の O_DIRECT オプション指定と同様にキャッシュを介さずにユーザメモリ空間と Direct I/O する機能を提供する。実行時パラメータによって定義されるサイズ以上の write 操作時、計算ノード内のキャッシュを flush した後に O_DIRECT と同様の実行セマンティックスをとる。read 操作時も同様である。

Linux2.4.13 以降提供されている readahead システムコールを提供する。本システムコールがアプリケーションによって発行されると、非同期に計算ノードのキャッシュにファイルデータが転送される。

なお、計算ノード内キャッシュの扱いは、ノード内テンポラリファイルシステムおよびグローバルファイルシステムにおいても同じセマンティックスである。ただし、グローバルファイルシステムにおける O_DIRECT のセマンティックスは第 3.4.2.4.7 節で記載している Direct I/O セマンティックスとなる。

3.4.2.2.7 close 処理

close 処理はストレージまでの書き込み終了を待つ同期処理と終了を待たずにアプリケーションプログラムに処理を戻す非同期処理の 2 つがある。ユーザは LLIO の実行時パラメータで選択可能である。close 処理を非同期処理している場合はプロセス終了時に close 処理が終了しているとは限らない。非同期 close 処理はジョブ終了時に同期する。

なお、close 処理の扱いは、ノード内テンポラリファイルシステムおよびグローバルファイルシステムにおいても同じセマンティックスである。

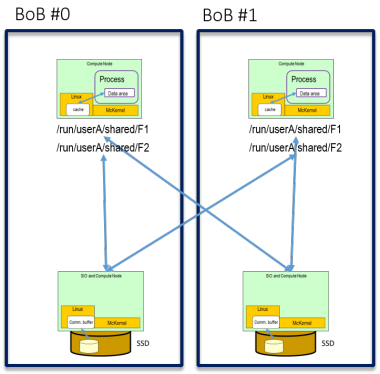


図 3.7: ノード内テンポラリファイルシステム例

3.4.2.3 共有テンポラリファイルシステム

3.4.2.3.1 名前空間

ジョブ毎に閉じた名前空間で管理されるが、同一計算ノード群を使ったジョブ間ではファイルシステムを共有できる。

3.4.2.3.2 生存期間

- LLIO 機構
  ファイルシステムをアンマウントするとそのファイルシステムは削除される。
- 富岳運用ポリシー
  運用ソフトウェアによりジョブ起動前に共有ファイルシステムが初期化され計算ノードにマウントする。ジョブ終了後アンマウントされる。

3.4.2.3.3 SIO 選択

ファイルの inode 番号からどの BoB に属する SIO を選択するか決める。図 3.7 の例では、/run/userA/shared/F1 ファイルは BoB#1 に属する SIO 上に、/run/userA/shared/F2 ファイルは BoB#0 に属する SIO 上にそれぞれ格納されている。共有ファイルシステムはストライピング機能は提供しない。

3.4.2.3.4 データ格納領域およびデータ転送

第 3.4.2.2.4 節で述べているノード内テンポラリファイルシステムと同様。

3.4.2.3.5 Direct I/O

第 3.4.2.2.5 節で述べているノード内テンポラリファイルシステムと同様のセマンティックスである。

3.4.2.3.6 計算ノード内キャッシュの扱い

第 3.4.2.2.6 節で述べているノード内テンポラリファイルシステムと同様である。キャッシュの一貫性については、NFS 同様 Close-To-Open キャッシュ一貫性セマンティックスを採用する。他

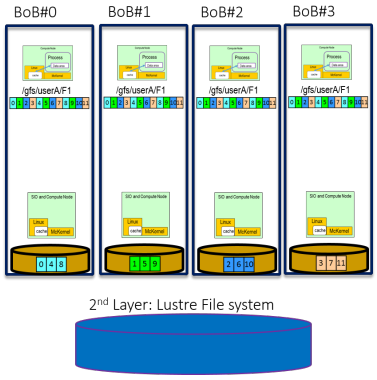


図 3.8: グローバルファイルシステム例

計算ノードとのキャッシュの一貫性は保たれず、ユーザが陽に fsync システムコールを発行しない限りファイルデータの一貫性は保たれない。

3.4.2.3.7　close 処理

close 処理のセマンティックスはノード内テンポラリファイルシステムの close 処理 (第 3.4.2.2.7 参照) と同じである。

3.4.2.4　グローバルファイルシステム

グローバルファイルシステムは第 2 階層上のファイルシステムとして提供される。計算ノードはこのファイルシステムを直接マウントするのではなく、LLIO が提供する SIO ノード上の SSD を使って実現される第 2 階層ファイルキャッシュシステムをマウントする。なお、LLIO は SIO ノード上に第 2 階層ファイルシステムをマウントしている。第 2 階層ファイルシステムは Lustre ファイルシステムが基になる。

第 2 階層ファイルシステムは、Lustre 2.10 をベースにしている。Lustre 2.10 を採用することにより京向けに富士通が拡張した機能が実現される機能を表 3.12 に示す。表 3.13 は京向けに富士通が拡張した機能を Lustre 2.10 に移植する機能を示す。

3.4.2.4.1　名前空間およびメタアクセス

第 2 階層ファイルシステムが提供する名前空間であり、全ての計算ノードおよびログインノードからアクセスできる。

同一ファイルを繰り返し open や stat を発行する場合は、SIO にマウントした Lustre 系ファイルシステムクライアントの属性キャッシュである inode やディレクトリエントリのキャッシュを活用する。属性情報が属性キャッシュに置かれている場合は、第 2 階層ストレージへメタアクセスを発行することなく属性キャッシュを用いる。

3.4.2.4.2　生存期間

ユーザが明にファイルを消去しない限りは永続的に存在する。

表 3.12: Lustre 2.10 に含まれる富士通拡張機能

機能	説明
性能・容量のスケーラビリティ	一つのファイルシステムを複数の MDS/OSS により構成し、スループットとメタアクセス性能、およびファイルシステム容量とファイルシステムで管理できるファイル数をスケールアップ
ファイルストライプ機能	ファイルデータを複数の OST にストライピングして格納し、OST 容量を超える巨大ファイル、並列アクセスに対応
ディレクトリストライピング機能	ディレクトリを複数 MDT にストライピングして格納し、単一ディレクトリのメタアクセス性能を MDT 数で向上
ユーザ/グループ Quota	ユーザやグループごとにディスク使用量および inode 使用量を制限
アクセス権管理	ユーザ/グループごとに UNIX 形式のファイルアクセス権に加えて POSIX ACL によりアクセス権限を管理
RAS 機能 (フェイルオーバ)	2 台のファイルサーバで Active/Active のフェイルオーバ構成を取り、片系故障時に他方のサーバに切り替えてクライアントからのファイル I/O およびメタアクセス処理を継続
NFS エクスポート機能	Lustre 領域を NFS エクスポート
統計情報	プロセスごとの I/O 統計情報採取

表 3.13: 富士通拡張機能

機能	説明	適用
ディレクトリ Quota	ディレクトリごとにディスク使用量、inode 使用量を制限	Lustre 改造
QoS (2.10 版の NRS 機能は使用しない)	ユーザ間の I/O リクエスト処理のフェアシェア制御、クライアント間の I/O 優先制御	Lustre 改造
RAS 機能 (ネットワーク)	1 台のクライアントやファイルサーバに複数の I/O 用ネットワークの通信ポートを持つ場合、片系故障時に縮退してクライアントからのファイル I/O およびメタアクセス処理を継続	Lustre 改造
RAS 機能 (自動 HA)	ファイルサーバの異常を検出し、フェイルオーバペアの交代先サーバに自動切り替え	外付け
アクセスハング抑止	クライアントがロックを保持したままの状態でダウンした場合に、ファイルサーバ上で当該クライアントを切り離し、他クライアントがロック獲得待ちでハングすることを抑止	外付け

表 3.14: 読み込み方式によるキャッシュおよびデータ転送先の違い

方式	SIO 上のキャッシング	データ転送先
オンデマンド	ユーザ指定	アプリケーションのデータ格納領域
自動先読み	ユーザ指定	計算ノードキャッシュ
readahead	あり	計算ノードキャッシュ

3.4.2.4.3 読み込み

第 2 階層ストレージのファイル読み込み方式にはオンデマンド、自動先読み、readahead がある。これらは SIO 上のキャッシュをどのように扱うかの違いであり、第 3.4.2.4.5 節で詳しく述べる。さらに複数プロセスで共通に読み込むファイル（共通ファイルと呼ぶ）を高速に読み出す機構を提供する共通ファイル配布コマンドを提供する。本コマンドは第 3.4.2.5 節で述べる。

3.4.2.4.4 SIO 選択とキャッシュ

SIO は第 2 階層ファイルシステムのキャッシュ処理として使用する。SIO の選択は、Temporary shared FS 同様、ファイルの inode 番号から決める。複数の SIO を使ってストライピングが可能である。ストライプカウント、ストライプサイズは実行時パラメータによってユーザが指定できる。

図 3.8 は/gfs/user/A/F1 ファイルのキャッシュ領域として、BoB#0 から BoB#3 に属する 4 つの SIO を使ってストライピングしている様子を示している。

同一ジョブ内の複数プロセスが同じファイルをアクセスしているときは同じ SIO のキャッシュ領域が使用される。異なるジョブではキャッシュ領域が異なる。

キャッシュ領域は、システムによりジョブ毎に割り当てられた SSD 領域からテンポラリ領域を差し引いた領域となる。テンポラリ領域のサイズはユーザが実行時パラメータで設定可能である。

SIO が保持する SSD を用いて第 2 階層ストレージのキャッシュを実現する。キャッシュの一貫性は NFS と同じセマンティックスであり、ユーザが陽に fsync しない限りジョブ間での同一ファイルのデータの一貫性は保たれない。キャッシュ領域に余裕がある限りファイルの close 処理後もキャッシュ領域は残す。ジョブ終了時キャッシュ領域は消滅する。

3.4.2.4.5 データ転送

計算ノードと SIO とのデータ転送は Tofu の RDMA 機能あるいはメッセージ Send/Receive 機能を使用し、CPU 負荷を減らす。

SIO と第 2 階層ストレージ (Lustre) 間は、SIO が Lustre 系クライアントとして動作し GIO 経由で Lustre ファイルサーバとデータ転送ならびにメタアクセスを行う。第 2 階層から SIO へのキャッシュfill-in、SIO から計算ノードへのデータ転送はパイプライン処理される。SIO のキャッシュの fill-in、flush は非同期に行う。

第 2 階層ストレージのファイル読み込み方式にはオンデマンド、自動先読み、readahead があり、これらの違いを表 3.14 に示す。

オンデマンド方式では、ファイル open 時に SIO 上にデータがキャッシュされていなければ SIO キャッシュをバイパスする。計算ノードは read システムコールの延長で第 2 階層ストレージからファイルデータを read システムコールで指定されたアプリケーションデータ格納領域に直接読み込む。すなわち計算ノード上ではキャッシュされない。このデータを SIO 上のキャッシュ領域に書

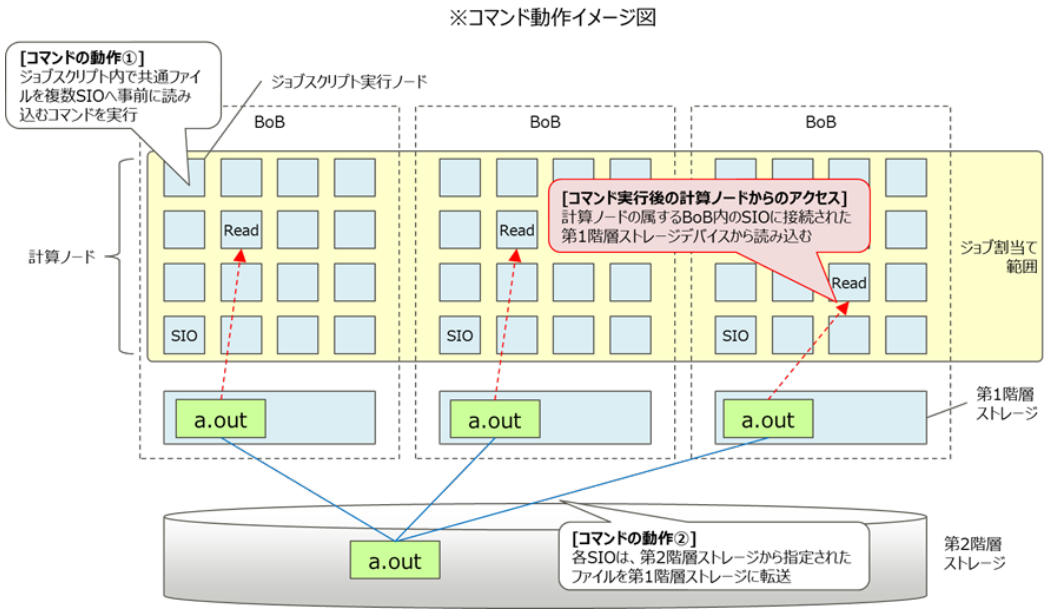


図 3.9: 共通ファイル読み込み高速化例

き込むかどうかは LLIO 設定パラメータで制御できる。

自動先読み方式では計算ノード内のキャッシュが有効となる。ファイル open 時に SIO 上にデータがキャッシュされていなければ SIO キャッシュをバイパスし、計算ノード内のキャッシュに転送される。このデータを SIO 上のキャッシュ領域に書き込むかどうかは LLIO 設定パラメータで制御できる。

アプリケーションが readahead(Linux2.4.13 以降提供) システムコールを発行して計算ノードから SIO へのファイルデータを読み込む要求は、自動先読み方式における read システムコールによるファイルデータ読み込み要求と同じである。ただし、SIO 上のキャッシュは常に有効となる。

3.4.2.4.6　計算ノード内キャッシュの扱い

第 3.4.2.3.6 節で述べている共有テンポラリファイルシステムと同様のセマンティックスである。なお、第 3.4.2.2.6 節で記載している O_DIRECT の実行セマンティックスは、グローバルファイルシステムの場合は以下第 3.4.2.4.7 節で記載している Direct I/O セマンティックスとなる。

3.4.2.4.7　Direct I/O

ファイルオープン時に O_DIRECT オプションが指定された場合には、Lustre ファイルシステムにおける O_DIRECT オプション指定時のセマンティックスに準拠する。すなわち、read 時 write 時共に、計算ノード内および SIO のキャッシュに flush されていないデータが存在する場合には flush が優先される。

3.4.2.4.8　close 処理

close 処理のセマンティックスは共有テンポラリファイルシステムの close 処理 (第 3.4.2.2.7 参照) と同じである。

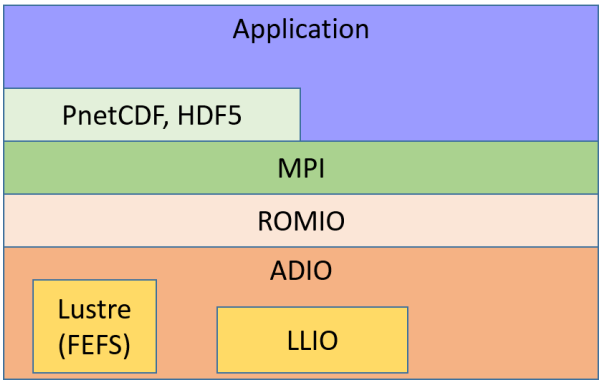


図 3.10: MPI-IO 向け LLIO

3.4.2.5 共通ファイル配布コマンド

初期設定ファイルなどプロセス群が同一ファイルを読み込む場合に、共通ファイルを各 BoB の SIO 上にファイルをコピーしファイル読み込みの負荷分散を行うことができるようにする。このために `llio_transfer` コマンドを用意する。`llio_transfer` コマンドの引数として定義されるファイル群が共通ファイル群として扱われる。

図 3.9 に `llio_transfer` コマンド実行イメージを示す。ここでは 48 ノードを使用したジョブ（3 つの BoB を使用）において a.out を `llio_transfer` コマンドで配布している。a.out は各 BoB の SIO の第 1 階層ストレージにコピーされる。各計算ノードは自身が属する BoB 上の SIO から a.out を読み込む。共通ファイルはグローバルファイルシステム上のファイルと以下の点で扱いが異なる。

- 共通ファイルは SIO の SSD にコピーされる。データの一部をキャッシュするのではない。
- 共通ファイルは SIO にまたがってストライプされない。
- 共通ファイルは第 2 階層ストレージのキャッシュ領域とは異なった管理をする。
- 共通ファイル配布コマンドの領域開放オプションにより SIO 上からファイルが削除される。陽に開放しない限りジョブが終了するまで SIO 上に残る。

3.4.2.6 MPI-IO ADIO インターフェイス

図 3.10 に示す通り、LLIO は MPI-IO の ROMIO 実装に必要な ADIO から呼び出せる LLIO インターフェイスを提供する。MPI-IO 向けには Global FS においては計算ノード内キャッシュを使わずユーザデータを直接 SIO に転送する。MPI-IO 向け LLIO インターフェイスが呼び出される前にアプリケーションが書き込み要求していた場合、計算ノード内キャッシュに未書き出しデータが存在する可能性がある。この場合、MPI-IO 向け LLIO インターフェイスは計算ノード内キャッシュの未書出しデータを SIO へ書き出す処理を優先する。

計算ノード間でファイルデータを共用するために書き込んだデータが第 1 階層ストレージまで反映されている必要があるためである。

表 3.15: プログラミング環境に関する要望と対応 (抜粋)

No	要望	対応
1	要素別処理手続の呼び出しを含むループのインライン展開	インライン展開を強化する。
2	C/C++の最適化を強化	STL 改善, 翻訳時間削減など。
3	手続き間最適化 (IPO)	リンク時最適化 (LTO) 機能を拡張 (C/C++)。module 間の最適化に対応 (Foratran)。
4	最適化および並列化の強制指示	自動並列および SIMD 化に対してサポートする。
5	最適化戦略の選択	翻訳プログラムの特性をコンパイラに伝えることで、最適化や最適化の強弱を選択し、翻訳性能および実行性能を改善する。
6	多次元の SIMD 化	サポートする。
7	単精度演算の最適化	富岳における命令セットアーキテクチャの強化に基づいて、最適化を改善する。
8	メモリ配置を指定する指示行	メモリのパディングを指示するためのコンパイラオプションまたは指示行を提供する。

3.5 プログラミング環境

3.5.1 アプローチ

富岳を効率的に利用するためには、コンパイラ、言語、ツールを含むプログラミング環境が重要である。そのため、我々は、京コンピュータにおける経験と、先行ユーザの要望に基づいて、富岳のプログラミング環境を設計・開発した。

先行ユーザ対する聞き取り調査で得られた要望 (抜粋) とそれへの対応を、表 3.15 に示す。

3.5.2 コデザイン成果

3.5.2.1 Partitioned Global Address Space Language (PGAS) 言語 XcalableMP

XcalableMP は、次世代並列プログラミング言語検討委員会および PC クラスタコンソーシアム並列プログラミング言語 XcalableMP 規格部会において検討されている分散メモリ向け並列プログラミング言語である。XMP プログラムにおいて、データ分散、ループ並列化、通信などは、簡単な指示文の形式で記述することができ、高い生産性と性能を実現できる。

我々は、R-CCS と筑波大が開発した *Omni XcalableMP* に基づいて、富岳向けの XcalableMP コンパイラを開発した。このコンパイラは、XcalableMP 1.2 仕様に準拠しており、ベース言語としては C と Fortran がサポートされている。

本章では、富岳で実施した XcalableMP プログラムの予備的な性能評価の結果を示す[1]。

評価に用いたバージョンは以下の通りである。

- Omni XcalableMP Version: 1.3.2, Git Hash:6d23f46
- Language specification: 1.2.25

富岳における XcalableMP の性能は、メニーコアプロセッサと新しい Tofu-D インターコネクトによって向上される。

[1] この結果は、富岳の試行期間における評価環境で得られたものである。共用開始以降の性能を保証するものではないことに注意されたい。

3.5.2.2 グローバルビュープログラミングの評価

3 次元流体コード IMPACT-3D をターゲットとして、富岳の 512 ノードを用いて、XcalableMP のグローバルビュープログラミング機能を評価した。その結果および京コンピュータとの比較を、図 3.11 に示す。このプログラムは、XMP と OpenMP によるハイブリッド並列により記述されている。すなわち、1 個の XMP ノードが 1 個の計算ノードに割り当てられ、48 個の OpenMP スレッドが 1 個のノード内で動作する。問題サイズは $512 \times 512 \times 512$ であり、3 次元のブロック分散を用いている。また、コンパイルオプションとして “-Kfast” を指定した。

図から、富岳において良好なスケーラビリティが得られていることがわかる。また、XMP ランタイムのステンシル通信の最適化 [25] の効果により、MPI 版より高い性能が得られている。

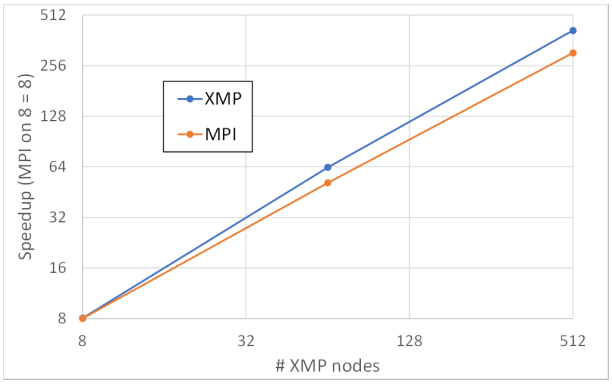


図 3.11: 富岳における Impact3D の性能向上と京コンピュータとの比較

3.5.2.3 ローカルビュープログラミングの評価

富岳が備える Tofu-D インターコネクトは、ハードウェアによって支援される RDMA (Remote Direct Memory Access) 操作をサポートしている。我々は、Tofu-D を利用して、XMP のローカルビュープログラミングのための片側通信機能を実装した。具体的には、富士通が提供する低レベル通信レイヤ uTofu API [21] を用いた。

ローカルビュープログラミングの評価のため、Fibber Miniapp Suite [4] [24] に含まれる CCS QCD と NTChem-MINI を用いた。

CCS QCD [2] の評価では、8 個の XMP ノードを 1 個の計算ノードに配置し、フラット並列化を行った。評価に用いた問題サイズと諸条件は以下の通りである。

- ターゲットデータ: Class 2 (32 x 32 x 32 x 32) (ストロングスケーリング)
- コンパイラオプション: -Kfast,zfill,simd=2
- 計測区間: 組み込み時間計測機能における “Clover + Clover_inv Performance” と “BiCGStab (CPU:double precision) Performance” の合計

図 3.12 に、富岳における性能向上と京コンピュータとの比較を示す。XMP 版は MPI 版とほぼ同等の性能を達成している。京コンピュータにおける XMP 版の性能低下は、通信バッファとして

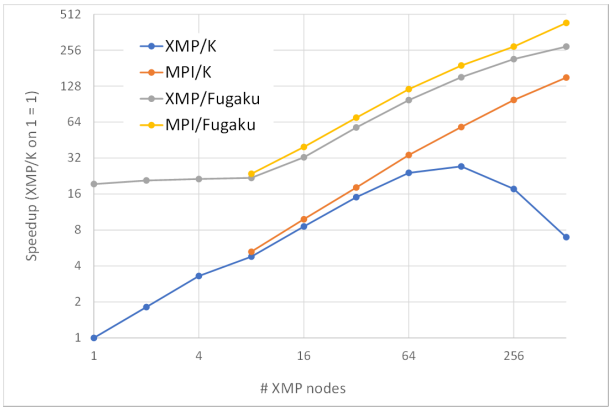


図 3.12: 富岳における CCS QCD の性能向上と京コンピュータとの比較

用いられる割付け coarray の割付け処理のオーバヘッドによるものである。富岳では、uTofu を用いることでこの問題は解消された。

NTChem-MINI は、大規模かつ複雑な分子系に対する第一原理量子化学計算のための高性能ソフトウェアパッケージ NTChem [8] から派生したミニアプリである。NTChem-MINI の評価では、1 個の XMP ノードを 1 個の計算ノードに割り当て、ノード内では 48 コアにより BLAS 関数が実行される。評価に用いた問題サイズと諸条件は以下の通りである。

- ターゲットデータ: taxol (ストロングスケーリング)
- コンパイラオプション: -Kfast,simd=2
- 計測区間: 組み込みの時間計測機能における “RIMP2_Driver”

図 3.13 に示すように、XMP 版は、元の MPI 版と同等の性能を達成している。

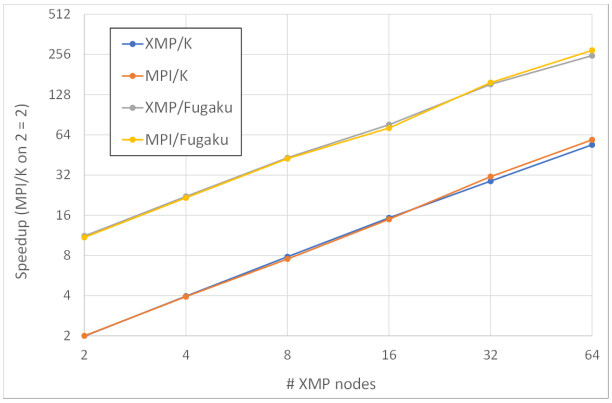


図 3.13: 富岳における NTChem-MINI の性能向上と京コンピュータとの比較

3.5.2.4 グローバルなタスク並列プログラミング

近年、Intel Xeon Phi のようなメニーコアプロセッサから構成される大規模クラスタが、多くのサイトに導入されている。メニーコアプロセッサのためのプログラムを作成するための共有メ

モリプログラミングモデルとしては、OpenMP が広く利用されている。ほとんどの OpenMP プログラムは、ループに対するワークシェアリングを使って記述されており、それはグローバルな同期を伴う。しかし、特に最近のメニーコアプロセッサでは、ワークシェアリングのためのグローバルな同期のコストは大きくなっており、コア数の増加とともにコア間の負荷不均衡も性能低下の原因となる。OpenMP 4.0 で導入されたタスク依存に基づくタスク並列プログラミングを用いると、ユーザが明示したデータ依存に基づくタスク間の細粒度の同期によりグローバルな同期を避けることができるため、そのようなメニーコアプロセッサにおける並列化を促進する手段として有望視されている。

我々は、タスク並列プログラミングモデルを拡張し、分散メモリシステム向けの XcalableMP の PGAS モデルに適応させることを検討している。これにより、コストの大きいグローバルな同期を削除できるだけでなく、通信と計算のオーバラップを実現することも期待できる。そのため、XMP 2.0 仕様において、グローバルタスク並列プログラミングを提案している。

本提案の詳細は、文献 [32] で述べられている。

3.5.2.4.1 OpenMP Task 指示文と XMP Tasklet 指示文

元々の OpenMP は、`parallel for` 指示文によるループのワークシェアリングに重点を置いていたが、OpenMP 3.0 では `task` 指示文によるタスク並列が導入された。この機能により、再帰的構造や不定回ループのように、動的かつ不規則に生成されるワークを並列に実行することが可能になる。さらに、OpenMP 4.0 で導入された、`task` 指示文の `depend` 節により、`in`、`out` および `inout` の種別を持つデータ依存関係を指定することが可能である。タスク依存を利用することで、ユーザが指定するデータ依存に従ってタスク間の細粒度の同期を行うことが可能になるため、スレッドチームのグローバルな同期を削減することができる。

XMP においてタスク並列をサポートするため、XMP2.0 において `tasklet` 指示文[2]が提案された。図 3.14 は、XMP におけるマルチタスク実行のための `tasklet`、`tasklets` および `taskletwait` の各指示文の文法を示している。`tasklet` 指示文は、`on` 節で指定されたノードに、直後のブロックに対応するタスクを生成する。生成されたタスクはスケジュールされ、タスク依存がない限り、当該ノードの任意のスレッドによってただちに実行される。タスク依存がある場合、当該タスクの実行は、全ての依存関係が満たされるまで遅延される。以上の挙動は、タスクが `tasklets` 指示文に囲まれている場合に起きる。タスクが `tasklets` 指示文に囲まれていない場合、指定されたノードにおいてタスクは逐次的に実行される。`tasklet` 指示文は、タスク依存を記述するいくつかの節を持つ。`in` 節、`out` 節および `inout` 節は、ノード内のタスク依存を表す。`in`、`out` または `inout` 節が `tasklet` 指示文に現れるとき、生成されたタスクは、それぞれ対応するデータ依存をノード内に持つ。これらのデータ依存の挙動は、OpenMP の `task depend` のそれと同じである。

`taskletwait` 指示文は、各ノードにおいて生成されたタスクの完了を待つ。この指示文はバリア同期の効果を含まないため、XMP の `barrier` 指示文により、全ノードの全タスクがこの時点で完了することを保証する必要がある。`tasklets` 指示文の最後には、各ノードにおける暗黙的なバリアが行われる。

OpenMP では、1 ノード内の逐次実行に基づくデータの読み書きの順序にしたがって、タスク依存関係が生成される。一方で、マルチノードにおけるタスク並列を考えると、各ノード内のスレッドは並列に実行されるため、OpenMP のタスク並列モデルを、異なるノードにおいて実行されるタスクの間の依存にそのまま適用することはできない。

OmpSs [19] では、ノード間の相互作用は、MPI 通信を実行する「MPI タスク」として記述す

[2]XMP には、OpenMP のそれとは異なる意味の `task` 指示文が存在する。

#pragma xmp tasklet *[clause[, clause] ...] [on { node-ref | template-ref }]*
(structured-block)

#pragma xmp taskletwait *[on { node-ref | template-ref }]*

#pragma xmp tasklets
(structured-block)

where *clause* is :
{in | out | inout} (*variable[, variable] ...]*)

図 3.14: tasklet, taskletwait, tasklets 指示文の文法

る。この場合、ノード間のタスク依存は、MPI の一対一通信の完了によって保証される。このアプローチによると、ノード間の依存関係は満たされるが、ユーザは、それぞれ異なる記法を持つ 2 つのプログラミングモデルを組み合わせて使うことになるため、さらなる生産性の低下を招く。それに対し、我々が提案する XMP の新しいタスク通信指示文では、言語の構文だけを使ってクラスタ環境におけるマルチタスキング実行を記述することが可能である。

3.5.2.4.2 グローバルなタスク並列プログラミングの提案

分散メモリ並列システムにおいてマルチタスク実行を実現するためには、ローカルなタスク依存グラフ内のタスクの間で一対一の通信を行う必要がある。XMP はいくつかの通信指示文を提供するが、それらの多くは集団的に実行されるため、実行ノード間の暗黙的な同期を伴う。そのような通信に参加するタスクは全タスクが完了するまで同期を待つため、性能低下を引き起こす。そこで、図 3.15 に示す、ノード間のデータ依存をための tasklet gmove と tasklet reflect の 2 つの指示文を提案した。これらは、一対一通信により、タスクに含まれるノード間のインタラクションを指定する。これらの通信は、各タスク内に含まれる送信者と受信者の間の一対一の同期のみを伴う。

これらの指示文の詳細は以下の通りである。

tasklet gmove 指示文 この指示文は、gmove 指示文と同様に、対応する代入文の右辺から左辺へ変数 (ローカルまたは分散) をコピーするが、タスクとして実行される。このコピー処理は、原則として全ての実行ノードで行われる。ただし、分散配列が指定された場合、それを保持しているノードのみが当該タスクにおいてコピー処理を実行する。また、on 節によって実行ノードを指定することもできる。in、out または inout 節が指定されているとき、tasklet 指示文と同様に、生成されたタスクは、ノード内で対応するタスク依存を持つ。

tasklet reflect 指示文 この指示文は、reflect 指示文と同様に、*array-name* で指定された配列のハロー領域を更新するが、タスクとして実行される。例えば、2 ノード上に分散された一次元配列の片側のハロー領域を更新するとき、その通信は 4 つのタスクに分割される。すなわち、ノード 1 の上側ハロー領域の送信、ノード 1 の上側ハロー領域の受信、ノード 2 の下側ハロー領域の送信、ノード 2 の下側ハロー領域の受信の 4 つである。この指示文では、生成されたタスクに対し、通信データに基づいてデータ依存が自動的に付加される。なぜなら、分散配列の境界のインデックスは、XMP ランタイムシステムによって動的に決定されるためである。chunksize 節を用いて、tasklet reflect 指示文により生成される依存を、ユーザが記述するタスク依存に一致させることができる。ある配列に対する計算がブロック

```
#pragma xmp tasklet gmove [clause[, clause] ... ] [on { node-ref | template-ref } ]
  (an assignment statement)

#pragma xmp tasklet reflect (array-name[, array-name] ... )
                    [blocksize (reflect-blocksize[, reflect-blocksize] ... ) ]

  where clause is :
    {in | out | inout} (variable[, variable] ... ])
```

図 3.15: tasklet gmove と tasklet reflect 指示文の文法

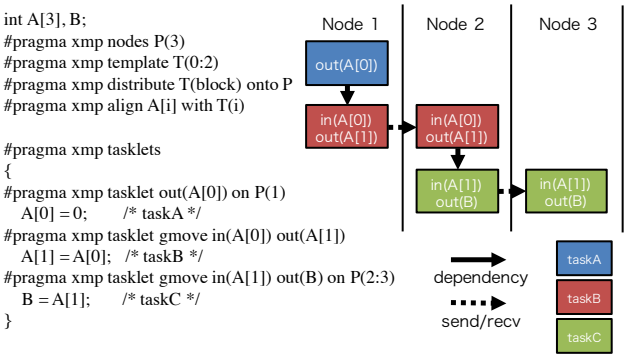


図 3.16: tasklet と tasklet gmove 指示文の例

化されているとき (ノード内のデータ依存を伴うキャッシュブロッキング技法を用いる場合など)、ユーザが指定したタスク依存と、ハロー交換に対して生成されるデータ依存が一致しない可能性がある。chunksize 節を指定することによって、ハロー領域は、等幅・連続のチャンクに論理的に分割され、指定されたチャンクサイズに基づいて、ハロー交換に伴うデータ依存を XMP ランタイムシステムが自動的に生成する。

図 3.16 は、tasklet gmove 指示文の例を示す。ここでは、大きさ 3 の配列 *A[]* が、等幅・連続のブロック形式で 3 つのノードに分散されている。このコードでは、3 種のタスクが生成される。TaskA と TaskC は、on 節で指定されたノードにおいて実行される。TaskB は、当該 tasklet gmove 指示文に対応する代入文で参照される *A[0]* と *A[1]* をそれぞれ保持するノード 1 とノード 2 において実行される。ノード 1 上の TaskA と TaskB の間には、*A[0]* に関するフロー依存が存在する。TaskA の実行の後、TaskB は *A[0]* を、分散配列 *A[1]* を保持するノード 2 に送る。ノード 2 において、TaskB はノード 1 から *A[0]* を *A[1]* に受け取る。TaskB の受信処理が完了次第に、*A[1]* に関するフロー依存が満たされ、TaskC の実行が開始される。TaskC は、ノード 2 の変数 *B* へ *A[1]* を送る。変数 *B* は各ノード上に存在するローカル変数であるため、その通信相手は、on 節で指定される実行ノードで決まる。

3.5.2.4.3 コード変換のプロトタイプ設計

提案する指示文を含むコードから、MPI と OpenMP に基づく XMP ランタイムの呼び出しを含むコードへの変換の方法を設計した。予備評価のため、この方法に従って変換された MPI+OpenMP のコードを手で作成した。

tasklets 指示文は、OpenMP の parallel 指示文と single 指示文に変換される。実行ノードを決める on 節は if 文に変換される。tasklet gmove 指示文と tasklet reflect 指示文は

表 3.16: サポートするプログラミング言語の規格/仕様

言語	規格/仕様
Fortran	ISO/IEC 1539-1:2018 (Fortran 2018)
	ISO/IEC 1539-1:2010 (Fortran 2008)
	ISO/IEC 1539-1:2004, JIS X 3001-1:2009 (Fortran 2003)
	ISO/IEC 1539-1:1997, JIS X 3001-1:1998 (Fortrn 95)
	Fortran 90 and FORTRAN77
	OpenMP Application Program Interface Version 4.0 July 2013
	OpenMP Application Program Interface Version 4.5 November 2015 (partially)
	OpenMP Application Program Interface Version 5.0 November 2018 (partially)
C	ISO/IEC 9899:2011 (C11)
	ISO/IEC 9899:1999 (C99)
	ISO/IEC 9899:1990 (C89)
	GNU extensions
	Clang extensions
	OpenMP Application Program Interface Version 4.0 July 2013
	OpenMP Application Program Interface Version 4.5 November 2015 (partially)
	OpenMP Application Program Interface Version 5.0 November 2018 (partially)
C++	ISO/IEC 14882:2014 (C++14)
	ISO/IEC 14882:2011 (C++11)
	ISO/IEC 14882:2017 (C++17)
	GNU extensions
	Clang extensions
	OpenMP Application Program Interface Version 4.0 July 2013
	OpenMP Application Program Interface Version 4.5 November 2015 (partially)
	OpenMP Application Program Interface Version 5.0 November 2018 (partially)

MPI_Send/Recv() に変換され、これらの MPI 関数は、ユーザが指定したデータ依存関係を持つ OpenMP タスクによって実行される。これらのコードにおいて、MPI_Send/Recv() のような MPI のブロッキング呼び出しが起きる場合、タスクスケジューリングの実装次第でデッドロックが起きる可能性がある。これを避けるため、実際には、MPI の非同期通信 (MPI_Isend/Irecv(), MPI_Test()) と OpenMP の `taskyield` 指示文を用いた。これにより、カレントタスクは、別のタスクが生成された時点で休止状態になり、異なるタスク切り替わることが可能になる。

3.5.2.5 コンパイラおよびプログラミング言語

3.5.2.5.1 コンパイラ

開発したコンパイラは表 3.16 に挙げるプログラミング言語をサポートし、富岳向けの最適化機能を有する。

3.5.2.6 半精度浮動小数点型 (FP16) のサポート

Fortran、C および C++において、IEEE754-2008 規格に準拠した binary16 フォーマットに対応する半精度浮動小数点型 (FP16) をサポートする。

各プログラミング言語においてサポートされる半精度浮動小数点型の種別は以下の通りである。

- Fortran
  - `REAL(KIND=2)`
  - `COMPLEX(KIND=2)`
- C/C++
  - `_Float16`
  - `__fp16`

3.5.2.7 その他のプログラミング言語

Java、Ruby および Python を利用できる。特に、科学技術計算の分野で利用が広まっている Python については、数値計算ライブラリを利用した高速化方法も提示する。

3.5.3 FDPS

FDPS (Framework for Developing Particle Simulators) [27, 28] は、領域分割、粒子移動、異なる計算領域での粒子間の相互作用を評価する方法などの必要な並列化アルゴリズムを実装せずに、アプリケーションプログラマが独自の高性能並列化粒子ベースシミュレーションコードを開発できるようにするためのフレームワークである。 FDPS を富岳で利用できるようにし、また富岳に特化した最適化を多数行った。また、粒子間相互作用カーネルの高水準記述から、高度に最適化された粒子間相互作用カーネルを生成するコードジェネレータ PIKG を追加した。PIKG は、AVX2、AVX512、Cuda、A64fx SVE に最適化したコードを生成することができる。A64fx 向けの最適化には、SIMD intrinsics の使用、ループアンローリング、ループ分割が含まれる。

3.5.4 ツール

3.5.4.1 GUI 統合開発環境

富岳における GUI 統合開発環境として、Eclipse [1] およびそのプラグインである PTP を提供する。PTP は、プログラム編集、システム監視、ジョブ制御などの機能を備える。特に、システム監視およびジョブ制御の機能に関し、富岳に対応するための改良を加えた。

3.5.4.2 デバッグ機能

富岳では、以下に挙げる 3 種のデバッグ機能を利用できる。

- デッドロック調査

  デッドロックが発生した箇所を特定し、バックトレース、メモリマップおよびスタック上の変数に関する情報を表示する。

- 異常終了調査

  プログラムがシグナルを受信した場合に、バックトレース、当該手続きの逆アセンブルおよびメモリマップに関する情報を表示する。

- コマンドファイルによるデバッグ

  全プロセスまたは特定のプロセスにおいて、コマンドファイルで指定されたデバッグ操作を実行する。

3.5.4.3 性能プロファイラ

富岳では、「基本プロファイラ」と「詳細プロファイラ」の 2 種の性能プロファイラを利用できる。

- 基本プロファイラ

  基本プロファイラは、サンプリング解析によりプログラム全体の統計情報を計測・表示する。この統計情報には以下が含まれる。

  - **時間統計情報**
    プログラムの経過時間、ユーザー CPU 時間およびシステム CPU 時間
  - **CPU 動作状況**
    メモリースループットや命令数、演算数など
  - **コスト情報**
    プログラム実行中にサンプリングした回数に基づく、手続、ループおよび行単位のコスト
  - **コールグラフ情報**
    手続の呼出し経路および手続の呼出し経路ごとのコスト
  - **ソースコード情報**
    ソースコードの各行にふかされたコスト

- 詳細プロファイラ

  詳細プロファイラは、アプリケーションの指定した区間の実行性能情報の計測・出力する。この性能情報には以下がが含まれる。

  - **時間統計情報**
    計測対象区間の呼出し回数、経過時間、ユーザー CPU 時間およびシステム CPU 時間の内訳など

- **MPI 通信コスト情報**
  計測対象区間の MPI 関数の実行回数、メッセージ長、実行時間および待ち時間の平均値、最大値および最小値
- **CPU 性能解析情報**
  計測対象区間のプログラム実行時の CPU 動作状況

3.6 数値計算ライブラリ

3.6.1 アプローチ

3.6.1.1 開発・設計方針

科学技術を対象とした大規模計算では、様々な物理問題の解決に対して標準的な問題形式や類似した計算手法が用いられることが多い。新しいスーパーコンピュータの性能を十分に引き出すためには、このような汎用的な処理のための数値計算関数を科学技術計算用のライブラリとして提供する必要がある。BLAS に代表される基本的な数値計算関数も、このような科学技術計算用のライブラリから内部的に利用される基本コンポーネントとして重要である。アプリケーションプログラムから直接利用される機会が多いため、ハードウェアアーキテクチャに合わせた実装により、様々な呼び出し条件で高性能な処理を実現することが重要となる。

富岳上の数値計算ライブラリスタックは、富岳の性能を十分に発揮できる汎用的な科学技術計算のために設計される必要がある。我々は開発方針として以下の 3 点を掲げた。

1. 京で提供されていた科学・数学ライブラリを富岳にも継承し、十分な性能を発揮できるようにする。特に、BLAS、LAPACK、ScaLAPACK、FFT 関連など重要どの高い関数は、アルゴリズムの詳細設計から実装まで、新アーキテクチャに合わせて開発する。

2. 富岳の開発では、アプリケーション開発者との共同設計を重視する。数値解析の観点から機能を強化するだけでなく、実際のアプリケーションプログラムからシステムを利用する際の問題点を検討する。

3. 応用数学・計算機科学の近代化に基づいて、新しい機能、重要性が増すアルゴリズム、実装方法を検討し、積極的に開発する。また、ライブラリのチューニングで得た知見を共有し、機能の移植や追加開発を理研や富士通と協力して行う。

以上の方針のもと、富士通と理研が協力して富岳の数値環境を構築した。具体的には、i) 京と同様の数値計算ソフト環境を継承する富士通ソフトスタック、ii) 従来のソフトの性能ボトルネックを考慮し、同じく理研提供の数値計算ソフトを継承する理研ソフトスタック、iii) アプリケーションユーザー間の緊密かつ強固な相互作用に基づくコデザインの取り組み、iv) 並列プログラミングの必然でもある OSS スタック、の 4 つのサブスタックで数値計算ソフトウェアスタックの設計開発を進めた。

3.6.1.2 利用者からのニーズ分析

理研が開発している重点課題アプリケーションからの要求に対して高優先度をもって整備する必要がある。必要とされる OSS や既存関数の機能拡張項目の選定は、アプリケーションからの要求

の重要性と、計算性能や精度にある程度の配慮が必要な大規模計算への対応を考慮して行う。個々のルーチンの優先度にしたがって、1）単純な移植のみ、2）コア部分の性能向上のためのソースコード修正のチューニング、3）単純なチューニングでは不十分な部分のアルゴリズム変更、という分類を行い、優先度に応じて性能向上作業を決定した。

数値計算ライブラリの設計段階では，主要なアプリケーションユーザや優先的に開発されるアプリケーションコードの開発者にインタビューを行った。さらに、京の一般ユーザーにもアンケートを実施し初期需要を把握した。最終的には、富岳の数値計算ライブラリを開発・充実させていく上で、既存ライブラリの移植に加えて以下の機能強化並びにチューニングの優先度を上げ、富岳向け数値計算ライブラリ提供することとした。

1. NICAM-LETKF が必要とするシングルノードでの高速な固有値ソルバーを重点的に強化する。
2. ADVENTURE では、より高精度な行列-行列間の乗算の必要性が高く、必要な機能拡張やチューニングを行う。
3. RSDFT では、分散システムにおける基底直交化アルゴリズムが重要であり、手法選定ならびにチューニングを行う。

コデザインでのインタビューやアンケートを通じて、チューニングが必要な OSS や、SSL II や RIKEN の開発ではカバーできない機能を把握した。ユーザーに最も頻繁に利用されている、あるいは京に実装されている数値計算ライブラリ（OSS および商用ソフトウェア）は以下のようになる。

- 第一回目の調査で挙げられた OSS:
  PLASMA、DPLASMA、ELPA、Elemental、EigenExa、PETSc、Trilinos、ppOpen-HPC, Lis, MUMPS, SuperLU, UMFPACK, METIS, (P)ARPACK, z-Pares, BLOPEX, JADAMILU, SLEPc, FFTW, FFTE, FFTSS, OpenFFT, FFTS, FFTPACK, SFMT, exib, GNU MP, QD, FMLIB, MPACK, GSL
- インタビューで提示された OSS:
  EigenExa, ELPA, RANLUX (PRNG), NL2Sol (非線形最小二乗法), LibInt (二体分子積分評価用関数) LibInt（2 体分子積分の評価関数)、PFAPACK（実数または複素数の斜対称行列のパフィアンを数値計算するライブラリ)、Boost（C++ライブラリ）など
- K に移植された OSS:
  SuperLU, SuperLU DIST, METIS, ParMETIS, gmp, mpc, mpfr, UMFPACK, ARPACK, PETSc, SLEPc
- 日本の HPC プロジェクトで開発された OSS:
  z-Pares, EigenExa, ppOpen-HPC, FFTE

実際、上記の OSS の多くは Spack フレームワークでサポートされ、移植可能な状況であるが、特に性能が重視される FFTW の SVE ベクトル化を富士通が実施することとした。

3.6.2 コデザイン成果

富士通のソフトウェアスタックは、京で実績のある Netlib ベースのライブラリと SSL II ベースのライブラリを提供する。理研のソフトウェアスタックは、富士通ソフトウェアを補完する形でより高度で高機能な数値計算ライブラリを含んでいる。数値計算ソフトウェアスタックは、富士通の A64FX 用にチューニングされた BLAS カーネルを利用し、高度に並列化、スレッド化された環境において、様々なオーバーヘッドを削減することで、富岳上での高性能な数値計算フレームワークをユーザーに提供する

3.6.2.1 富士通ソフトウェア 1: Netlib

BLAS、LAPACK、ScaLAPACK など、いわゆる netlib で配布されている業界標準の数値計算ソフトウェアについては、ハードウェア性能やユーザーの要望を反映して、各機能のチューニングを行った。特に、シングルコア用に設計された BLAS カーネルについては、プロセッサの演算性能を引き出すために、SVE 命令を用いたアセンブラレベルのプログラミングとアセンブラチューニングを行い、DGEMM については必須目標性能の 87.4%の約 3TFlops を達成している。また、多くのアプリケーションで要求の高い DGEMM については、正方行列の条件だけでなく、小型化や縦長形状などの特殊なケースにも対応できるようにチューニングを行った。また、ARMv8 でサポートされている IEEE754 の半精度（いわゆる FP16）に対応した BLAS ライブラリの API とカーネルコードを決定し、それぞれ実装した。さらに、

- 行列のコピーと転置のルーチン
- シングルスレッド専用 API
- SIMD 長単位の実部虚部分離配列対応の行列行列積機能

など、利用者の要求にこたえる形で機能拡張、追加実装を行った。

図 3.17 は、BLAS 行列積ルーチンである GEMM カーネルの性能を示している。BLAS 行列積ルーチンは、倍精度、単精度、半精度演算器の物理的な SVE ユニットの長さである 8、16、32 ワードに等しく比例する理論上のピークに応じて最適な性能を示すように A64FX プロセッサで実行される。完成時のベンチマーク測定で、DGEMM, SGEMM, HGEMM では、CPU 周波数 2.0GHz のシングル CMG で、それぞれ約 710Gflops, 1.5Tflops, 2.7Tflops の性能を達成した。レベル 2 BLAS の代表的なルーチンである DGEMV の十分大きなサイズでの性能は、CPU 周波数 2.0GHz、12 スレッド（1CMG/1 ノード）実行で 50.15Gflops であった。これは、理論ピーク性能の 6.52%に相当する。

3.6.2.2 富士通ソフトウェア 2: SSL II

SSL II は、数値解析の 10 分野をカバーする富士通製高性能ライブラリである (全体で約 300 の単精度および倍精度のルーチンをサポートする)。netlib の開発と同様に、SIMD 命令生成ビルトイン機能を用いて、A64FX CPU アーキテクチャへの性能チューニングを行う。富岳の処理形態に対応して、スレッド並列版、C/Fortran 版、MPI 分散並列版と複数のバージョンを用意している。

利用者のヒアリングからも、FFT 機能は，科学技術の分野で重要なサブルーチンとして認識されており、十分な性能チューニングが必要である。まず、シングルコア FFT カーネルを対象配列

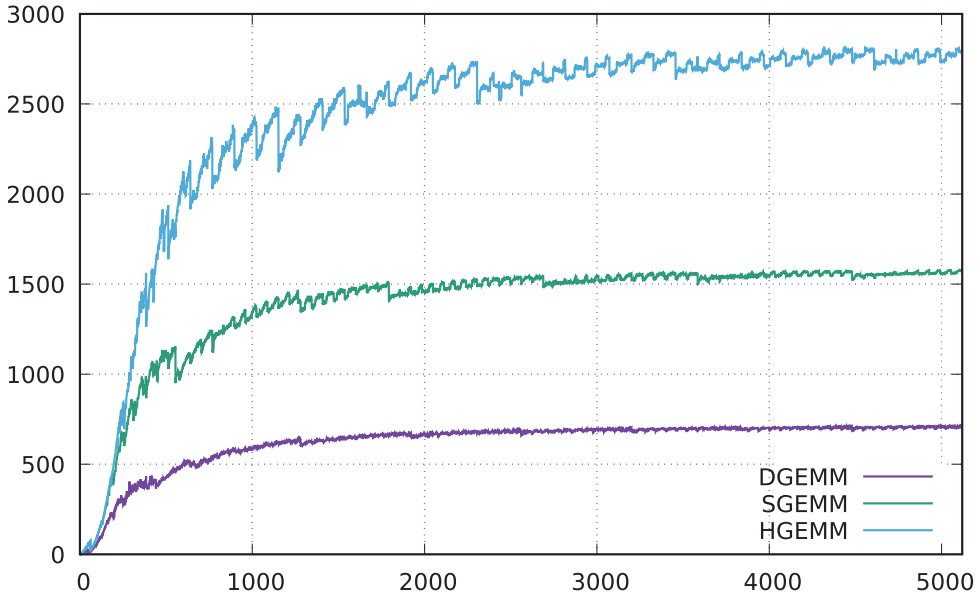


図 3.17: 2.0GHz モードのシングル CMG における [DSH]GEMM 性能（Gflop/s)

を L1 キャッシュに配置できるという条件で性能チューニングする。それを活用し関連ルーチンを構成する方式を採用した。また、逐次版やスレッド並列版を呼び出すために京で開発された既存の API を継承するとともにし、アプリケーション開発者の要望に応えて 2 軸分割された 3 次元 MPI 並列関数を開発した。図 3.18 は A64FX のシングルコア上での FFT カーネルの性能ベンチマークの結果でである。SSL II の 1d-FFT と OSS の代表ライブラリである FFTW3 のナイーブなコンパイル版と SVE 最適化版の性能も同時に示している。

3.6.2.3 理研ソフトウェア 1: EigenExa

コ・デザインの結果、固有値・特異値分解ルーチンも頻繁に使用される重要な数値解析ツールであることが明らかになった。これらのルーチンは、標準的な数値計算ライブラリである LAPACK や ScaLAPACK に含まれているが、高度な並列システムにおけるスレッド並列、分散並列、ハイブリッド並列のための（現実的な面での）高速な実装はない。特異値計算は固有値計算から派生したものであるため、理化学研究所が開発した EigenExa の知見を活用し、最終的には固有値計算ルーチンと特異値計算ルーチンの両方を同時に高並列化した。

本開発では、固有値計算アルゴリズムの中でも特に並列化効率の伸びが鈍化しがちな分割統治法に対して、以下のような方針での新規実装を行った。

1. マージ操作時の固有ベクトルのデータ分配法 ELPA のデータ分割方式を一般化し、行列積の実行を一方通行リング通信で並列に行う。
2. プロセスグリッドの分割順序は、縦軸、横軸ともに固定せず、より柔軟に、実行時のプロセスグリッド情報に合わせて分割方法を変更することができる。
3. マージプロセスの再帰的構造を反映した簡潔で保守しやすいコードを生成する。

固有値ソルバー開発におけるその他の技術的課題は、プロセッサの性能向上に対して、ネットワークの性能が京から相対的に低下していることであり、これが最終的にアンバランスにつながっ

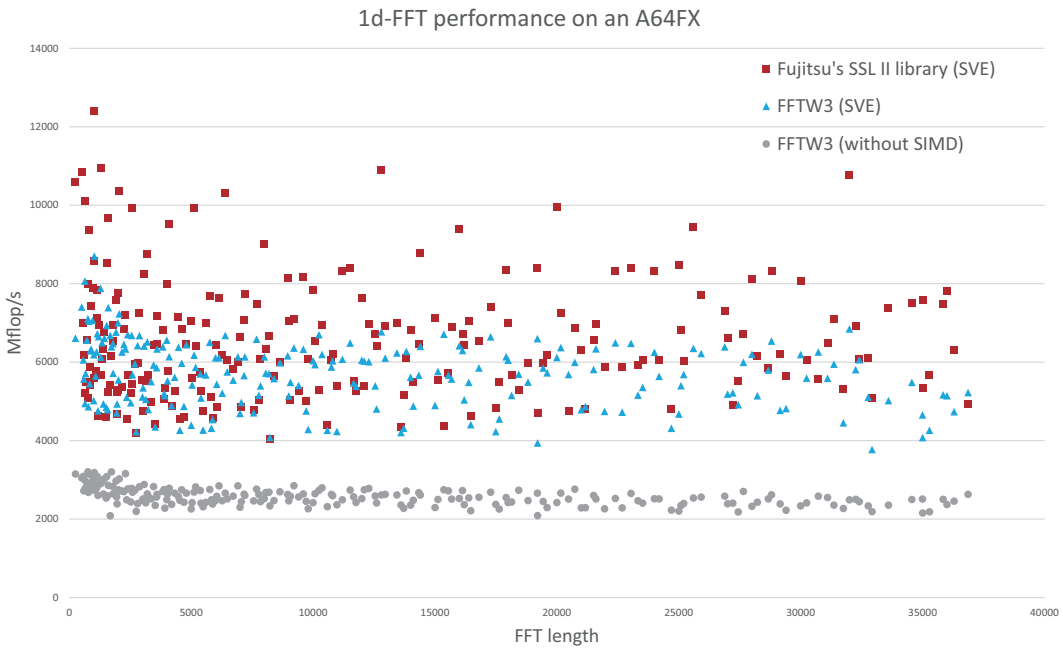


図 3.18: A64FX のシングルコアにおける 1d-FFT カーネル性能ベンチマーク結果

ている。 実際、近年の小規模なスパコンでも、高度に並列化されたスパコンでは通信が最大のボトルネックとなっており、富岳の規模ではこの状況を無視できないことは、様々な性能評価の結果から裏付けられている。結果として以下の 2 つの主要テーマに集約された。

1. ハウスホルダー三重対角化における富岳の通信回避技術を実現する。

2. 分割統治法の負荷バランスの改善に配慮した新しいプロセスマッピング法を採用する。

これらの技術により、京の中規模以上の並列環境で、大幅な性能向上が確認されている。また、これらは富岳上でも実用的な規模の問題で大幅な性能向上が期待できる。EigenExa の開発初期段階から、MPI、OpenMP、高性能 BLAS などの様々な並列プログラミング言語・ライブラリや、一般的な SIMD ベクトル化技術、Fortran90 コンパイラ技術を用いて開発されてきた。これらの技術的優位性に加えて、A64FX プロセッサで強化された SVE による長ベクトル技術や、複数 CMG で構成される NUMA 環境への対応は、富岳の新たな技術課題と認識された。

図 3.19 は、N=65,536, 131,072, 262,144 などの 3 ケースで、富岳上の 128〜16,384 ノードを用いて、EigenExa2.6c と ScaLAPACK に含まれる PDSYEVD ルーチンの総計算時間を示している (ただし、PDSYEVD は、高性能が期待できないため、N=262,144 では測定していない)。大規模な固有問題を解く際には，良好なスケーラビリティが見られる。例えば，最も重要なケースである 262,144 次元の行列を解く場合，128 ノードから 4,096 ノードまでは良好な並列効率とスケーラビリティが見られ，徐々にスケールダウンしているが、並列性能は十分に向上している。一方、PDSYEVD は EigenExa2.6c の 2〜4 倍程度の計算時間を要し、かつ高並列時の性能飽和が顕著である。

図 3.20 は、EigenExa2.6c の 3 つの主要ルーチンの時間内訳を示している。本プロジェクトで開発した分割統治法は、中小規模の領域では絶対的な性能が向上しているが、高並列環境では当初想定していなかったほど性能が向上していないことがわかる。更なる分析が必要だが、プロセス

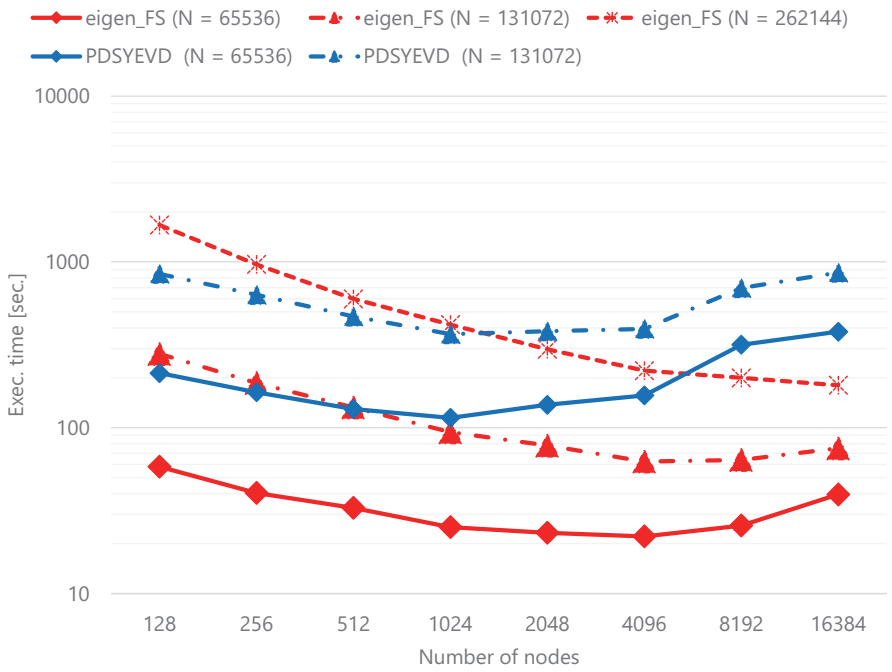


図 3.19: 富岳における EigenExa2.6c の総計算時間 (経過時間の単位: 秒)

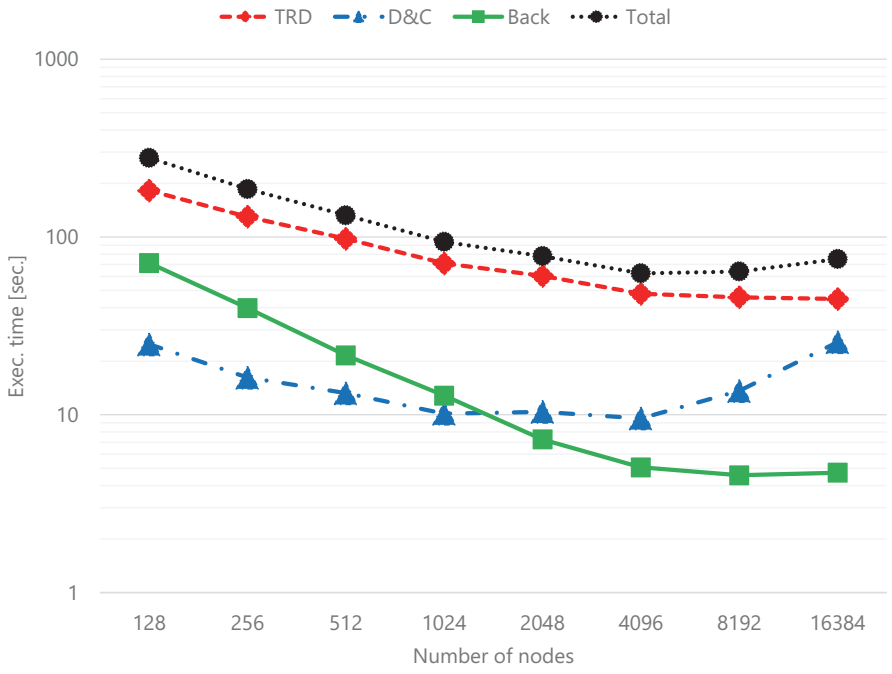


図 3.20: N= 131,072 の場合の 3 つのカーネルルーチンの時間内訳

マップと Tofu トポロジーの間の二分木分割のマッピングにおいて、高次プロセスの場合にサブプロセスグループ間の衝突や重複が発生し、それが性能低下の原因となっている可能性があり、改善を計画している。

3.6.2.4 理研ソフトウェア 2: BatchedBLAS

富岳は A64FX のメニーコア・プロセッサを使用している。数値線形代数演算をスレッドレベルで並列化するためには、シングルコアにシングルジョブを割り当てるのではなく、各物理コアに小さなサイズの BLAS タスクやその内部のサブタスクを大量に割り当て、コアのリソースをすべて使用するような戦略的な方法を採用する必要がある。Batched BLAS は、このようなメカニズムに基づいて、数値線形代数演算の新しいスケジューリング形態を提供する。実験的な Batched BLAS は、その一部が既に AI などに利用されており、実用化が待ち望まれていた。Batched BLAS は富士通の数値計算ライブラリ（SSL II）に収録されていなかったため、理研の開発チームは、まだ標準化されていない仕様にも容易に適用できる設計方針に基づいて、2017 年に Batched BLAS のフルバージョンを実装した。2018 年 7 月にリリースされた正式規格への対応も行い 2021 年 2 月にパッケージを公開している。さらに、FP16 の強化にも一部対応している。

図 3.21 は、numactl コマンドで計算プロセスを A64FX の特定 CMG にバインドした状態で測定した性能である。GEMM では, 理論ピーク (DGEMM で 750GFlops/CMG) に換算して 50~80% を達成しており、適切なスケジューリングにより 1CMG の能力を使い切ることに成功している。

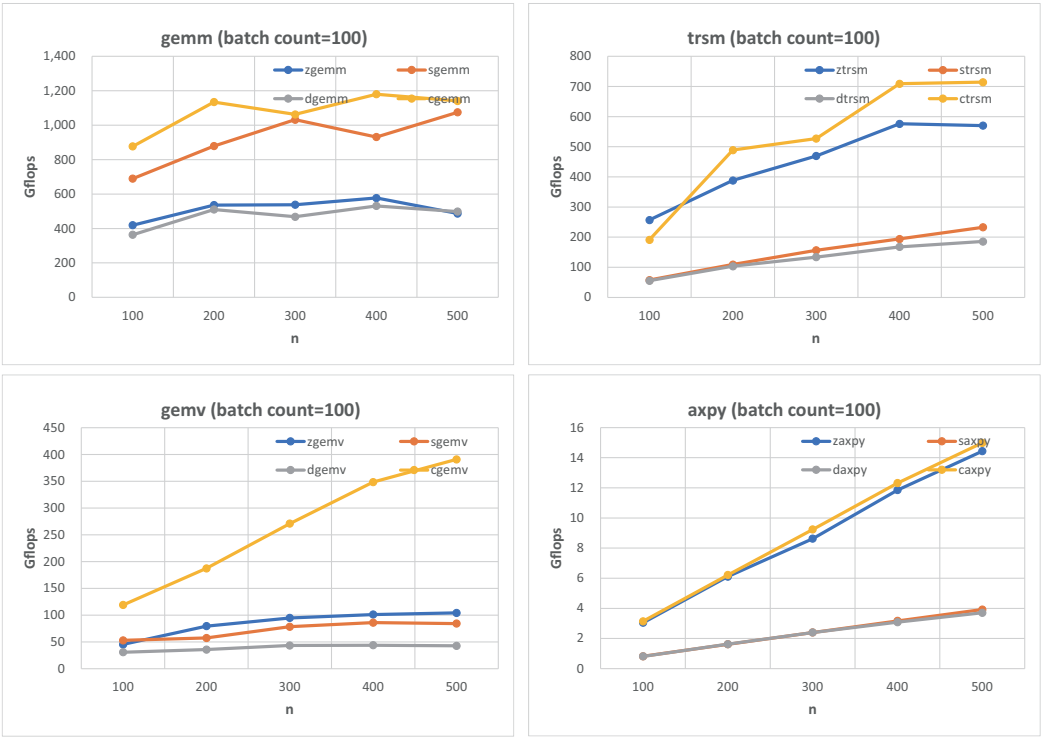


図 3.21: Preliminary benchmark results of Batched BLAS kernels on an AFX64 with a single CMG binding

3.6.2.5 理研ソフトウェア 3: 2.5D-PDGEMM

富岳では、京に比べて通信性能と計算性能のバランスが計算重視になっているため、多くの場合で通信律速状態になりやすい。この問題は、数値計算の基本カーネルであり、計算量の多いタスクとして知られている行列乗算でも発生する。通信回避型分散型行列乗算アルゴリズムである 2.5D algorithm はこの問題を解決する。このアルゴリズムは、2 次元に分散した行列を 2 次元のプロセス上に分散させる従来のアルゴリズム（2D アルゴリズム）に対し、2 次元に分散した行列を 3 次元のプロセス上に複製して積み重ねる（2.5D 分散）ものである。従来の 2D アルゴリズムでは、行列の要素をステップごとに通信で循環させることで、行列の乗算における内積を完成させるが、2.5D アルゴリズムは、従来の 2D アルゴリズムと同じ原理を少数・複数の 2D プロセスグリッド上で使用することで、通信ステップ数を削減する (図 3.22)。

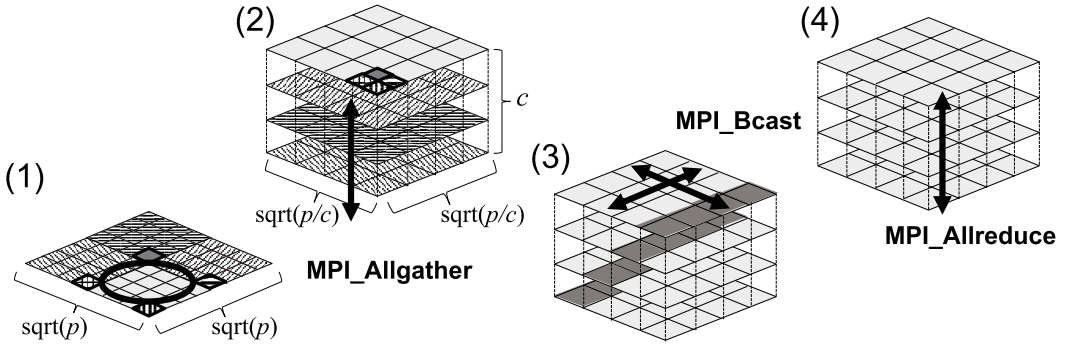


図 3.22: 2.5PD-GEMM の実装方法

開発段階においては、理化学研究所の京と、Joint Center for Advanced High Performance Computing (JCAHPC) の Oakforest-PACS システム (OFP) などの高並列環境を活用し、手法の正当性ならびに性能の評価・解析を行った。計算性能と通信性能のバランスを考えると，OFP の方が京よりも 2.5D アルゴリズムが有効であることが示された [39] [40]。OFP は比較的富岳に近いシステム構成であり、実機使用の前段階で間接的に同手法の富岳における効果が示される形となった。

図 3.23 は、富岳で実測したストロングスケーリング性能を示している（言語バージョン 1.2.25-02、MPI バージョン M0514）。 図中、‘ScaLAPACK’ は ScaLAPACK の PDGEMM ルーチン、‘SUMMA(c=n)’ はスタックサイズ n の 2D 互換の 2.5D-PDGEMM を示す (‘c=1’ は、2D アルゴリズムと同等)。また、‘NN/NT/TN/TT’ は、オペランドの行列 A と B の転置モードを示す（‘N’: 非転置、‘T’: 転置)。プロセスあたりの問題サイズが小さい場合には，2.5D アルゴリズムが有効であることがわかる．

3.7 運用系ソフトウェア

3.7.1 アプローチ

3.7.1.1 設計・開発方針

京の運用経験およびソフトウェア技術を踏まえ、京からの継続性、京の運用課題の解決による安定した運用の実現、使い勝手のよいシステムの追求、世界トップレベルの成果を創出するための新しい取り組みへのチャレンジ、京を超える規模のシステムを運用可能なソフトウェアの実現、

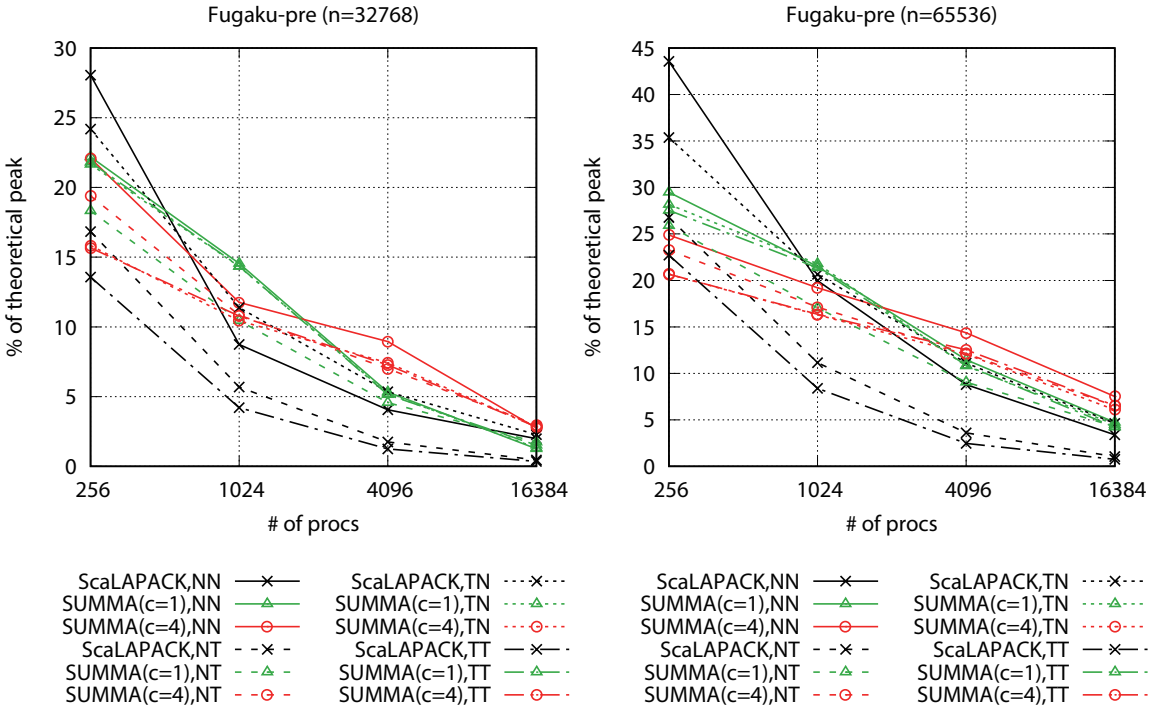


図 3.23: Strong scaling performance of 2D-compatible 2.5D-PDGEMM on the supercomputer Fugaku (on the pre-evaluation environment).

といった観点から運用系ソフトウェアの基本設計を行い、運用系ソフトウェアへの要求事項を以下のようにまとめた。

- 京で開発したソフトウェアの継承
- 京での課題の解決
- 京を超える規模のシステム運用への対応
- 電力管理・電力最適化への対応

3.7.1.2 開発目標

運用系ソフトウェアの開発にあたり、要求事項を踏まえて以下のような目標を設定した。

- 保守性
  - 計算リソース提供率 95%の達成を目標としたソフトウェア保守時間の短縮
    京ではソフトウェア保守に最大 2 日程度かかっていたが、富岳ではソフトウェア保守時間を半減させ 1 日で実施できるようにする。
  - 京での登録課題数・利用者数を超えるアカウント管理
    京を超えるノード数で構成される本体システム、複数の役割をもつ周辺システムにおいてアカウント情報を一元的に管理・操作できるようにする。
- 高可用性
  - システムの一部の故障がシステム全体に波及しない実装
    大規模システムでの一部の故障は、システム全体に波及することにつながりかねない。

これを防ぐため、システムの一部の故障が全体に波及しないような実装とする。また、HA 構成におけるフェイルオーバー処理時間の短縮等、復旧に要する時間を短縮する。
  - 新機能への対応富岳での新たな構成 (階層化ストレージ、McKernel 等) に対応する。

- 運用性
  - 京の運用性の維持と問題の解決
    京の運用性を維持しつつ、京の運用で問題となった課題を解決する。また、富岳の運用開始後や新規ユーザの獲得により新たに生じた運用要件に対して、カスタマイズ機能を用いることで修正提供無しに運用要件を満たせるようにする。
  - 運用情報や障害情報解析の効率化
    システム監視機能や大量に出力されるログの解析手段を改善し、運用性を向上させる。
  - 導入機能の改善
    インストールサポート機能の柔軟性を改善し、ユーザの利便性及び運用性を向上させる。
  - アカウント管理方式の改善
    適切なアクセス権限管理（運用者／課題管理者／利用者毎に参照可能な情報を制限）を実現する。
  - 電力管理・電力最適化への対応富岳では電力を資源の一つとしてとらえ、電力の最適な管理・制御を実現する。

- 利便性
  運用者/利用者の双方が、コマンドまたは Web 画面で各機能を容易に利用できるようにする。
- 大規模対応
  京を超える規模でのシステム制御や運用、ユーザ管理を可能とする。ただし、一般へのソフトウェアの提供を踏まえ、最小構成のシステム (数十ノード規模) でも問題のないような構成を取ることを可能とし、制御や運用でも不自由が無く利用できることも目標とする。

3.7.2 コデザイン成果

開発目標に対する取り組み結果について述べる。

3.7.2.1 保守性

- ソフトウェア保守時間の削減

  富岳では、以下に示すシステムの起動・停止に関わる処理の改善を行い、ソフトウェア保守時間の削減を図った。

  - 計算ノード群の起動にかかる時間の削減
    * 突入電流・電力変動を考慮した起動
    * 計算ノード群の一斉起動
    * 富岳本体システム (BoB) の一括起動
  - 保守後に実施する再起動時間の削減
    * ウォームリブート・サービス再起動
  - 修正適用時間の削減

* ハード構成の変更に伴う修正適用時間の削減

富岳での本体システムの起動時間および修正適用時間の測定結果を表 3.17 および表 3.18 にそれぞれ示す。

富岳の本体システムの起動時間は、目標である 2 時間以内で起動したことを確認した。この測定では、144 ラックごとに 3 回に分けて本体システムを起動している。

表 3.17: 本体システム起動時間の改善結果

システム	**計算ノードの起動時間**
京（実測値）	3 時間 45 分 （45 分 × 5）
富岳（実測値）	1 時間 57 分 33 秒

表 3.18: 本体システム全体のパッチ適用時間

システム	**パッチ適用時間**
京（実測値）	4 時間 30 分
富岳（実測値） （本体システムに運用ソフトのパッチ適用時間）	48 分 40 秒

- 京での登録課題数・利用者数を超えるアカウント管理

  富岳では、 京と同じく利用者管理にディレクトリサービス (LDAP) を用い、その構成は 京を踏襲する。京では、passwd/group ファイルは手動配布であったが、富岳では LDAP の更新を自動検知・自動配布とし、京での登録課題数・利用者数を超えるアカウント管理を確実に実施する。

3.7.2.2 高可用性

- システムの一部の故障がシステム全体に波及しない実装
  富岳の主要な機器は冗長構成となっており、機器が故障した場合はフェイルオーバー処理を行い運用を継続するが、フェイルオーバーが完了するまでは運用に影響がでる場合がある。この時間を短縮するために、システム監視処理の改善を行った。この改善により、京では最大 39 分かかっていたフェイルオーバー処理が、富岳では最大 12 分 10 秒で完了する。
  また、ユーザデータを保存しているファイルシステムに障害が発生した場合の処理も改善した。京では、障害が発生したファイルシステムのみを運用から切り離すことができなかったため、障害から復旧するまで運用を停止する場合があったが、富岳ではシステム監視機能と連携し障害の影響範囲の局所化を行い、障害が発生しているファイルシステムの利用を回避しシステム運用を継続する。
- 新機能への対応
  富岳では、McKernel、階層化ストレージ等の新たな機能が実装された。
  - 仮想マシン上のジョブ実行環境
    富岳では、McKernel を含めた仮想マシンのジョブ実行環境をユーザは利用することができる。このジョブ実行環境はシステム管理者が用意する環境のほか、ユーザが独自に用意することもできる。ジョブ実行環境はジョブ投入コマンドのオプションで指定する方式とし、ジョブ実行時に動的に実行環境を起動し、終了後に通常の実行環境に戻る。

– 階層化ストレージ
ジョブ実行時に第 1 階層ストレージのファイルシステムの利用形態（ノード内テンポラリ領域、共有テンポラリ領域、第 2 階層ストレージのキャッシュ）に対するパラメータを指定できるように実装した。また、経過時間制限超過や計算ノードダウンなどでジョブが異常終了した場合に対し、不完全なファイル (未書き出しファイル) の情報を収集・出力する機能を実装した。
– 任意資源のサポート機能
富岳では、カスタム資源と呼ばれるジョブの実行に必要な任意の資源（電力やソフトウェアライセンスなど）をリソースユニット単位またはリソースグループ単位で定義する機能を実装した。カスタム資源の制限はジョブ ACL 機能で設定する。例えば、商用アプリケーションのソフトウェアライセンスをこのカスタム資源として定義し、ユーザがジョブの投入時にジョブ実行に必要なライセンス数を要求すると、ジョブで使用するソフトウェアライセンス数が足りている時間帯にジョブが実行される。
– アシスタントコアの管理
富岳では、OS ジッタによるユーザプログラムの実行性能低下を軽減するため、ユーザプログラムの実行を阻害するシステム処理 (システムデーモンやハードウェア割り込み処理) をアシスタントコアで行う。さらに、ユーザプログラムの高速化のため、MPI 非同期通信処理や McKernel モードにおけるホスト Linux 上でのプロセス動作処理などジョブの一部処理をアシスタントコアにオフロードする。
アシスタントコアをジョブの処理に利用するためにアシスタントコアをジョブ用資源として管理する。このアシスタントコアの資源管理には Linux 標準の資源管理機能 (cgroups) を利用する。
– 並列ジョブ実行環境の提供
富岳の並列実行環境では、ジョブに割り当てられたノードで MPI プログラムを起動する機能、および、MPI ライブラリ関数実装のために必要な機能として、MPI プログラムを起動するための並列実行環境のコマンド plexec およびリソースマネージャ・ジョブスケジューラなどとの連携やプロセス間通信確立に必要な情報をプロセス間で共有するための API である PMIx (Process Management Interface - Exascale) ライブラリとのインターフェースを実装した。このインターフェースを利用することで、富士通製 MPI 以外の MPI ライブラリ（MPICH など）を利用するジョブの実行が可能となる。

3.7.2.3 運用性

- 京の運用性の維持と問題の解決
  – ジョブスケジューリング機能の改善
  京の運用において生じだ問題を解決するために、ジョブスケジューラー機能の改善および機能拡張を実施した。富岳では運用形態・状況に応じてジョブの優先度制御およびジョブスケジューリングアルゴリズムを自由にカスタマイズすることが可能となるスケジューラー API を実装した。また、きめ細やかな運用設定を実現するため、リソースグループ機能の改善を行った。
  – ユーザインターフェースの改善
  ユーザインタフェースの改善として、コマンドのオプション体系の改善およびジョブ管理系コマンドの独自実装を可能とするコマンド API を実装した。この API を利用す

ることにより、ユーザは自由にコマンドをカスタマイズすることが可能になる。このAPI の利用権限制御は標準コマンドと同様にジョブ ACL を用いて実装されている。

– ジョブ統計情報の拡張
ジョブ統計情報は、ユーザのジョブ資源利用状況の把握や性能チューニングの指標としての用途、および、システム管理者による運用改善のためのシステムの統計処理やジョブ課金情報の収集を目的として利用される。京のジョブ統計情報に対し、富岳において追加すべき項目について検討し実装を行った（表 3.19）。

表 3.19: 統計情報への追加項目結果

分類	項目	検討結果
ジョブ性能・特性	Tofu PA 情報	追加する
	CPU PA 情報	追加する
	階層化ストレージ関連（IO 量など）	追加する
	電力関連（消費電力、パワーノブ操作記録）	追加する
	アプリケーション情報（コンパイル情報、MPI 関連情報等）	追加しない
利用資源	セクタキャッシュ、ハードウェアバリア	追加する
	アシスタントコア	追加する
	カスタム資源	追加する
割当・実行時情報	スケジューリング情報（フェアシェア値、epoint 評価値）	追加しない
	マスターワーカー ノードダウン時間積	追加する
	pjsub オプション	追加する
	使用ノード論理座標	追加する
	McKernel 関連情報（起動時間、終了時間等）	追加する

- 運用情報や障害情報解析の効率化
京では、システム運用状況の確認やトラブル調査におけるログ解析に時間を要していた。富岳では、ログの一元管理および関連するログのひも付けを行いログ解析の効率化を図る。出力ログのフォーマットの統一、ログ収集の効率化を行うほか、オープンソースのログ解析ツールを活用し高速なログ検索を実現する。
- 導入機能の改善
京では容易にシステムを構築できるように、少ないオペレーションでシステム全体を一括で構築できる方式であった。そのため構築作業の自由度が少なく、構築作業の途中でミスが発生すると復旧するまでに時間がかかるなどの問題があった。富岳ではシステム構築の各機能をホワイトボックス化し、オペレーションを機能単位で分割する方式とした。
保守の面では、運用ソフトで一括操作・設定チェックを行う方式とした。これにより、システム管理者はノード単位の自由な設定変更が可能となり、設定チェック結果を確認後、必要に応じて再設定を行うことができる。
- アカウント管理方式の改善
京では、計算資源は課題単位で管理され、課題が複数のサブ課題の集合体で構成されているような場合、課題代表者が課題内で自由に計算資源の割り当てを変更することができなかった。富岳では課題を階層管理とし、階層管理の枠組みの範囲でシステム運用者および運用者が許可したユーザが自由に計算資源を管理できるようにする。アクセス権限についても階層管理に応じて適切なアクセス権限を設定し、システム運用者の負担軽減およびユーザの利便性の向上を図る。
- 電力管理・電力最適化への対応

富岳の消費電力は 京よりも多く電力施設への負荷も大きくなる場合があることからシステムの消費電力のキャッピング機能を実装した。全計算ノードの消費電力の上限をまとめて管理し、個々の計算ノード単体の性能は低下させずに同時に実行するジョブの数を調整することでシステム内の計算ノード全体の消費電力を抑える。ジョブ投入時にジョブの消費電力を予測し、その値を要求電力としてジョブのスケジューリングを行う。システム全体の消費電力が上限値を超過しそうな場合は、自動的にジョブを停止し消費電力を削減する。
また、計算ノードや外部機器において計測された消費電力や消費電力量を収集するシステム電力の収集機能およびシステム電力の収集・可視化支援機能内部で保存している消費電力情報を出力するシステム電力の可視化支援機能を実装した。
ユーザが電力制御を行うための機能としてパワーノブを実装した。富岳の CPU である A64FX は性能を維持しつつ消費電力を削減する機能を持っており、ユーザは Sandia Power API またはジョブスクリプトに指示行を記述することでパワーノブを使用することができる。

3.7.2.4 利便性

富岳では前述の通り、各種機能を容易にカスタマイズできるようにスケジューラ API やコマンド API を実装し、システム管理者がジョブスケジューリングアルゴリズムをシステムソフトウェアの修正をすることなく変更できたり、ユーザが自身で各種コマンドをカスタマイズすることを可能とした。また、ポータル機能として申請書の管理機能や各種情報の公開機能を実装したほか、NEWT ベースの REST-API によるインターフェース機能を実装し，既存の OSS・フレームワークを利用した様々なサービスとの連携を可能とした。

3.7.2.5 大規模対応

富岳では、以下の要件を満たすシステム管理機能を実装した。

- 多数のハードウェアやソフトウェアを、一つのシステムのように利用可能
- システムの一部が故障しても、全体としては正常に運用を継続可能
- 必要に応じてシステムを分割し、互いに干渉しない運用環境を構築可能

システム管理機能は、構成管理機能、システム制御機能、システム監視機能、システム保守機能、運用支援機能、ログ管理機能、インストール機能、バックアップ・リストア機能から構成され、大規模なクラスタシステムの導入や、運用、保守において、システム管理者による効率のよい、安定したシステム運用を実現する。

第4章　ターゲットアプリケーションのコデザイン

4.1　アプリケーションにおけるコデザインの概要

4.1.1　コデザイン手法とその対象

コンピュータシステムとアプリケーションのコデザインとは、アプリケーションを開発する立場からは、将来利用可能になるであろう高性能コンピュータシステムの利用を前提として、そのような環境における計算成果の早期創出を実現するために多面的な設計開発整備を行う手法を示す広義の概念であると言える。富岳の開発においては、その基本的な仕様案と消費電力の上限値を念頭に置きながら計算科学アプリケーションの高性能計算実現を特に重視して、富岳の詳細仕様とアプリケーションの詳細仕様とを協調的・相互的に最適化するコデザインプロセスを実施して設計開発を進めた。以下、アプリケーションを開発する立場から、コデザイン手法とその適用対象・課題を概観する。

まずシステム全体のアーキテクチャに注目した、巨視的な観点のコデザインとして、以下のような課題があげられる。

- 超並列計算のためのデータ分散／処理分散の設計と改良
- 多重パラメータ試行のための実行フレームワーク
- 階層的な並列化の適用

また主にノードのマイクロアーキテクチャに注目した、より微視的な観点のコデザインとして以下のような課題があげられる。

- 既存システム上での性能評価と富岳性能推定手法の確立
- アプリのチューニングパターンと富岳性能への効果検証
- 富岳高性能実現に必要な改善課題の特定
- コンパイラ・MPI・数値計算ライブラリの最適化機能

前者（巨視的なコデザイン）は、各アプリケーションがシステムのアーキテクチャ（基本的なハードウエア・ソフトウエア構成の仕様）を念頭におきながら、計算性能・消費資源の両面においてふさわしいプログラムモデルの開発をすすめることを意味する。

システムのアーキテクチャとして主にハードウエアの観点からシステムの規模（ノード数、CPU/CMG/コア数、ノード間ネットワーク等）、プロセッサのタイプ（サポートする命令のタイプ、加速機構の有無・種類、SIMD 演算の種類・効率、メモリの種類・量・バンド幅）、消費電力（CPU、メモリ、ネットワーク）、等の資源が大枠として定まり、それに従って浮動小数点演算最大性能、メモリアクセス最大性能、ノード間通信最大性能などの実行性能を制約する数値上限が定まる。この仮定された制約のもとで必要な機能・性能を発揮する数理手法をアプリケーションプログラムとして開発し、必要であればシステム全体のアーキテクチャへフィードバックを行うのが巨視的なコデザインである。またシステムのソフトウエア構成への要求も同様な位置付けである。

システム全体のアーキテクチャへフィードバックを行うという事は、基本的な仕様構成に影響を与えることになる。基本設計・詳細設計（１）で行ったシステム構成ノード数の決定、メモリタイプの選択、などはこのような巨視的なコデザインによるシステム設計へのフィードバックの例である。

後者（微視的なコデザイン）は、各アプリケーションにおいて以下の様な手順を繰り返しながら、CPU 内部のキャッシュ機構動作制御などのマイクロアーキテクチャにおける設計パラメータの詳細な決定や言語コンパイラが生成する命令列の最適化など個別のシステムソフトウエア詳細設計を、アプリケーション自身の開発過程と同時並行的にすすめ、技術的に最適と判断される設計パラメータのセットとそれに対応して最適化された状態のアプリケーション開発を実現し、富岳上でのアプリケーションの性能評価を可能とする設計手法である。

すなわち、設計パラメータから検討可能な組み合わせを合理的に仮定して、その設計パラメータ値における 富岳上での詳細な性能推定を行う中で性能に影響をあたえる主要なハードウエア資源を特定する。この資源を前提としたアプリケーション側でのさらに高度な最適化可能性を検討する。この最適化と並行して、性能を律速するハードウエア資源をさらに強化した場合どのように性能が変化するかの検討を行う。検討後は次の設計パラメータ組み合わせを仮定して同様な検討を繰り返す。設計パラメータの選択組み合わせを網羅的に行うのは時間的・資源的に困難であるため、技術的・経済的に実装が可能でかつ実装の効果が定量的に評価可能な組み合わせに限定される。この結果、アプリケーションからみて最適と判断される設計パラメタ候補が定義され、マイクロアーキテクチャ開発主体へ具体的な実装を要請することになる。また同時にプログラミング手法としてのアプリケーション最適化の方針が決定される。

このような微視的なコデザインはほぼ全てのアプリケーションに共通する部分と、個別のアプリケーションにおいて固有な手法が必要になる部分とがある。

詳細設計（２）で行った L1 キャッシュの eviction 操作の決定、 インダイレクトアクセスのまとめ上げ機構などのマイクロアーキテクチャへの実装要請や、本詳細設計（３）で行った コンパイラが生成する命令群の変更による性能向上などシステムソフトウエアの機能改善要請などは微視的なコデザインによるシステム設計へのフィードバックの例である。

アプリケーションにおけるコデザインでは上記の巨視的・微視的アプローチがともに必要であり、特に区別されることなく実践される。

以降特に断りのない限りアーキテクチャとはハードウエア・ソフトウエアを含むシステムアーキテクチャを指すこととする。

4.1.2　重点課題とターゲットアプリケーション

富岳の開発においては、コデザイン開始までに、文部科学省「ポスト「京」で重点的に取り組むべき社会的・科学的課題についての検討委員会」のもとに、9 つの重点課題と 4 つの萌芽的課題が公募により採択された。

この 9 重点課題の各実施機関によって、以下の基準で選定されたアプリケーションが、コデザイン対象の重点課題ターゲットアプリケーションとして提示され、採用した。

- 各重点課題の要となる計算手法を有するアプリケーションであること
- アプリケーションの開発体制やライセンス形態がコデザインに適用できるものであること
- 全ターゲットアプリケーション群は、計算科学的手法の網羅性を有していること

9 重点課題とその課題概要、実施機関、ターゲットアプリケーションを表 4.1 に示す。富岳はこれらの重点課題ターゲットアプリケーションを用いてコデザインが行われた。

表 4.1: 重点課題とターゲットアプリケーション

課題番号	課題名	課題概要	課題実施機関	アプリケーション
1	生体分子システムの機能制御による革新的創薬基盤の構築	超高速分子シミュレーションを実現し、副作用因子を含む多数の生体分子について、機能阻害ばかりでなく機能制御までをも達成することにより、有効性が高くさらに安全な創薬を実現する。	理化学研究所生命システム研究センター他 5 機関	GENESIS
2	個別化・予防医療を支援する統合計算生命科学	健康・医療ビッグデータの大規模解析とそれらを用いて得られる最適なモデルによる生体シミュレーション（心臓、脳神経など）により、個々人に適した医療、健康寿命を延ばす予防をめざした医療を支援する。	東京大学医科学研究所他 5 機関	Genomon
3	地震・津波による複合災害の統合的予測システムの構築	内閣府・自治体等の防災システムに実装しうる、大規模計算を使った地震・津波による災害・被害シミュレーションの解析手法を開発し、過去の被害経験からでは予測困難な複合災害のための統合的予測手法を構築する。	東京大学地震研究所他 4 機関	GAMERA
4	観測ビッグデータを活用した気象と地球環境の予測の高度化	観測ビッグデータを組み入れたモデル計算で、局地的豪雨や竜巻・台風等を高精度に予測し、また人間活動による環境変化の影響を予測し監視するシステムの基盤を構築する。環境政策や防災、健康対策へ貢献する。	海洋研究開発機構地球情報基盤センター他 3 機関	NICAM+ LETKF
5	エネルギーの高効率な創出、変換・貯蔵、利用の新規基盤技術の開発	複雑な現実複合系の分子レベルでの全系シミュレーションを行い、高効率なエネルギーの創出、変換・貯蔵、利用の全過程を実験と連携して解明し、エネルギー問題解決のための新規基盤技術を開発する。	自然科学研究機構分子科学研究所他 8 機関	NTChem
6	革新的クリーンエネルギーシステムの実用化	エネルギーシステムの中核をなす複雑な物理現象を第一原理解析により詳細に予測・解明し、超高効率・低環境負荷な革新的クリーンエネルギーシステムの実用化を大幅に加速する。	東京大学大学院工学系研究科他 11 機関	ADVENTURE
7	次世代の産業を支える新機能デバイス・高性能材料の創成	国際競争力の高いエレクトロニクス技術や構造材料、機能化学品等の開発を、大規模超並列計算と計測・実験からのデータやビッグデータ解析との連携によって加速し、次世代の産業を支えるデバイス・材料を創成する。	東京大学物性研究所他 8 機関	RSDFT
8	近未来型ものづくりを先導する革新的設計・製造プロセスの開発	製品コンセプトを初期段階で定量評価し最適化する革新的設計手法、コストを最小化する革新的製造プロセス、およびそれらの核となる超高速統合シミュレーションを研究開発し、付加価値の高いものづくりを実現する。	東京大学生産技術研究所他 6 機関	FFB
9	宇宙の基本法則と進化の解明	素粒子から宇宙までの異なるスケールにまたがる現象の超精密計算を実現し、大型実験・観測のデータと組み合わせて、多くの謎が残されている素粒子・原子核・宇宙物理学全体にわたる物質創成史を解明する。	筑波大学計算科学研究センター他 7 機関	LQCD

4.1.3　アプリケーションの計算手法と主なコデザイン観点

次節以降に、各アプリケーションとそのコデザインに関する具体的な内容が詳細に説明されるが、本節では、各アプリケーションの計算手法の概要とその特徴、コデザインにおける主な観点を抽出して示す。

- GENESIS

  GENESIS は、分子動力学計算を行い粒子の軌跡を追跡する、粒子系計算アプリケーションである。粒子間の力と座標を時間発展で計算し、計算時間の大半は粒子間の力の計算に費やされる。シミュレーション対象領域は一定サイズのセルで領域分割され、各プロセスが担当セル内の原子の力の計算・時間発展を並列に処理する。力は分子の共有結合に関わる結合性相互作用と、分散力と静電相互作用からなる非結合性相互作用に分けられ、粒子数の増加に伴って非結合性相互作用の計算量が支配的となる。非結合相互作用はさらに粒子間距離により短距離と長距離に分けて計算される。

短距離の計算では、ペアリストに基づいて各粒子の情報をメモリから読み込んで直接計算を行うため、リストデータの読み込みと力の計算を各々SIMD 命令により最適に実行することがコデザインの主要な観点となる。

長距離の計算ではフーリエ変換による逆格子空間での計算により演算量を削減している。対象領域の全電荷に対する 3 次元 FFT 計算を FFT 前処理、FFT 計算、FFT 後処理、MPI 全体通信の組み合わせとして実施するため、高性能な FFT ライブラリがシステムに実装されること、および FFT 前後処理は SIMD 命令により効率良く演算が実行されることがコデザインの観点となる。

- Genomon

  Genomon は、がんのヒト全ゲノム関連データ解析プログラムである。入力データを加工して出力する複数の小規模なプログラムを、パイプライン的に実行してゲノム解析を行う、ワークフロー型のアプリケーションであり、所与の時間内に解析可能な検体数を最大化すること（スループット性能）が求められている。

  Genomon の特徴としては、文字列を対象とした整数演算が中心であること、パイプライン中にファイル I/O 律速なプログラムが存在すること、検体あたり 1TB 規模の入出力ファイル群に対して複数のプロセスから同時に I/O が発生すること、プログラムの多くがスクリプト言語 Python で記述されていることがあげられる。

  従って、整数演算の高速な実行を実現すること、ファイル I/O の性能を確保すること、がコデザインの主要な観点となる。また、Python とその主要 Python ライブラリである NumPy/SciPy が最適化された状態であることが求められる。

- GAMERA

  GAMERA は地盤地震動シミュレーションのための陰解法・非構造格子有限要素法プログラムである。各時間ステップにおける大規模な連立一次方程式の求解には、共役勾配法をベースに精度混合演算・マルチグリッドを組み合わせたソルバーが使われており、共役勾配法の行列ベクトル積には Element-by-Element(EBE) 法が使われる。

  EBE 法では要素行列を行列ベクトル積を実行する度に計算する。要素行列と要素・節点マッピング行列のデータを格納して計算を進めるため、一般的な有限要素法の手法である全体係数行列を格納して計算する方法に比べてデータ量が大幅に少なくなり、メモリアクセス量を減らすことができる特徴をもつ。

  計算時間の大部分は単精度 EBE カーネルが占め、要素行列を計算するために多くの変数を使った計算がキャッシュ上で必要となり、変位ベクトルのランダムロードと外力ベクトルへのランダムアクセス・更新が必要になる。

  EBE 法の計算において SIMD 命令を効果的に適用して性能を得ること、間接インデックスによるランダムなベクトルのロード・ストアをキャッシュレベルで効率よく実現すること、がコデザインの主要な観点となる。

- NICAM+LETKF

  NICAM+LETKF は全球高解像度気象モデル NICAM とアンサンブルデータ同化システム LETKF から構成され、地球大気の物理モデルシミュレーションと観測データを融合しながら時間発展を解く複合アプリケーションである。

NICAM は構造格子系を用いたステンシル計算に分類され、メモリ・キャッシュの転送性能やレイテンシの評価、スレッド並列やスケジューリング、アウトオブオーダー実行による演算効率の向上などがコデザインの主要な観点となる。

LETKF はデータ同化アプリケーションであるが、同時にフレームワークとしても機能し、主要演算部分では単一ノードに収まる小規模な固有値計算の性能、フレームワーク機能では NICAM と共有する大量のデータファイル I/O サービスと大域通信性能の評価が、それぞれコデザインの観点となる。

- NTChem

  NTChem は大規模分子の第一原理的電子状態計算を目的としたアプリケーションである。

  NTChem には高精度な電子相関理論が複数実装されている中で、Resolution-of-Identity(RI) MP2 法に基づく電子相関エネルギー計算を対象としてコデザインを進めた。RI-MP2 モジュールでは電子反発積分計算における行列要素の計算や行列-行列積計算などの密行列演算が主要な演算処理となる。ターゲット問題およびそれ以外の中規模・小規模行列に対して高い計算性能を発揮する密行列積計算ルーチンを汎用ライブラリとして実装すること、および関連して要求 B/F 値が低く演算量の多い計算を効率的に実行することがコデザインの第一の観点である。

  また当初は、ターゲット問題として設定したレイアウトによるプロセスあたりの必要メモリサイズが 30GiB を超えていた。プロセスあたりの必要メモリサイズが富岳の CMG あたりのメモリ容量以下となるように、プログラムの分散並列化をさらに進めてメモリサイズを縮小することが重要な課題となった。このため並列軸の階層的な変更、リング通信の採用、2 中心積分・3 中心積分に新しい分散アルゴリズムを採用するなど、広範囲に見直しを行った。分散並列の高度化によるメモリサイズの削減が、コデザインの第二の観点である。

  通信においては MPI 集団通信と 1 対 1 通信を併用し、通信データ量も多い。このため MPI 通信ライブラリの性能の高さが要求される一方、利用する側においても、1 対 1 通信のホップ数が特に高くなるペアが生じないように適切なランクマップを考慮する配慮が必要である。

  また、計算時間短縮を目的とした倍精度計算と単精度計算の部分的な混合について、性能への効果と計算精度に及ぼす影響に関して詳細な検討が必要である。

- ADVENTURE

  ADVENTURE は有限要素法による構造解析プログラムである。領域分割法による並列処理を行い、領域間境界上節点の変位に関する連立一次方程式を、共役勾配法によって反復的に解くのが ADVENTURE の主要な計算部分である。ADVENTURE では反復回数を減少させるために、部分領域毎の対角スケーリングと、コースグリッド修正の 2 段階からなる前処理を、反復 (BDD 反復) の各回に行う。BDD 反復での主要な計算は密行列ベクトル積 (倍精度演算、倍々精度演算) であり、その性能がコデザインの主要な観点となる。また、隣接領域境界面や全体での通信が発生し、メッセージ長の短い集約通信（MPI_Allgather）が特に重要となる。

- RSDFT

  RSDFT は実空間差分法に基づく密度汎関数法による第一原理電子状態計算および構造最適化計算を行うアプリケーションである。RSDFT の性能は、演算性能が DGEMM に依存し、

通信性能が集団通信性能に依存するため、これらがコデザインの主要な観点である。ターゲット問題の計算において、初期に評価された 1 プロセスあたりメモリ使用量は 10.8GiB であり、富岳の CMG あたりメモリ容量を超えるため、新アルゴリズムの開発とその実装によるメモリ分散の高度化が必要である。

- FFB

  FFB は有限要素法に基づく流体アプリケーションである。

  流れ場の計算は、移流・拡散項の計算により速度予測子を求め、速度予測子から圧力を求め、圧力から速度予測子を修正して速度を決定する。速度予測子と圧力の求解は陰的に行われる。計算に用いる行列は疎行列であり、非構造格子を用いるためメモリアクセスのパターンは入力データ依存のランダムアクセスとなり、メモリ転送量が増えるという傾向がある。

  陰解法ソルバーとして共役勾配法を用いており主要な演算は疎行列とベクトルの積であることが特徴である。

  リストによるインダイレクトアクセスで B/F 要求値が高く、高いメモリスループットを要求することがコデザインの観点となる。

- LQCD

  LQCD は素粒子標準模型の理論に基づく一般的な格子 QCD アプリケーションである。LQCD において計算時間が最もかかるのは、ディラック方程式を 4 次元格子上で離散化した大規模連立方程式を解く QCD クォークソルバーと呼ばれる計算である。ウィルソン型の離散化によれば、その係数行列は構造格子に基づく大規模ステンシル疎行列となり、一般に反復法で解かれる。反復法による連立方程式の求解はしばしばメモリバンド幅に律速される。メモリバンド幅律速の解消あるいは低減がコデザインの主な観点である。

  また、大規模な系を解くためには適切な MPI 並列化が必要であるが、構造格子の計算では計算領域と隣接する領域との袖データの MPI 通信による交換が全ての領域において発生する。構造格子袖データの交換通信の効率化あるいは隠蔽化もコデザインの観点である。

表 4.2: 計算・通信の種類によるアプリケーションの分類

アプリケーション	構造格子計算	非構造格子計算	粒子法計算	密行列計算 (局所的)	密行列計算 (大域的)	通信・低レイテンシ	通信・高バンド幅	隣接通信	大域的通信	大容量 I/O
GENESIS			○					○	○	
Genomon										○
GAMERA		○				○	○		○	
NICAM+LETKF	○			○			○	○		○
NTChem				○	○		○	○	○	
ADVENTURE		○		○	○	○		○	○	
RSDFT	○			○			○	○	○	
FFB		○						○	○	
LQCD	○					○	○	○	○	

上述した重点課題ターゲットアプリケーションは全体として、計算・通信の種類を広く網羅する。重点課題ターゲットアプリケーションをその計算空間の種類、プロセス間通信の種類、I/O 負荷によって分類した表 4.2 を示す。

重点課題ターゲットアプリケーションは全て富岳の開発開始以前から開発が進められてきたものであるが、各アプリケーションにおいて富岳に向けた適切なコード整備あるいはフレームワークの再構成を行う開発作業が行われることになり、富岳が稼働開始する時点で、期待される計算性能が十分に発揮される状態になることを目指すものである。重点課題ターゲットアプリケーションのコデザインを通して知見として蓄積される富岳向け推奨プログラム事例等は、一般の計算科学アプリケーション開発者が参照することができるように、整理して文書化することとした。

重点課題ターゲットアプリケーションにおけるコデザインの実際の進行は、各 アプリケーション 毎に組織したワーキンググループ (WG) の中で詳細な検討と開発を進めながら、全 WG の代表者が集うアプリケーション検討会において情報共有並びに関連するテーマ毎の議論を行う形式で進行した。

4.1.4 性能評価用ターゲット問題の設定

富岳上での各ターゲットアプリケーションの性能を評価する基準として定められた具体的な計算問題の設定はターゲット問題と呼ばれる。ターゲット問題設定を表 4.3 にに示す。

表 4.3: 重点課題ターゲットアプリケーションの問題設定

課題番号	アプリケーション	並列処理のタイプ	ノード数/ジョブ	問題設定
1	GENESIS	多重処理	1	より効果的で安全な創薬候補物質のスクリーニングのため、タンパク質、溶媒 (水分子) を含む 10 万原子の薬剤候補結合の全原子分子動力学シミュレーションを 1 ケースにつき 100ns 行う。10 万ケース実行。
2	Genomon	多重処理	96	がんの解明に向けた全ゲノム解析。リード長 150 塩基対、リード数 14 億。1 日あたり 1000 検体のスループット実現。
3	GAMERA	大規模単一	158976	都市域の地盤歪の計算に該当する、1 兆自由度の非構造格子有限要素モデルの非線形地盤地震動解析 (120 秒、12 万時間ステップ) を実行する。
4	NICAM+ LETKF	多重処理 + 大規模単一	131072	局地的豪雨や竜巻などの高精度な予測を実現するため、多重ケース処理型計算では、全球 3.5km 水平メッシュ1024 メンバーのアンサンブル気象計算と観測データとの同化を 3 時間毎に行うシミュレーションを 2ヶ月間分、大規模単一問題型計算では、全球 220m 水平メッシュ鉛直 94 層の大気シミュレーションを 72 時間分実行する。
5	NTChem	多重処理	17820	化学反応のメカニズムを解明し、エネルギー変換材料の候補物質スクリーニングを行うための高精度第一原理電子状態計算の典型的計算例として、720 原子、19680 電子軌道のカーボンナノグラフェン分子複合体のエネルギー計算を設定する。20 ケース実行。
6	ADVENTURE	多重処理	4096	複雑な形状の構造物の最適な全体設計を実現するため、有限要素法に基づく構造解析を行う。薄板状領域の非線形応答問題。ソリッド四面体二次要素分割で計 16.5 億自由度。1 時間ステップ当たり BDD 反復 500 回を 1 万時間ステップ実行。100 ケース実行。
7	RSDFT	多重処理	10368	複数の異種物質から構成されるナノ界面を解明するため、量子力学的第一原理計算に基づき、原子数 11 万・バンド数 22 万・SCF200 回の SCF 計算によるシリコンデバイスの構造最適化の計算を行う。24 ケース実行する。
8	FFB	大規模単一	158976	水力機械の内部流れや自動車の空力の乱流計算など、複雑な形状の構造物まわりの熱発生率、冷却・排気損失、ノッキング、サイクル変動等の予測の正確な評価を行うため、有限要素法に基づく流体計算を 6700 億要素規模で 10 万時間ステップ行う。
9	LQCD	大規模単一	147456	素粒子から宇宙全体にわたる物質創成史を解明するため、クォークを 192^4 個の格子上の場としたクォーク伝搬関数を反復解法で計算する。

4.1.5　富岳向け性能推定の基本的方針

重点課題ターゲットアプリケーションの富岳向け性能推定は、アプリケーションの性能を律速する重要な計算区間をカーネルコードとして切り出して、性能電力予測ツール、複数の性能シミュレータ、検証エミュレータ、試作機 CPU など、各設計期間において利用可能な様々な性能推定用ツールを用いてカーネルコードの評価性能の精緻化をはかるというのが基本方針である。性能評価のプロセスはアプリケーションのソースコードレベルでの性能チューニングと並行して実施され、多数回の性能評価作業が行われた。本報告書における各ターゲットアプリケーションの富岳向け性能推定の説明は、この性能評価作業の要約に相当する。

4.2 重点課題 1 GENESIS のコデザイン

4.2.1 アプリケーションの特徴とコデザインの方針

GENESIS(GENeralized-Ensemble Simulation System) は、理化学研究所で開発された MD ソフトウェアであり、主にタンパク質、核酸などの生体分子の研究に用いられている。LGPLv3 ライセンスのフリーソフトとして公開されている。[6]

分子動力学 (MD) 計算は、分子の運動方程式を解くことにより、その軌跡を追跡する手法である。また自由度が膨大であるため、一定の時間幅（ステップ）毎に粒子間に働く力を計算し、数値的な時間発展を用いる。アプリケーションは、初期設定の後、粒子間の力の計算と、座標の時間発展をステップ回数だけ繰り返す。MD 時間の演算時間の大半は粒子間の力の計算である。

GENESIS の並列化手法としては、シミュレーションシステムをある一定サイズのセルに区切り、セルを各プロセスに割り当てるドメイン分割法を用いている。各プロセスは担当するセル内の原子の力の計算・時間発展を行い、プロセス間通信を大幅に減少している。この手法では、原子情報等も担当分のみを保持するため、メモリ使用量が減少され、大量なメモリアクセスを必要としない。

GENESIS では、力の計算は粒子間距離に依存した経験的なエネルギー関数を用いる。力は分子の共有結合に関わる結合性相互作用と、分散力と静電相互作用からなる非結合性相互作用に分けられる。結合性相互作用は粒子の数 (N) に対して計算量は、O(N) であるが、非結合性相互作用は $O(N^2)$ となり、粒子数が多くなると膨大な計算量となる。そこで、非結合相互作用は距離（カットオフ長）を基準として、各ペアに対して直接計算を行う短距離と長距離の二つの部分に分けて計算される。GENESIS では長距離の静電相互作用はフーリエ変換 (FFT) を使って逆格子空間で計算を行う smooth particle mesh Ewald (SPME) 法を用いる。（今回の問題では、長距離の分散力を考慮しない力場を使用）カットオフの導入により演算量は大幅に削減されるが、依然として非結合性相互作用が演算時間の大半を占める。

短距離相互作用での演算量はペアの数と力の計算の演算数から決められる。また、演算中にメモリに格納された各粒子の情報を適宜読み込むことが必要となる。そのため、SIMD による効率の良い演算または（かつ）ペア数の減少が速度を上げるためには必要となる。

長距離相互作用は前段で述べたようにシステム上の全電荷に対する 3 次元 FFT 計算が必要となる。これは複数回の FFT 単体と、全体通信 (allgather、alltoall) の組み合わせとして実施される。また、更に各粒子の電荷を FFT 用のグリッドに分配する部分 (FFT 前処理) と FFT 実施後にグリッド上に振り分けられた力を各粒子に分配する部分 (FFT 後処理) も必要となる。演算時間に対する割合としては、全体通信と FFT 前後処理の演算時間が大きな割合を占めている。通信の高速化のため通信 WG、FFT カーネルの高速化を目指す数値計算ライブラリ WG とのそれぞれ連携を行った。また、FFT 前後処理に関しては、SIMD による効率の良い演算が必要となる。

GENESIS は、粒子系計算として、演算における SIMD 最適化、全体通信と隣接通信を必要とするアプリケーションとしてコデザインに寄与する。

2017 年 4 月までは、温度・圧力制御をしない NVE(粒子数・体積・エネルギー一定) 条件を用いた MD 計算での評価を行ってきた。生物物理分野に関してかつては、NVE 条件での計算はボトルネックとなる非結合相互作用計算の性能を反映しやすく、MD プログラムの性能評価では広く使用されていたためである。その一方、実際の薬剤候補結合計算においては、生体での温度・圧力に設定された条件——いわゆる NPT(粒子数・圧力・温度一定) 条件——が用いられる。そのため、重点課題 1 実施機関から、実際の薬剤候補計算に対応する NPT 条件での評価を要求されていた。

この要請を受け、GENESIS 開発チームは NPT 条件での高速で高精度な数値積分スキーム [29] の開発を行った。

重点課題 1 実施機関とも打ち合わせを行い、1) 開発された数値積分スキームを導入した GENESIS を用いて、コデザインにおけるポスト「京」性能評価を行うこと、2) 温度・圧力制御部分を評価区間に加えること、の 2 点を確認した。

また、数値積分スキームの改善により、時間ステップ幅に対して従来の 2fs から 2.5fs への増加が可能となった。

さらに、短距離、長距離相互作用計算など、数値積分の精度に大きく影響しない事が確認されている部分を単精度計算での評価に切り替えた。

ターゲット問題は、薬剤候補結合計算としてタンパク質、溶媒 (水分子) を含む 10 万原子の全原子 MD シミュレーションを 1 ケースにつき 100ns 行う。課題全体では 10 万ケースの計算を行い、問題としては多重ケース処理型計算である。そのため、比較的少ないノード数での演算と、1 ステップに 6 回 (ベースライン) もしくは 5 回 (詳細設計 (1)) の全体通信が特徴と言える。京との性能倍率に関しては、単体性能の向上率に富岳と同時実行可能なケース数の比を乗じている。

4.2.2 短距離相互作用計算での演算待ちの減少

短距離相互作用部分 (区間 Nonb15F) は京における性能評価では、全体の演算時間の約半分を占めており、この区間の高速化は最重要課題である。PA 情報から、浮動小数点演算待ちと浮動小数点ロード L1D アクセス待ちのコストが Nonb15F の 7 割近くを占めることが明らかとなった。オリジナルコードでは短距離相互作用を行う範囲 (カットオフ) 内のセルペアを単位として考え、最内のループ回転数はセル内の原子数となっていた。GENESIS が採用するドメイン分割法 [31] では、ペアリストやループの構造が単純となる利点があるものの、セル内の原子数は 40-60 程度である。ループ長が短くソフトウェアパイプラインの効果が十分に発揮されていない事が原因であった。そこで、基本設計において、セルペア単位のループを、原子単位のループへと変更を行った。この変更により、短距離相互作用ペアを格納したペアリストの生成・更新 (区間名:PairList) も合わせて修正を行った。原子単位への変更により、ペアリスト・ループ構造共に複雑となり、新たなリスト配列が必要となった。この修正により、Nonb15F における浮動小数点演算待ちは 1/10 程度に減少したが、新たなリストアクセスの出現により浮動小数点ロード L2 アクセス待ちが区間の半分を占めるようになった。また、PairList における演算時間も増大した。

4.2.3 短距離相互作用計算での浮動小数点ロード L2 アクセスの減少

基本設計での修正により、新たなリストアクセスが出現し、浮動小数点ロード L2 アクセス待ちが区間の半分を占めるようになった。そこで、基本設計から詳細設計（1）の間で、GENESIS 開発者により一つのペアリストから複数のリスト値を与えるように改変し、整数演算は増えるがリスト配列の数自体を減らす修正を行った。また、virial 計算の部分も分離し、演算数を減少させた。これらの改変により、アクセス待ちと浮動小数演算数が減少し、この区間の演算時間は基本設計時に比べて約 8 割程度となった。

4.2.4 短距離相互作用計算でのペアリストを使用しないアルゴリズムの開発

詳細設計（1）の後、2016 年度の実施項目として、性能向上を阻む要素の一つであるリストアクセスを減らすアルゴリズムの開発を行った。まず、短距離相互作用計算でのペアリストを使用しないアルゴリズムを GENESIS 開発チームが開発し、検証を行った。ペアリストを使用しないアルゴリズムは、基本設計時から企画されていたが、1. 分子内相互作用と、2. 演算数の増加という 2 つの問題があり、実施されていなかった。分子内相互作用については、創薬計算で扱う生体分子では、短距離相互作用計算を行わない。これまでの解析において、すべてのペアについて相互作用の足し合わせをした後に分子内相互作用分を差し引くアルゴリズムは、単精度演算では科学的に無視できない誤差が生じることを確認している。演算数の増加については、ペアリストを使用しないことでカットオフ外の原子ペアの演算を行い、つまり演算数は粒子数に対して $O(N^2)$ となるため、SIMD 効率では補えないほどの増加となる。

これらの 2 つの問題点を克服するべく、原子ペアにおけるそれぞれの原子が所属するセルをカテゴリー分けするアルゴリズムを開発した。これは、原子ペアを同一セル内、隣接セル間、それ以外の近傍セルに分ける。分子内相互作用の扱いは同一セル、隣接セル間までのみで考えるようにセルのサイズ、もしくは隣接セルの基準値を調整する。さらに、1 バイト整数によるマスキングを行い分子内相互作用を排除した。また、セル情報更新のタイミングで原子ペアの確認を行い、ある原子に対してカットオフにバッファーを加えた距離以内の原子が存在しないセルについて演算をスキップする処理も追加した。このアルゴリズムの利点は最内のループは原子単位のループとなり、座標や力についてはリストアクセスを利用しない。その結果、SIMD による最適化はなされた。さらに、最内ループ長を変えるためにセルサイズを変更させたカーネルを 5 つ作成し、最も高速なコードを得た。

しかし、詳細設計（1）のカーネルと比較して、演算数は 1.3 倍で抑えられているにも関わらず、演算時間は 2.4 倍となった。また、座標や力の配列の形状をループ変数に合うように変更したが、変更前も SIMD 最適化が効いていたため性能の向上とはならなかった。演算時間が伸びた理由は、浮動小数点ロード L1D アクセス待ちが 3 倍程度上昇したことと、詳細設計（1）のカーネルでは、ほとんど見られなかった浮動小数演算待ちが全体の半分を占めるほどに増大したためである。

この浮動小数演算待ちが増大する問題は、GENESIS カーネルに限られたものではなく、MD 計算のプロトタイプであるカーネルコードでも 4 割が演算待ちであることが報告された。このカーネルコードは、クーロン相互作用のみを力場とした 1 種類の単原子分子に対応する単純な物であるが、lookup table、ペアリスト、原子種毎に異なるパラメータの読み込みなど、GENESIS で指摘されてきたリストアクセスがすべて排除された「理想的条件」の MD 計算となっている。そのため、カーネル WG を通して議論を行い、コンパイラの修正を求めた。さらにチューニングを行った結果、浮動小数演算待ちは縮小された。

4.2.5 短距離相互作用計算での lookup table について

GENESIS では、短距離の非結合性相互作用に対しては精度良くかつ高速に計算を行うための Inverse lookup table 法 [30] を用いている。これは、クーロン相互作用と van der Waals 相互作用を、短距離と長距離のエネルギー項をスムーズにつなぐための switch/shift 関数も含んだエネルギー計算を一定の間隔の距離に対して行い、配列に格納するものである。配列の格納は初期設定時に一回行うだけで良い。利点は、複雑な多項式、補誤差関数を含むエネルギー計算が 2 点間の線形補完に置き換えることが出来、演算量を大幅に減らすことができること、生体分子に適用さ

れる様々な種類の力場の対応も table の差し替えのみで対応できるという事が挙げられる。その一方で、ランダムな配列アクセスが必要となり、アクセス待ちが増大する。

基本設計において lookup table によるエネルギー計算を廃止し、直接エネルギー計算を距離から求めることにより、アクセス待ちを減少させるアルゴリズムを試みた。SIMD 化は有効になり、演算性能は上昇した。しかし、ソフトウェアパイプラインはレジスタ不足のため無効のままであり、演算時間自体は 2.6 倍となった。

詳細設計（1）の後に、再度 lookup table のアクセス待ちの解消が試みられている。部分的に table を用いるアルゴリズムなどが提案されたが、クーロン相互作用と van der Waals 相互作用では、table が不可欠になる距離範囲が異なり、現時点では決定的な改変には至っていない。また、創薬計算で一般的に用いられている自由エネルギー摂動法 (FEP) では、更に非常に短い距離の部分に soft-core 関数を付け加える必要がある。検討の結果、table を利用したコードを利用することとなった。

4.2.6 長距離非結合相互作用の高速化

非結合相互作用のうち、長距離部分の区間では 3 次元 FFT を使用する。この区間は、複数回の FFT 計算、前処理、後処理、全体通信からなり、評価区間も分けられている。

- FFT 計算については、MD で利用される小規模の FFT カーネルの高速化を数値計算ライブラリ WG と提携しながら模索中である。
- 前処理、後処理の部分は SIMD が効く部分と効かない部分に分けるチューニングを行い、これら区間の演算時間が 7.5 割程度となった。
- 全体通信に関しては、通信 WG と提携しながら、高速化を試みている。特にノード内通信に関してはより高速な方式の採用をシステム側に要求した。
- GENESIS 開発者で全体通信の回数を減らしたアルゴリズムを導入した。
- 4.2.1 節に記した通り、数値積分スキームの改善を行い、長距離相互作用計算の頻度を 2 ステップに 1 回に減らした。
- 2017 年 5 月から FFT におけるノード間通信が発生しない様にすべく、1 ジョブを 1 ノードで実行するレイアウトに変更した。

4.2.7 PairList のメモリ減少について

PairList の演算時間を減少させるため 1 バイト整数でマスキング処理するアルゴリズムを採択した。その結果、Pairlist 部分の演算時間が 1.4 倍減少した。

4.2.8 Constraint のインダイレクトアクセス

分子内拘束 (区間名:constraint) におけるインダイレクトアクセスを削除し、演算速度を上げた。

4.2.9 コデザインによるシステムへの実装要請項目

本コデザインを通してシステムへフィードバックした実装要請・改善項目毎に整理して以下に示す。

- 浮動小数演算待ちが増大する問題に対して、コンパイラの修正
- 3 次元 FFT の全体通信において、ノード内通信の高速な方式の採用の要求
- Fortran のビット演算の高速化
- ソフトウェアパイプライン効率向上のためのコンパイラ修正

表 4.4: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (GENESIS)

実装要請課題	対応
分散 FFT 演算高速化のためのノード内通信性能改善	ノード内通信のみの MPI_Alltoall について、1 コピー方式を実装済み。実機上で性能評価を行う
高速な逐次 FFT 演算ライブラリ (実数/複素数、2/3/5/7 混合規定) の提供	富岳向けにチューニングした FFT 機能を開発する
浮動小数点演算待ち時間削減によるアプリケーション性能向上	コンパイラおよびアプリケーション双方のチューニングによって浮動小数点演算待ち時間を短縮
Fortran プログラムにおけるビット演算の高速化	LOGICAL 型データへビット情報をパッキングする手法提案。参考コードの提供済み
配列ポインタを用いるループに対する最適化機能強化	ポインタ配列に形状および contiguous 属性指定
ポインタ型を OpenMP private 指示節に指定した場合にエラーとして検出する機能	デバッグ用オプション -Nquickdbg=undef により検出できるようコンパイラを改良する
分散並列 FFT 高速化のための新インタフェイス設計	分散並列 FFT における入出力 2 軸分散機能を開発する

4.2.10 短距離相互作用計算でのプリフェッチ最適化

短距離相互作用区間では、メモリ／キャッシュのアクセス待ちの時間がボトルネックとなっており、その解消のために L1, L2 キャッシュの双方のプリフェッチの最適化を行った。具体的には、2 重ループの最内にプリフェッチ指示行を置き、内側のループインデックスの数により外側ループの次のインデックスの変数をプリフェッチするものである。コンパイル指示行である ocl prefetch_read をループインデックスにより if 文で制御するものとなり、より簡易に記述出来るようにシステムへの実装要請を行った。システムによる検討の結果、仕様が複雑になりすぎるため実装は見送られた。

4.2.11 ソフトウェア調整 (1) での長距離非結合相互作用の高速化

長距離非結合相互作用に対して、ソフトウェア調整 (1) でも引き続き以下の高速化を行った。

- 後処理部分の SIMD 最適化

- FFT3D のスレッド負荷バランスの改善

4.2.12　袖通信高速化

ノード間通信の高速化を目的に、袖通信に関して 4 方向同時に通信を行うように変更した。

4.2.13　ボトルネック部分外を含む全区間のでの高速化

詳細設計 (2) までは、ボトルネックである性能区間での高速化が中心であった。性能区間での性能調査、高速化が一通り完了したため、ボトルネック部分外の高速化に取り組んだ。特に、共有結合で結ばれている原子間の相互作用である結合性相互作用部分 (bond, angle, dihedral angle)、エネルギー出力の非結合性相互作用計算を取り上げた。これらのルーチンでは、スレッド間の負荷バランスが悪く、性能劣化につながることが分かった。そこで、openmp のループを dynamic 分割に変更した。また、SIMD の最適化を行った。もっとも効果の大きかった dihedral angle の計算で演算時間が半分程度となった。

4.2.14　富岳向け性能推定

性能電力予測ツールおよび性能シミュレータ等の環境を用いた富岳向け性能推定は第 4.1.5 節に説明される手法により実施した。

4.2.15　富岳で達成された性能倍率

上述したコデザインの成果を反映して富岳で達成された京に対する性能倍率及び消費電力を 4.5 に示す。

表 4.5: GENESIS の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
GENESIS	131 倍	22 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

表 4.6: GENESIS の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
GENESIS	131 倍		119 倍		126 倍		115 倍	
	22 MW	22 MW	20 MW	20 MW	16 MW	16 MW	14 MW	14 MW

4.3 重点課題 2 Genomon のコデザイン

4.3.1 アプリケーションの特徴とコデザインの方針

Genomon は、東京大学医科学研究所ヒトゲノム解析センター（東大医科研 HGC）で開発中の、がんのヒト全ゲノム関連データ解析プログラムである。入力データを加工して出力する小規模なプログラムをパイプライン的につなげることによりゲノム解析を行う、ワークフロー型のアプリケーションである。

Genomon に代表されるゲノム解析プログラムを用いた、京、富岳といった大規模スーパーコンピュータ上での高効率なアンサンブル計算を実現するためには、アプリケーションに共通する以下の特徴に留意する必要がある。

- パイプラインを構成する個々のプログラムの多くは、スレッド並列化されていない、あるいは、スレッド並列化不可能な処理を実行している。
- アプリケーション全体は、ノード間並列のレベルで見ると、並列度の異なる複数のパートから構成されている。
  - アライメント計算パートは、入力データの分割により任意ノード数での並列計算が可能であり、ロードバランスも良好である。
  - その他の計算パートは、アルゴリズム的に（主に生物学的な理由から）並列度が限定され、また、ロードバランスも悪い。
  - 1 ノード上での逐次計算が必要な部分の存在も無視できない。

このようなアプリケーションに対して、利用可能な計算機資源下で、与えられた時間内に何サンプルの解析が可能かというスループット性能を最大化することが求められている。

コデザインの観点から Genomon を見ると、文字列を対象とした整数演算が中心であること、パイプライン中にファイル I/O 律速なプログラムが存在すること、最大 TB クラスの入出力ファイル群に対して並列化されたパイプライン中の複数プロセスからの同時 I/O が発生する、などの特徴があげられる。これらの特徴により、Genomon は整数演算性能およびファイル I/O 性能の面からコデザインに寄与する。

また、アプリケーション全体を見ると、次のような複数の並列性の階層が存在するため、これらの個々の階層においてコデザインの検討・実施を行う必要がある。

- パイプラインを構成する個々のプログラム内の並列性
- 解析アルゴリズムに内在する並列性
- ワークフロー（パイプライン構成）に内在する並列性
- 複数検体（入力サンプル）同時解析としてのアンサンブル計算

基本設計、詳細設計（1）では、Genomon シリーズの内、がんの原因となる融合遺伝子というタイプの変異の検出を目的とした Genomon-fusion の京移植版である Genomon-fusion for K (GFK) を対象に、最適化および富岳性能推定を行った [55]。ただし、重点課題ターゲットアプリケーションとしての Genomon は、基本設計完了時点で、オリジナルアプリケーション開発者の要望により、GFK から Genomon2 へと変更されることが決定された。

Genomon2 は、エキソーム解析、全ゲノムシークエンス解析、トランスクリプトーム解析に対応した、がんゲノム解析アプリケーションであり、Genomon-fusion とは完全に別プログラムとして新規開発されたものである。富岳では、Genomon2 を用いた全ゲノムシークエンス解析を行うことを想定している。Genomon2 も、ゲノム解析アプリケーションとしての基本的な特徴は Genomon-fusion および GFK と同一であるが、以下のような相違点が認められる。

- Genomon2 では富岳ターゲット問題である全ゲノムシークエンス解析が可能。Genomon-fusion および GFK は RNA シークエンス解析専用。
- Genomon2 ではホットスポットであるアライメントプログラムが、Blat から、より高速な BWA に変更された。
- Genomon2 パイプラインのワークフロー制御およびジョブスケジューリングには Python で実装されたフレームワーク Ruffus[1] とグリッドエンジン（Sun Grid Engine/Univa Grid Engine）のアレイジョブ機能を用いている。Genomon-fusion ではシェルスクリプトとグリッドエンジンのアレイジョブとホールドジョブ機能を用いて、GFK ではシェルスクリプトとジョブ制御用 MPI プログラムを用いて、これらを実装していた。
- Genomon2 では、パイプラインを構成するプログラムのうち、オープンソースソフトウェア以外の独自開発プログラムは、Python により記述されている。Genomon-fusion および GFK では Perl が使用されていた。

なお、本報告書では、GFK と Genomon2 の両者を比較する際には、それぞれを Genomon v1, Genomon v2 などと記述する場合もある。

Genomon2 は、東大医科研 HGC のスーパーコンピュータシステム専用アプリケーションとして 2016 年 5 月にリリースされ、その後、京への移植作業が続けられたが、2016 年度末時点では、まだ京上での全ゲノムシークエンス解析における全ワークフローの完全な動作確認が完了していなかった。そのため、詳細設計（2）では、GFK を用いた富岳性能推定値のアップデート、Genomon2 に関しては、京への移植支援、ファイル I/O 量調査、および、ホットスポットとなることが確実なアライメントプログラム BWA のみを対象としたコデザインを実施した。

詳細設計（3）では、Genomon2 の京上での全ゲノムシークエンス解析が可能になったため、それをもとに富岳性能推定を実施した。また、2017 年 9 月に、塩基配列読み出し装置（シークエンサー）の将来における予想性能を見直し、富岳ターゲット問題の問題サイズを規定するパラメータに対して、塩基配列の断片データ（リード）の数およびリード内に含まれる塩基対数（リード長）の変更を行った。

変更前 リード数: 39 億リード、リード長: 200 塩基対

変更後 リード数: 14 億リード、リード長: 150 塩基対

これを反映して、GFK を用いた富岳性能推定値のアップデートも行なった。

詳細設計（4）では、Genomon2 のパイプライン構成プログラムレベル、および、ワークフローレベルでの高速化を実施し、それをもとに富岳性能推定値のアップデートを行なった。また、ファイル I/O 量調査を継続して実施し、その結果をもとに、SSD で構成される富岳第 1 階層ストレージを有効に活用するためのファイル配置の検討を行った。

[1] http://www.ruffus.org.uk/

ソフトウェア調整（1）では、富岳ターゲット問題実行に必要な規模での 150 塩基対リードデータの入手が当面困難であることが判明したため、富岳実機上でのターゲット問題実行性能は、これまでの性能推定時と同様に、100 塩基対リードデータに対する実行結果に対して各パイプライン構成プログラム実行時間ごとに換算係数を乗じて評価することとなった。ソフトウェア調整（1）中に入手できた 150 塩基対リードデータで実行可能な範囲のパイプライン構成プログラムについては、京上で 100 塩基対リードデータ使用時との実行時間の比較を行い、各換算係数が妥当な値であることを確認した。また、アライメントプログラム BWA の富岳への移植を行い、試作機上での実行性能、消費メモリ、消費電力の計測を実施した。BWA 以外についても、パイプライン構成プログラム単位で、試作機および ThunderX2 搭載機を用いて動作検証を行った。

ソフトウェア調整（2）から共用前評価環境 III にかけての期間では、富岳試作機および実機上での Genomon2 の動作確認、最適化を行い、実行性能、消費メモリ、消費電力の計測を実施した。

4.3.2 GFK を対象にしたコデザイン

ここでは、詳細設計（1）で実施した GFK に対するコデザインについてまとめる。

4.3.2.1 パイプライン構成プログラムに対する最適化

アプリケーション全体のホットスポットとなるアライメントプログラム Blat に対して、主に、スレッド並列におけるスケーラビリティの向上を目標に最適化を行った。具体的には、OpenMP ループ分割のダイナミックスケジューリングへの変更、逐次計算していた前処理部分の並列化、不要なロック/アンロック処理の削減を行った。この最適化により、京での 8 スレッド実行、FX100 での 16 スレッド実行において、充分なスケーラビリティが得られた。

4.3.2.2 解析アルゴリズムに内在する並列性の利用

GFK では、入力データである塩基配列断片データ群を参照ゲノム配列にアライメントした後、アライメント結果データへのランダムアクセスを可能とするために、それらに対して、アライメント位置によるソート、重複除去、インデックス付けを含んだ後処理を行っている。オリジナル Genomon-fusion では、この後処理計算を逐次実行（1 ノード 1 コア実行）していたため、ここが実行時間全体に対するボトルネックとなっていた。この後処理計算部分に対して、アライメント先の染色体毎の計算が可能であることに着目して、染色体単位でのノード間並列化を実施した。その結果、パイプライン構成プログラムレベルでの最適化と合わせて、京上での逐次実行では 7 時間弱かかっていた後処理計算が、26 ノード使用の並列計算により 21 分で可能となった。

4.3.2.3 ワークフローに内在する並列性の利用

ワークフローに内在する同時実行可能タスクとしての並列性を利用して、主にノード内の遊休コア削減を目的とした最適化を行った。具体的には、アライメント計算部において、入力リード対データへのパイプライン処理の 2 並列化、アライメント結果のフォーマット変換計算のコア単位並列化、融合遺伝子検出計算の一部に対して 3 並列化を実施した。

4.3.2.4　複数検体（入力サンプル）同時解析における並列性

基本設計段階での GFK は、使用ノード数の異なる 3 つの MPI プログラム GFKalign, GFKdedup, GFKdetect から構成されている。富岳では、これら 3 プログラムをひとつのパイプラインにまとめ（全体パイプライン）、複数検体の同時解析を行うことを想定して、その性能評価を行った（図 4.1）。

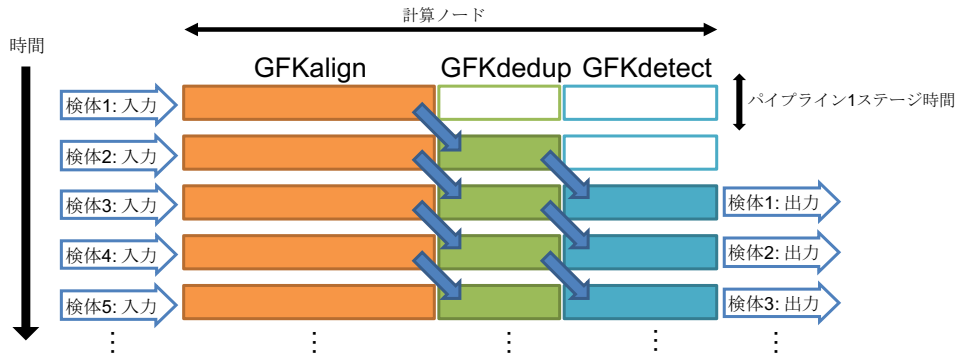


図 4.1: GFK ワークフロー全体のパイプライン実行

全体パイプライン実行における各部分の使用ノード数は、次のようにして決定される。GFKdedup と GFKdetect では、アルゴリズム的に MPI 並列におけるプロセス数が 26 と 44 に固定されているため、これにより使用ノード数が決まる。Genomon 全体パイプラインにおける 1 ステージ実行時間は、1 検体あたりにおける GFKdedup と GFKdetect の実行時間の大きい方で律速されると考えられる。一方、GFKalign は任意プロセス数での MPI 並列計算が可能なため、

$$\text{GFKalign 実行時間} < \max(\text{GFKdedup 実行時間}, \text{GFKdetect 実行時間}) \tag{4.1}$$

となるように GFKalign の使用ノード数を決めるものとする。これにより、GFKdedup と GFKdetect の実行時間の大きい方が、Genomon 全体パイプラインにおける 1 ステージ時間になる。1 本の全体パイプライン実行に必要なノード数は、(GFKalign 使用ノード数)+26 + 44 となる。

アプリケーション全体の実行性能は、全体パイプラインを複数本同時実行するものとして、以下のように定まるスループットにより評価した。

$$\text{単位時間あたりの解析検体数} = \frac{\text{パイプライン数}}{\text{パイプライン 1 ステージ時間}} \tag{4.2}$$

ここで、「パイプライン数」は利用可能な計算機資源下での同時実行可能な全体パイプラインの本数である。

また、ある検体入力データを全体パイプラインに投入してから、その解析結果が得られるまでの時間としての 1 検体解析時間は

$$\text{1 検体解析時間} = \text{パイプライン 1 ステージ時間} \times 3 \tag{4.3}$$

で与えられる。

4.3.3 Genomon2 におけるワークフロー制御とジョブスケジューリング

Genomon2 などのパイプライン型プログラムでは、アプリケーション全体の処理内容はタスクという単位に分割される。各タスク間には入力データと出力データをとおして実行順序の依存関係があり、各タスクを頂点とした有向非巡回グラフとしてワークフローが定義される。

4.3.3.1 Genomon2 におけるワークフロー制御

Genomon2 では、ワークフローの管理・制御に Python で実装された Ruffus フレームワークを使用している。このフレームワークを利用すると、アプリケーション開発者は、各タスクの処理内容を Python 関数として実装し、そのタスクの入力データの他タスクへの依存関係をデコレータ（関数を修飾する Python の機能）を用いて記述するのみでよく、以下にあげる機能は Ruffus 側で自動的に処理される。

1. タスク間の依存関係からワークフロー全体を構成
2. ワークフローに従い、適切な順番にタスクを実行
3. ワークフローレベルで並列化可能な部分を検出し、（可能なら）並列に実行
4. その他、ログ出力、エラー処理、チェックポイントなどの機能

Ruffus そのものは 1 ノード（マスタノード）上で実行される。上記 3 の並列処理もマスタノード内での並列実行である。

Genomon2 における DNA 解析パイププラインのワークフロー全体像を図 4.2 に示す。

4.3.3.2 Genomon2 における「ジョブ」スケジューリング

Genomon2 のワークフローを構成する主要なタスクでは、マスタノード上で処理を行わず、タスク内からジョブスケジューラ（オリジナル Genomon2 ではグリッドエンジン）を利用して、「ジョブ」[2]として別ノードに実行が委託される。この時、そのタスクが担当する処理が embarrassingly parallel 的に並列化可能な場合には、複数「ジョブ」をひとつのキューとして扱うアレイジョブとして投入している。具体的には、入力ゲノムシークエンスデータに対して分割処理が可能なアライメント計算や、染色体ごとに並列化が可能な処理などは、アレイジョブとして、このレベルで並列化がなされている。

Genomon2 の京への移植にあたり、Genomon2 開発者により、必要とされるグリッドエンジン機能と同等な機能を有する Virtual Grid Engine (VGE) の開発が続けられている。VGE は、マスター–ワーカ型の MPI プログラムとして、Python により実装されている。VGE を利用した Genomon2 の実行では、アプリケーション全体は通常の MPI プログラムと同様にひとつのジョブとして京の複数計算ノード上で実行され、VGE マスタプロセスは Ruffus と共にマスタノード上で、ワーカプロセスはその他のノード上で実行される。マスタプロセスは、ワークフロー中のタスクからの実行要求に従い「ジョブ」キューを管理し、空いたワーカへの「ジョブ」投入を行う。以後の節では、この VGE をとおして実行される「ジョブ」を VGE ジョブと記すことにする。

[2] ここでは、タスク内から他ノードに実行を委託する処理単位をカッコ付きの「ジョブ」と表現することにする。

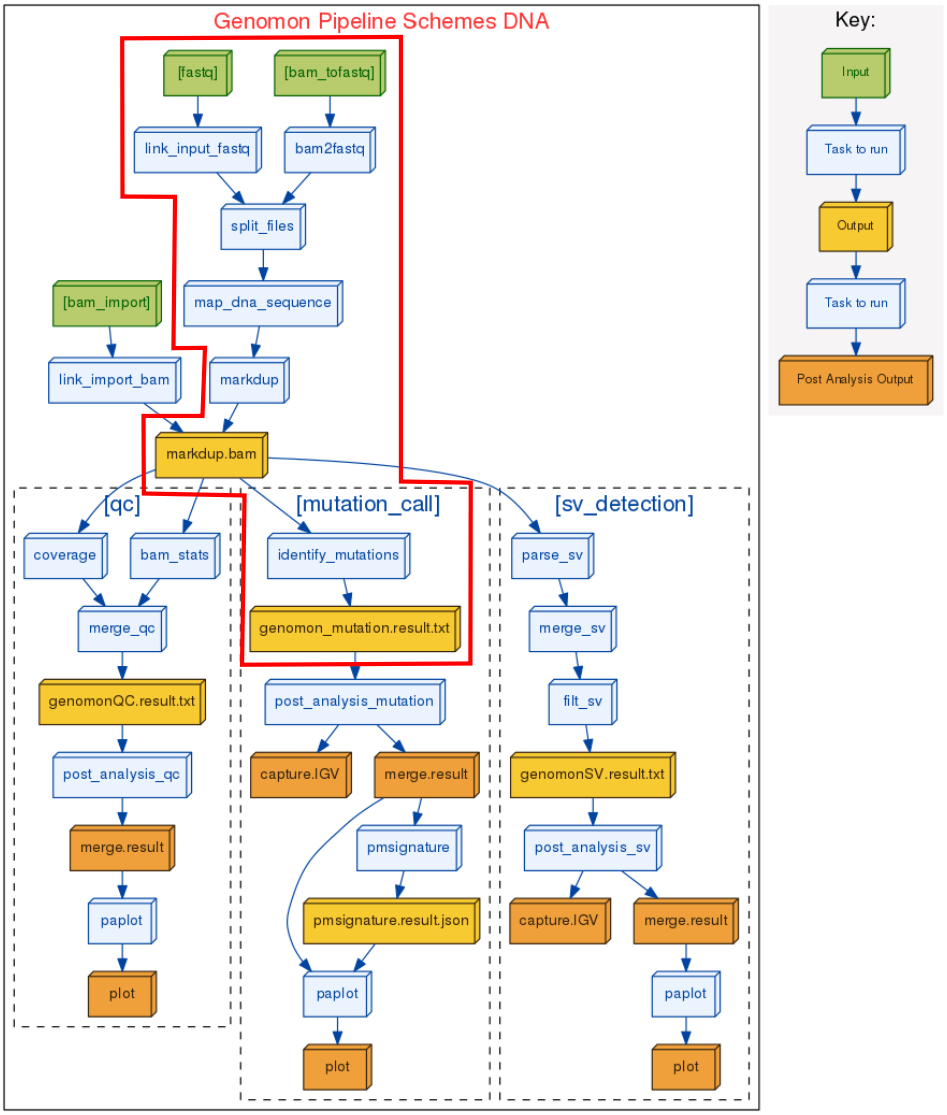


図 4.2: Genomon2 DNA 解析パイプラインのワークフロー（赤枠内が富岳ターゲット問題としての評価対象）

4.3.3.3　グリッドエンジン型ジョブスケジューラの提供

現在、Genomon をはじめとするバイオインフォマティクス分野での大規模解析アプリケーションの多くは、グリッドエンジンの使用を前提に並列化がなされている。そのため、今後富岳上でのこれらより多くのアプリケーションの実行を実現するためには、富岳のシステムジョブスケジューラ側において、グリッドエンジンと互換性のある機能とインタフェースを提供することが望まれる。その場合、次のような仕様が必要となるであろう。

- 計算ノード上のプロセスからジョブが投入可能（京ではログインノードのみから投入可能)
- ジョブの最小割当単位はノードまたは CMG（京では Tofu 単位、12 ノード）

- アレイジョブ内の同時実行可能ジョブ数の制限緩和（京バルクジョブ[3]では 15 ジョブ）

富岳において、このようなジョブスケジューリング機能の提供が可能かどうか、運用系ソフトウェア開発側とともに検討を要請した。ただし、Genomon2 においては、現在の京上での実行形態と同様に、従来型のシステムジョブスケジューラにより通常 MPI ジョブとして VGE を実行することにより対応することが決定された。

4.3.4 Genomon2 におけるファイル I/O 調査

Genomon2 により現在の問題規模でのヒト全ゲノム解析を京上で行った場合のファイル I/O 量、富岳ターゲット問題としてのヒト全ゲノム解析を富岳上で行った場合の推定ファイル I/O 量、それぞれについての調査を行なった。また、これらをもとに、SSD で構成される富岳の第 1 階層ストレージを有効活用するための、ファイル配置の検討を行った。

調査では、各パイプライン構成プログラムの実行前後における入出力ファイルのファイルサイズを記録するだけではなく、I/O プロファイラ Darshan[4]を用いて、パイプライン構成プログラム実行終了時には消去される一時ファイルのプログラム実行中の最大サイズ、各ファイル I/O のスループットなども計測した。この目的のために、MPI プログラム対応版 Darshan[5]に、さらに、I/O 実行関数呼び出し時のスタックトレース記録機能の追加、記録対象にストリーム系 I/O 関数群の追加、複数回 open〜close されるファイルについては close 時の最大ファイルサイズを記録する機能、multiprocessing モジュールにより並列化された Python プログラムの I/O 情報記録に対応、計測時のオーバヘッド削減、などの改良を施した。

4.3.4.1 Genomon2 入出力ファイルサイズ

表 4.7 に現在の問題規模の解析時、表 4.8 に富岳ターゲット問題の解析時の入出力ファイルの一覧を、そのファイルサイズとともに示す。両表中のタスク識別子とタスク名の関係は以下のとおりである。

Task1A: bam2fastq
Task2A: link_input_fastq
Task3A: split_files
Task4A: map_dna_sequence
Task5A: markdup
Task6A: identify_mutations
Task7A: merge_mutation
Task8A: post_markdup

ここで、図 4.2 中に無い merge_mutation は、並列度の違いを考慮して Genomo2 の京移植時に identify_mutations から分離したタスク、post_markdup はワークフロー改良により markdup の一部を分離したタスクである。

[3]京ジョブスケジューラにおけるアレイジョブ相当の機能。
[4]https://www.mcs.anl.gov/research/projects/darshan/
[5]http://www.sys.r-ccs.riken.jp/releasedsoftware/ksoftware/darshan/

表 4.7: Genomon2 入出力ファイル一覧（現在の問題規模に対する 1 検体解析）

ファイル番号	ファイル名	ファイル数	1個あたりのファイルサイズ	前処理	Task1A	Task3A	Task4A	Task5A	Task1B	Task6A	Task7A	備考
1	$INDIR/testdata/normalreference*.bam（シンボリックリンク先）	#list	ファイル29と同程度									
2	$INDIR/testdata/normalreference*.bam.bai（シンボリックリンク先）	#list	ファイル31と同程度									
3	GRCh37.fa	1	3GB					◇		◇		
4	GRCh37.fa.amb	1	100KB				◇					これら5ファイルはTask4Aのリファレンス用入力ファイルである。Task4Aでは5424(2332+3092)プロセスで動作し、これらの全プロセスから参照される。また、検体が異なる実行においても共通に利用される。（Genomon v2の実行においてユーザーIDは同じでジョブIDが異なる状況でアクセスされる）
5	GRCh37.fa.ann	1	10KB				◇					
6	GRCh37.fa.bwt	1	3GB				◇					
7	GRCh37.fa.pac	1	750MB				◇					
8	GRCh37.fa.sa	1	1.5GB				◇					
9	$database/simpleRepeat/simpleRepeat.bed	1	82MB							◇		これら4ファイルはTask6Aのリファレンス用入力ファイルである。現在のバージョンでは30プロセスで動作し、これらの全プロセスから参照される。また、検体が異なる実行においても共通に利用される。（Genomon v2の実行においてユーザIDは同じでジョブIDが異なる状況でアクセスされる）
10	$database/simpleRepeat/simpleRepeat.bed.bgz	1	26MB							◇		
11	$database/simpleRepeat/simpleRepeat.bed.bgz.tbi	1	1.5MB							◇		
12	$database/simpleRepeat/simpleRepeat.txt	1	86MB							◇		
13	$INDIR/sample.csv	1	KB	◇								
14	control_panel.txt	1	KB	◆								
15	$INDIR/testdata/${TYPE}.bam	2	106G (=(92G+120GB)/2)		◇							
16	temp.txt	2	17GB		★×							
17	sample_${TYPE}/*_1.fastq	2	158GB (=(180G+136G)/2)		◆	◇						
18	sample_${TYPE}/*_2.fastq	2	158GB (=(180G+136G)/2)		◆	◇						
19	single_end_output.txt	2	KB		◆							
20	unmatched_first_output.txt	2	KB		◆							
21	unmatched_second_output.txt	2	KB		◆							
22	sample_${TYPE}/1_*.fastq_split	5424 (=2332+3092 =2712×2)	58.3MB (=158GB/2712)			▲	◇×					
23	sample_${TYPE}/2_*.fastq_split	5424 (=2332+3092 =2712×2)	58.3MB (=158GB/2712)			▲	◇×					
24	fastq_line_num.txt	1	KB			◆	◇					
25	sample_${TYPE}_*_bwa.sam	5424 (=2332+3092 =2712×2)	116.5MB (=316GB/2712)				★×					
26	sample_${TYPE}_*.sorted.bam	1	106G				▲	◇×				
27	sample_${TYPE}_*.sorted.bam.bai	1	10MB～				▲	◇×				
28	sample_${TYPE}_*.sorted.bam.tmp	1	106GB				★×					
29	sample_${TYPE}_markdup.bam	2	106GB					◆		◇		bai/metrics/md5のインプットファイルでもある
30	sample_${TYPE}_markdup.metrics	2	KB					◆		◇		
31	sample_${TYPE}_markdup.bam.bai	2	10MB～					◆		◇		
32	sample_${TYPE}_markdup.bam.md5	2	KB					◆				
33	reference*/reference*.markdup.bam	#list							S(ファイル1)	◇		◇ ･･･入力ファイル ◆ ･･･Task7A終了後も保存される出力ファイル ★ ･･･ファイルの生成と削除を行うタスクが同じ場合の一時作業用出力ファイル ▲ ･･･ファイルの生成と削除を行うタスクが異なる場合の一時作業用出力ファイル × ･･･一時作業ファイルが削除されるタイミング S ･･･シンボリックリンクを作成(カッコ内はシンボリックリンク先を表す) TYPE･･･normalまたはtumor * ･･･ ファイルごとに挿入される一意の数字 #list･･･$INDIR/sample.csvの[controlpanel]に含まれるリストの数
34	reference*/reference*.markdup.bam.bai	#list							S(ファイル2)	◇		
35	$database/GRCh37/GRCh37_noScaffold_noDecoy.interval_list	1	KB							◇		
36	sample_tumor.fisher_mutations.*.txt	30	KB							★×		
37	sample_tumor.realignment_mutations.*.txt	30	KB							★×		
38	sample_tumor.indel_mutations.*.txt	30	KB							★×		
39	sample_tumor.breakpoint_mutations.*.txt	30	KB							★×		
40	sample_tumor.simplerepeat_mutations.*.txt	30	KB							★×		
41	sample_tumor.ebfilter_mutations.*.txt	30	KB							★×		
42	sample_tumor_mutations_candidate.*.hg19_multianno.txt	30	KB							▲	◇×	
43	sample_tumor_genomon_mutations.result.txt	1	KB								◆	

表 4.8: Genomon2 入出力ファイル一覧（富岳ターゲット問題に対する 1 検体解析）

ファイル番号	ファイル名	ファイル数	1個あたりのファイルサイズ 平均値 東大医科研様提供値	1個あたりのファイルサイズ 平均値 採用値(GB)	前処理	Task2A	Task3A	Task4A	Task5A stage1	Task5A stage2	Task1B	Task6A	Task8A	Task7A	備考
1	$INDIR/testdata/sample_${TYPE}1.fastq (シンボリックリンク先)	32	31.3GB(gz化で6～10GB) (=500GB/16)	1.0E+01											CA2-170822-R-4-problem-changed.pdf に記載されている「1件あたりのゲノム情報容量が1TB」とは、「normalの合計ファイルサイズとtumorの合計ファイルサイズの平均値(本シートではファイル番号1とファイル番号2の合計ファイルサイズの平均値(=(32×31.3GB+32×31.3GB)/2))が1TBであることと同義である。 平均ファイルサイズはgz化された場合の最大値を使用
2	$INDIR/testdata/sample_${TYPE}2.fastq (シンボリックリンク先)	32	31.3GB(gz化で6～10GB) (=500GB/16)	1.0E+01											
3	reference*/reference*.markdup.bam (シンボリックリンク先)	11	ファイル32と同程度	3.3E+02											100bp,6.78億の問題規模とファイル数は同じと仮定
4	reference*/reference*.markdup.bam.bai (シンボリックリンク先)	11	ファイル34と同程度	1.0E-02											
5	GRCh37.fa	1	3GB	3.0E+00	→	→	→	→	→	→	→	◇	→	→	Task6Aのリファレンス用入力ファイルであり275プロセス(=69ノード)が同一ファイルを参照する。
6	GRCh37.fa.fai	1	KB	1.0E-06	→	→	→	→	→	→	→	◇	→	→	平均ファイルサイズは下限値を使用
7	GRCh37.fa.amb	1	100KB	1.0E-04	→	→	→	◇	→	→	→	→	→	→	これら5ファイルはTask4Aのリファレンス用入力ファイルである。Task4Aでは11326プロセスで動作し、これらの全プロセスから参照される。また、検体が異なる実行においても共通に利用される。(Genomon v2の実行においてユーザーIDは同じでジョブIDが異なる状況でアクセスされる)
8	GRCh37.fa.ann	1	10KB	1.0E-05	→	→	→	◇	→	→	→	→	→	→	
9	GRCh37.fa.bwt	1	3GB	3.0E+00	→	→	→	◇	→	→	→	→	→	→	
10	GRCh37.fa.pac	1	750MB	7.5E-01	→	→	→	◇	→	→	→	→	→	→	
11	GRCh37.fa.sa	1	1.5GB	1.5E+00	→	→	→	◇	→	→	→	→	→	→	
12	$database/simpleRepeat/simpleRepeat.bed	1	82MB	8.2E-02	→	→	→	→	→	→	→	→	→	→	これら4ファイルはTask6Aのリファレンス用入力ファイルである。現在のバージョンでは275プロセスで動作し、これらの全プロセスから参照される。また、検体が異なる実行においても共通に利用される。(Genomon v2の実行においてユーザIDは同じでジョブIDが異なる状況でアクセスされる)
13	$database/simpleRepeat/simpleRepeat.bed.bgz	1	26MB	2.6E-02	→	→	→	→	→	→	→	◇	→	→	
14	$database/simpleRepeat/simpleRepeat.bed.bgz.tbi	1	1.5MB	1.5E-03	→	→	→	→	→	→	→	◇	→	→	
15	$database/simpleRepeat/simpleRepeat.txt	1	86MB	8.6E-02	→	→	→	→	→	→	→	→	→	→	
16	$INDIR/sample.csv	1	KB	1.0E-06	◇	→	→	→	→	→	→	◇	→	→	平均ファイルサイズは下限値を使用
17	control_panel.txt	1	KB	1.0E-06	◆	→	→	→	→	→	→	◇	→	→	平均ファイルサイズは下限値を使用
18	sample_${TYPE}/*_1.fastq	32	-	1.0E+01		S(ファイル1)	◇	→	→	→	→	→	→	→	
19	sample_${TYPE}/*_2.fastq	32	-	1.0E+01		S(ファイル2)	◇	→	→	→	→	→	→	→	
20	sample_${TYPE}/1_*.fastq_split	11326	88.3MB	8.8E-02			▲	◇×							25万リード、150bp(予備1bp)、リード名50字と仮定 ファイルサイズ=(151*2+50+1)*250,000=88.3MB 分割数=500GB/88.3MB=5663
21	sample_${TYPE}/2_*.fastq_split	11326	88.3MB	8.8E-02			▲	◇×							
22	fastq_split_num.txt	1	KB	1.0E-06			◆	◇	→	→	→	→	→	→	平均ファイルサイズは下限値を使用
23	sample_${TYPE}_*_bwa.sam	11326	215MB (=1TB×1.215/5663)	2.1E-01				★× 1							fastq->sam時の増分を100bp時と同等と仮定。 増加割合は253B*0.3/353B=0.215
24	sample_${TYPE}_*.chr*	985362	2.47MB (=215MB/87)	2.5E-03				▲	◇×						リファレンス中の染色体数(ウィルスも含む)が87種類 ヘッダファイル(1個)はカウントしていない。
25	sample_${TYPE}.chr*	174	14.0GB (=1TB×1.215/87)	1.4E+01					★× 1						fastq->sam時の増分を100bp時と同等と仮定。 増加割合は253B*0.3/353B=0.215
26	*_bamsort_000000	50	14.8GB×(353/253)	2.1E+01					★× 1						京の実測問題規模での最大値を使用(予定)
27	sample_${TYPE}.chr*.sorted.bam	174	4.66GB (=1TB×1.215/3/87)	4.7E+00					★× 1						sam->bam圧縮効果を1/3と仮定。
28	*_bamcollate	172	133MB×(353/253)	1.9E-01					★× 1						京の実測問題規模での最大値を使用(予定)
29	*_readpairs	171	419MB×(353/253)	5.8E-01					★× 1						
30	*_readfrags	172	771MB×(353/253)	1.1E+00					★× 1						
31	sample_${TYPE}.chr*.duplicated.bam	174	3.79GB (=330GB/87)	3.8E+00					▲ 1	◇×					bamファイルのファイルサイズ更新は未適用
32	sample_${TYPE}_markdup.bam	2	330GB	3.3E+02						◆	→	◇	◇	→	bai/metrics/md5のインプットファイルでもある bamファイルのファイルサイズ更新は未適用 同一VGEジョブ内で複数回読み込まれる
33	sample_${TYPE}_markdup.metrics	2	KB	1.0E-06						◆	→	◇	→	→	平均ファイルサイズは下限値を使用
34	sample_${TYPE}_markdup.bam.bai	2	10MB～	1.0E-02						◆	→	◇	→	→	平均ファイルサイズは下限値を使用
35	sample_${TYPE}_markdup.bam.md5	2	KB	1.0E-06									◆	→	平均ファイルサイズは下限値を使用
36	reference*/reference*.markdup.bam	11	ファイル32と同程度	3.3E+02	S(ファイル3)	→	→	→	→	→	→	◇	→	→	
37	reference*/reference*.markdup.bam.bai	11	ファイル34と同程度	1.0E-02	S(ファイル4)	→	→	→	→	→	→	◇	→	→	
38	$database/GRCh37/GRCh37_noScaffold_noDecoy.interval_list	1	KB	1.0E-06								◇	→	→	
39	sample_tumor.fisher_mutations.*.txt	275	KB	1.0E-06								★× 1	→		平均ファイルサイズは下限値を使用
40	sample_tumor.realignment_mutations.*.txt	275	KB	1.0E-06								★× 1	→		平均ファイルサイズは下限値を使用
41	sample_tumor.indel_mutations.*.txt	275	KB	1.0E-06								★× 1	→		平均ファイルサイズは下限値を使用
42	sample_tumor.breakpoint_mutations.*.txt	275	KB	1.0E-06								★× 1	→		平均ファイルサイズは下限値を使用
43	sample_tumor.simplerepeat_mutations.*.txt	275	KB	1.0E-06								★× 1	→		平均ファイルサイズは下限値を使用
44	sample_tumor.ebfilter_mutations.*.txt	275	KB	1.0E-06								★× 1	→		平均ファイルサイズは下限値を使用
45	sample_tumor_mutations_candidate.*.hg19_multianno.txt	275	KB	1.0E-06								▲	→	◇×	平均ファイルサイズは下限値を使用
46	sample_tumor_genomon_mutations.result.txt	1	KB	1.0E-06										◆	平均ファイルサイズは下限値を使用

◇ ・・・入力ファイル
◆ ・・・出力ファイル
★ ・・・一時ファイル(同一タスク内に閉じて生成と削除が行われる)
▲ ・・・一時ファイル(異なるタスクで生成と削除が行われる)
× ・・・一時ファイルが削除されるタイミング
S ・・・シンボリックリンクを作成(カッコ内はシンボリックリンク先を表す)
TYPE・・・normalまたはtumor
* ・・・ファイルごとに挿入される一意の数字を表す
#list・・・$INDIR/sample.csvの[controlpanel]に含まれるリストの数
→ ・・・各利用ツールまたはコマンド実行では使用しないが、継続して存在するファイル

4.3.4.2 第 1 階層ストレージの有効活用

富岳では、第 1 階層ストレージとしての SSD を、以下の用途ごとの領域に分けて使用可能である。

(1) ノード内テンポラリ
(2) ノード間共有テンポラリ
(3) 第 2 階層ストレージキャッシュ(SSD 間でのファイル複製あり)
(4) 第 2 階層ストレージキャッシュ(SSD 間でのファイル複製なし)

Genomon2 で使用されるファイルは、入力ファイル、出力ファイル、一時ファイルに分類され、それぞれ、以下のように第 1 階層ストレージに配置されるのが理想である。

- 入力ファイル (Genomon2 実行開始時に必要なファイル)

  単一ノード上のパイプライン構成プログラムからのみアクセス → (4)

  それ以外 → (3)

- 出力ファイル (Genomon2 実行終了後に回収が必要なファイル) → (4)

- 一時ファイル

  単一のパイプライン構成プログラム内での一時ファイル → (1)

  単一の VGE タスク内での一時ファイル → (1)

  複数の VGE タスクにまたがり使用される一時ファイル → (2)

表 4.9 に、富岳ターゲット問題解析時において、第 1 階層ストレージへの理想的なファイル配置を行った場合の必要ストレージ量の見積もりを示す。ここでは、第 2 階層ストレージへのキャッシュとして、SSD 間での複製を配布をした場合、しない場合、2 種類の見積もりを行っている。

表 4.9: 第 1 階層ストレージへの理想的なファイル配置時のストレージ消費量

タスク名		2A	3A	4A	5A stage1	5A stage2	6A	8A	7A	最大必要ストレージ量
1検体あたりの必要ストレージ量(GB)										
ノード内テンポラリ領域※1		0.0	0.0	82.2	4,272.5	0.0	0.0	0.0	0.0	4,272.5
共有テンポラリ領域		0.0	2,000.2	2,430.0	2,430.0	660.0	0.0	0.0	0.0	2,430.0
第2階層ストレージのキャッシュ※2	複製を配布	50.7	50.7	50.7	50.7	710.7	710.7	710.7	710.7	710.7
	複製なし	8.4	8.4	8.4	8.4	668.5	668.5	668.5	668.5	668.5
合計	複製を配布	50.7	2,050.8	2,562.8	6,753.2	1,370.7	710.7	710.7	710.7	7,413.2
	複製なし	8.4	2,008.6	2,520.6	6,710.9	1,328.5	668.5	668.5	668.5	7,371.0
ノードあたりの必要ストレージ量(GB)										
ノード内テンポラリ領域※3		0.0	0.0	0.9	157.1	0.0	0.0	0.0	0.0	157.1
共有テンポラリ領域		0.0	20.8	25.3	25.3	6.9	0.0	0.0	0.0	25.3
第2階層ストレージのキャッシュ	複製を配布	0.5	0.5	0.5	0.5	7.4	7.4	7.4	7.4	7.4
	複製なし	0.1	0.1	0.1	0.1	7.0	7.0	7.0	7.0	7.0
合計	複製を配布	0.5	21.4	26.7	182.9	14.3	7.4	7.4	7.4	189.8
	複製なし	0.1	20.9	26.3	182.5	13.8	7.0	7.0	7.0	189.4

※1 ストレージの再利用を考慮
※2 出力ファイルについては1回あたりの書き込みサイズ分のストレージ量が必要。現状はデータが揃っていないため、出力ファイルサイズ分を1回で書き込むことを仮定
※3 ストレージの再利用を考慮し、かつノードあたり4VGEジョブの同時実行を仮定

富岳ターゲット問題解析時には、1 検体解析に 96 ノードを用いるため、全第 1 階層ストレージ量は 9,600GB、ノードあたりでは 100GB が使用可能である。理想的なファイル配置では、全必要

ストレージ量は上限値 9,600GB 以内だが、Task5A_stage1(Task5A の並列実行部分) でのノード内テンポラリ領域使用量が 157.1GB となっているのが原因で、ノードあたりでは上限値 100GB を超過してしまっている。

そこで、Task5A_stage1 の全 VGE ジョブ数は 174 であり、ノードあたり 4VGE ジョブを同時実行すると仮定すると、44 ノードしか使用しないことを利用して、一部の一時ファイルの配置先をノード内テンポラリ領域から共有テンポラリ領域に移すことによる、ノードあたりのストレージ消費量の削減を目指した。表 4.10 に、Task5A_stage1 の一時ファイルの内、ストレージ不足を補えるサイズであり、かつ現問題規模での京での実行時の I/O 時間が短かったいくつかのファイルの配置をノード内テンポラリ領域から共有テンポラリ領域に移動した場合の必要ストレージ量の見積もりを示す。この変更後のファイル配置では、第 2 階層ストレージへのキャッシュを SSD 間で複製する場合でも、ノードあたりの必要ストレージ量は 88.6GB となり、上限値 100GB 以内に収まった。

表 4.10: 第 1 階層ストレージ消費量 (ファイル配置変更後)

タスク名		2A	3A	4A	5A stage1	5A stage2	6A	8A	7A	最大必要ストレージ量
1検体あたりの必要ストレージ量(GB)										
ノード内テンポラリ領域※1		0.0	0.0	82.2	2,430.0	0.0	0.0	0.0	0.0	2,430.0
共有テンポラリ領域		0.0	2,000.2	2,430.0	2,430.0	660.0	0.0	0.0	0.0	2,430.0
第2階層ストレージのキャッシュ※2	複製を配布	50.7	50.7	50.7	50.7	710.7	710.7	710.7	710.7	710.7
	複製なし	8.4	8.4	8.4	8.4	668.5	668.5	668.5	668.5	668.5
合計	複製を配布	50.7	2,050.8	2,562.8	2,562.8	1,370.7	710.7	710.7	710.7	5,570.7
	複製なし	8.4	2,008.6	2,520.6	2,520.6	1,328.5	668.5	668.5	668.5	5,528.5
ノードあたりの必要ストレージ量(GB)										
ノード内テンポラリ領域※3		0.0	0.0	0.9	55.9	0.0	0.0	0.0	0.0	55.9
共有テンポラリ領域		0.0	20.8	25.3	25.3	6.9	0.0	0.0	0.0	25.3
第2階層ストレージのキャッシュ	複製を配布	0.5	0.5	0.5	0.5	7.4	7.4	7.4	7.4	7.4
	複製なし	0.1	0.1	0.1	0.1	7.0	7.0	7.0	7.0	7.0
合計	複製を配布	0.5	21.4	26.7	26.7	14.3	7.4	7.4	7.4	88.6
	複製なし	0.1	20.9	26.3	26.3	13.8	7.0	7.0	7.0	88.2

※1 ストレージの再利用を考慮
※2 出力ファイルについては1回あたりの書き込みサイズ分のストレージ量が必要。現状はデータが揃っていないため、出力ファイルサイズ分を1回で書き込むことを仮定
※3 ストレージの再利用を考慮し、かつノードあたり4VGEジョブの同時実行を仮定

4.3.5 アライメントプログラム BWA

詳細設計（2）では、Genomon2 の京上での全ゲノムシークエンス解析における全ワークフローの完全な動作確認が完了していなかったため、先行して、Genomon2 パイプライン全体に対するホットスポットとなることが確実なアライメントプログラム BWA のみを対象としたコデザインを実施した。BWA は C 言語で記述されたオープンソースソフトウェアである。Genomon2 では、BWA の提供する機能のうち、Maximal Exact Matches (MEM) アルゴリズムを使用した BWA-MEM コマンドにより、ゲノムシークエンスの断片データ（リード）の参照ゲノム配列へのアライメント（リードデータの参照ゲノム配列上での位置の同定）を行っている。

詳細設計（3）では、BWA 実行時の消費メモリ量に関する調査を行い、入力データの分割方法にラウンドロビン分割に導入することにより、富岳上の 1 ノードで 4BWA 実行が可能な程度まで消費メモリ量を削減できることを明らかにした。

ソフトウェア調整（1）では、オリジナル BWA において Intel SSE2 組み込み関数により SIMD 化されていた箇所を SVE に移植し、富岳試作機上での実行性能、消費メモリ量、消費電力の測定を実施した。

4.3.5.1 ファイル一括読み込み時の遅延

京上で BWA-MEM を実行する際、共有ディレクトリ上の約 750MB の参照ゲノム配列データファイルの fread 関数による一括読み込みに 30 分以上かかった。京のファイルシステムのページキャッシュサイズが 128MB に設定されており、ファイル読み込みを 16MB 単位で行うようにプログラムを修正すると、遅延は発生せず、数秒で参照ゲノム配列データファイルの読み込みが完了するようになった。

このように、現在は、ファイル読み込み時の遅延を避けるには、システム利用者側にプログラム修正レベルでの対処が要求されている。Genomon に代表されるゲノム解析アプリケーションでは、オープンソースソフトウェアなど他者が開発したプログラムを複数組みあせて利用するケースが非常に多い。富岳を利用すると想定される一般的なバイオインフォマティクス研究者にとっては、それらのプログラムはできればブラックボックスとして扱えることが望まれるであろう。そのため、富岳においては、ファイル遅延現象はシステム側で解決策を講じるよう要請した。

4.3.5.2 BWA の SIMD 演算

BWA-MEM では、Smith-Waterman 法によるアライメント計算をする関数 ksw_u8 がホットスポットのひとつになっている。オリジナル BWA-MEM では、ksw_u8 に対して、Intel SSE2 組み込み関数 (intrinsic) を用いた SIMD 化（8 ビット整数に対する 16SIMD）がなされている。SSE2 組み込み関数がハードコードされているため、オリジナル BWA-MEM は京などの非 Intel 環境上では実行不可能である。そのため、Genomon2 開発者による京移植時に、SSE2 組み込み関数相当の関数を C 言語により実装する対応がとられた。この関数を「移植版 ksw_u8」と呼ぶことにする。移植版 ksw_u8 では、SSE2 組み込み関数相当関数の呼び出しや、SIMD 化によって生じた本来の Smith-Waterman アルゴリズムには必要のない処理などのオーバヘッドが顕著であった。そこで、関数 ksw_u8 の機能を Smith-Waterman 法をもとに素直に、最内ループを 32 ビット整数に対する演算として、実装した「非 SIMD 版 ksw_i32」関数を開発した。非 SIMD 版 ksw_i32 を組み込んだ BWA-MEM に、ファイル読み込み時の遅延対策、後述する OpenMP 化を施したコードによる京上での実行性能を、今後の富岳性能推定におけるベースライン性能とするものとした。

ksw_u8 に対応する関数として、FX100 上の HPC-ACE2 用 SIMD 組み込み関数を用いて、「4SIMD 版 ksw_i32」関数を実装した。詳細設計（２）における BWA の富岳性能推定では、この 4SIMD 版 ksw_i32 を組み込んだ BWA-MEM を対象とした。また、京上の HPC-ACE 用 SIMD 組み込み関数を用いて「2SIMD 版 ksw_r64」関数も実装した。図 4.3、図 4.4 に、リード長 100 塩基対 × 250,000 リードのテストデータに対する、これらの実行時間を示す。ここで、KT_FOR1 は BWA-MEM の I/O を含まないアライメント計算メインループ中の ksw_u8/ksw_i32/ksw_r64 を含まないパート、KT_FOR2 は SIMD 演算対象となるそれらを含むパートの実行時間である。「全体」は、I/O を含めた BWA-MEM 全体の実行時間である。

4.3.5.3 BWA の OpenMP スレッド並列化

ベースとしたオリジナル BWA-MEM（バージョン bwa-0.7.15）では Pthread を用いたスレッド並列化がなされている。性能電力予測ツールにより富岳性能推定をするためには OpenMP によりスレッド並列化されている必要があるため、プログラムを改良して OpenMP 化を実施した。ただし、オリジナル BWA-MEM にあった I/O スレッドを別に生成し、アライメント計算と I/O をオーバラップして行う機能は OpenMP 化版には実装していない。

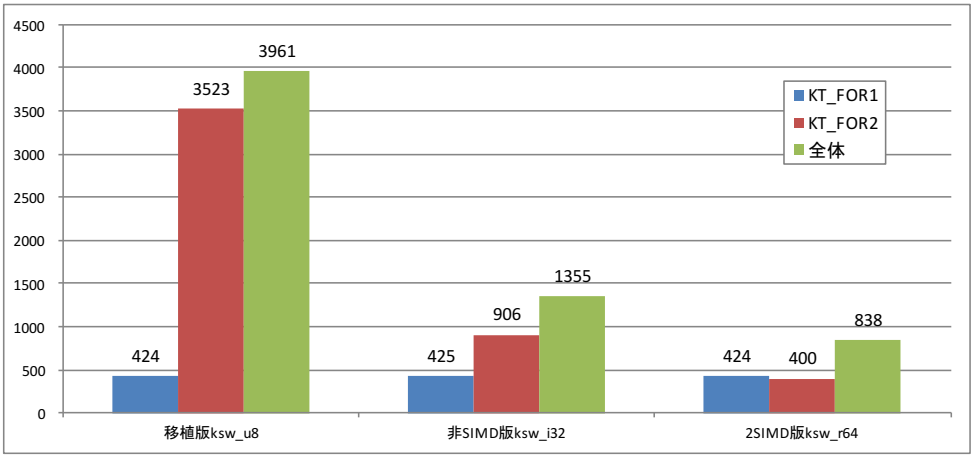


図 4.3: 京、1 スレッドでの BWA-MEM 実行時間（秒）

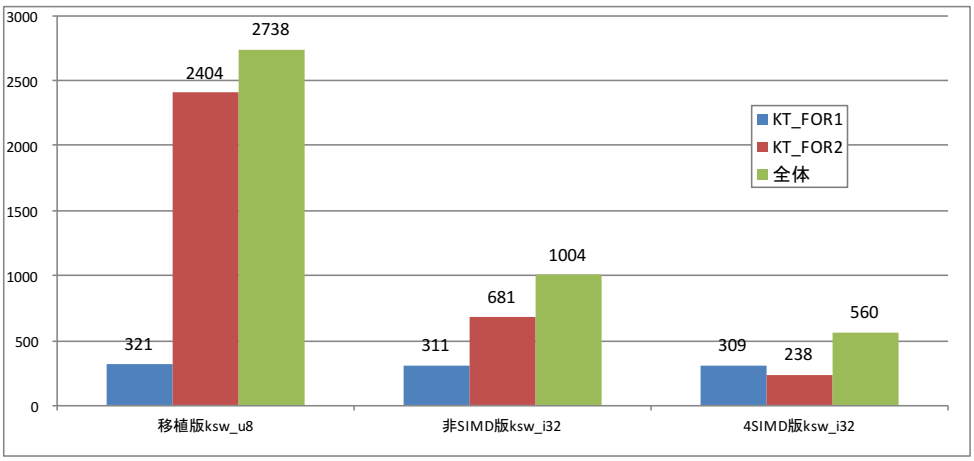


図 4.4: FX100、1 スレッドでの BWA-MEM 実行時間（秒）

リード長 100 塩基対 × 250,000 リードのテストデータに対して、Pthread 版ではアライメント計算と I/O のオーバラップを行わないという条件の元で、京で 8 スレッド、FX100 で 16 スレッドまで、OpenMP 版は Pthread 版と同等な計算性能を持つことを確認した。

4.3.5.4　ラウンドロビン分割の導入による BWA 消費メモリ量の削減

Genomon2 の性能評価においては、現在入手可能なリード長 100 塩基対のサンプルデータを分割して、1 分割ファイルあたり 250,000 リードデータを入力として BWA の計算を実行している。しかし、京での 8 スレッド実行時に、ノードあたり消費メモリ量が 9GiB を超える分割ファイルが多数存在した。富岳において 1CMG あたりの利用可能メモリ量の目安は 7GiB と考えられる。富岳ターゲット問題では入力データのリード長は 150 塩基対に増加され、また BWA は 1CMG 上で 12 スレッド実行を想定しているため、さらに消費メモリ量が増加する可能性があり、1CMG 上での BWA 実行が不可能になる恐れがあった。

このため、まず BWA のプログラム内部を解析し、その計算手順と消費メモリ量内訳の調査を行った。BWA（正確にはペアエンドリードに対する BWA MEM）では、次のように 2 段階の計算によりリードデータのマップ（アライメント）先となる塩基位置を同定している。

マッピング 1 各入力リードデータに対して、インデックスデータ (suffix array) を参照して起点となる塩基位置（seed）を求め、そこからアライメント計算を開始

マッピング 2 ペアとなるリード対において、片方のみ「マッピング 1」でマッピングできなかったリードに対し、「マッピング 1」でのマッピング時の統計情報をもとにマップ先の塩基位置範囲を限定し、再度アライメント計算を実施

現在の計算条件では、分割入力ファイル内の全リードデータに対して「マッピング 1」を実施し、その結果に対して統計解析を行った後、それをもとに「マッピング 2」に移っている。そのため、「マッピング 1」終了時に、統計計算に必要な入力リード全てに対するマッピング情報がメモリ上に保持されており、それが消費メモリ量の大きな部分を占めていることが判明した。統計解析に必要な情報には、「マッピング 1」における最終的なマッピング結果の他に、マップ先候補となる塩基位置が複数あった場合には、その個々のマップ候補先についての情報も含まれる。この「マッピング 1」におけるマップ先候補数については、最終的なマッピング結果との間に相関があり、デコイ配列 hs37d5 にマップされるリードデータについては顕著に多くなる（全リードデータ平均の 8 倍程度）ことも明らかになった。

これまで実施していた Genomon2 の性能評価では、既にマップ先の塩基位置によりソートされたリードデータを分割前の入力ファイルとして用いていた。そのため、リードデータファイルを先頭から 250,000 リードずつ分割していくと、デコイ配列 hs37d5 のみにマップされる分割ファイルが複数個存在していた。これらの分割ファイルを入力ファイルとする BWA 計算において、顕著に消費メモリ量が増加して、9GiB を超えていることも確認できた。そこで、BWA 実行時の消費メモリ量削減を目的として、リードデータの分割方法をリードごとに分割先ファイルを巡回的に割り振るラウンドロビン分割に改め、ひとつの分割ファイル内での「マッピング 1」におけるマップ先候補数を平均化を行った。このラウンドロビン分割の導入により、消費メモリ量の分割入力データに渡る最大値を、従来の 9.18GiB から 6.08GiB まで削減できた。

また、「マッピング 1」におけるマップ先塩基位置の候補数は、リード長が現在の 100 塩基対から富岳ターゲット問題の 150 塩基対に増加すると、逆に減少することが予想され、BWA 実行時の消費メモリ量も減少する可能性が期待される。そこで、疑似リードデータ作成プログラム wgsim[6] により、リード長 100 塩基対と 150 塩基対、それぞれの疑似リードデータを作成して、調査を実施した。その結果、疑似リードデータからラウンドロビン分割により得られた分割入力ファイルに対して、「マッピング 1」におけるマップ先塩基位置の候補数が、リード長 100 塩基対での 9.7 からリード長 150 塩基対では 7.6 に減少することが確認できた。このリード長 150 塩基対の疑似リードデータに対する BWA 実行時の消費メモリ量は、京での 1 ノード 8 スレッド実行時で 6.20GiB、FX100 での 1CMG12 スレッド実行時で 6.33GiB となった。

4.3.5.5　BWA の SVE 移植

4.3.5.2 節で述べたように、オリジナル BWA ではホットスポットのひとつである Smith-Waterman 法によりアライメント計算をする関数 ksw_u8 を Intel SSE2 の組み込み関数を用いて SIMD 化している。ソフトウェア調整（1）では、その部分を A64FX に実装された SIMD 拡張である SVE の組み込み関数で書き換える事により移植した [53]。オリジナル BWA では、SSE2 レジスタ長が 128bit であることから、8bit 符号なし整数変数に対する 16SIMD となっている。そこで移植版コードにおいても、512bit 長の SVE レジスタをプレディケートレジスタにより 128bit にマスクする

[6] https://github.com/lh3/wgsim

ことにより、8bit 符号なし整数変数に対する 16SIMD とした。また別途、プレディケートレジスタのマスク幅を 256bit に広げることにより、オリジナル BWA にはない 32SIMD 版のコードも作成した。

富岳の OS である Linux のカーネルでは、prctl システムコールにより、ベクトルレジスタ長を 128bit 単位で設定変更できる機能を提供している。そこで富岳試作機上での性能評価では、移植した BWA をそのまま実行する他に、実行時にベクトルレジスタ長を有効な SIMD 幅に合わせて、16SIMD 版コードでは 128bit に、32SIMD 版コードでは 256bit に縮小した状況下での実行も行った。

図 4.5 に、SIMD 化部分 (4.3.5.2 節での KT_FOR2) の実行時間を示す。「VL 縮小」とあるのが実行時にベクトルレジスタ長を縮小したものである。比較のために FX100(CPU 周波数 1.975GHz) 上での 3 ケースの実行時間も記載した。それぞれ、4.3.5.2 節の、「NO SIMD」は非 SIMD 版 ksw_i32、「HPC-ACE(2SIMD)」は 2SIMD 版 ksw_r64、「HPC-ACE2(4SIMD)」は 4SIMD 版 ksw_i32 に対応する。コア性能を比較するために、富岳試作機においても FX100 と同様に 8 コアで実行している。これら図から分かるとおり、実行時にベクトルレジスタ長を有効 SIMD 幅に縮小することにより、2 倍程度の性能向上が得られた。また、SIMD 並列度を 32 に増やすことは、100 塩基長リードデータでは効果が無いが、150 塩基長リードデータには有効であることが分かった。

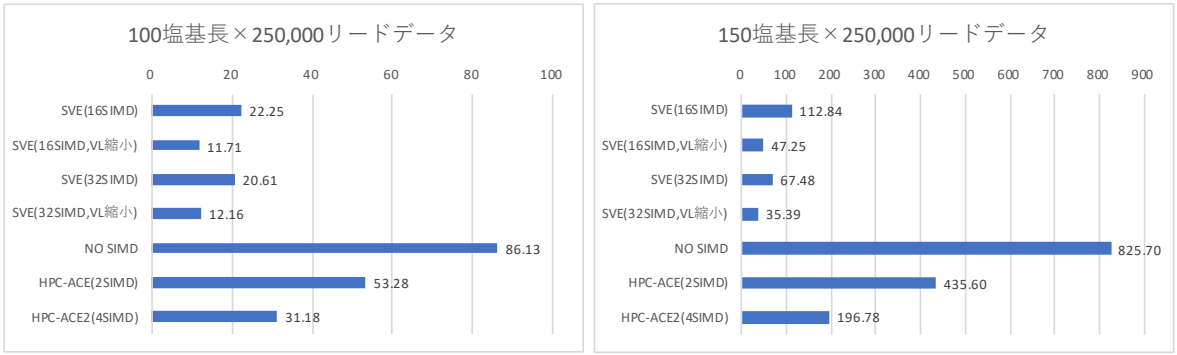


図 4.5: BWA SIMD 化部分（KT_FOR2 区間）実行時間（秒）

「SVE*」は富岳試作機、「NO SIMD」「HPC-ACE*] は FX100(1.975GHz) 上で実行。全て 8 スレッド実行。

図 4.6 に BWA のアライメント計算部分の実行時間を示す。これらは、BWA の全実行時間からファイル I/O 時間を除いた時間にほぼ等しい。

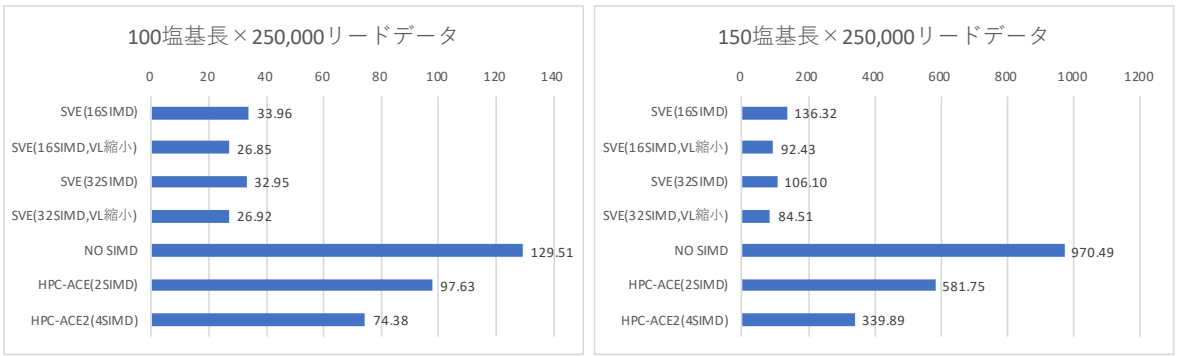


図 4.6: BWA アライメント計算部分 (KT_FOR1 + KT_FOR2 区間) 実行時間（秒）

「SVE*」は富岳試作機上で 12 スレッド、「NO SIMD」「HPC-ACE*] は FX100(1.975GHz) 上で 8 スレッド実行。

BWA で使用されている SIMD 化アルゴリズムでは、SIMD 化部分の最内ループの回転数が入力

データのリード長で決まってしまう。実際の最内ループの回転数は、100 塩基対データに対しては 16SIMD で 7、32SIMD で 4、150 塩基対データに対しては 16SIMD で 10、32SIMD で 5 となる。そのため、SVE のベクトルレジスタ長 512bit をさらに活用して SIMD 並列度を上げる改良は容易だが、現在入手可能なリードデータでは最内ループの回転数不足により性能向上は見込めない。

4.3.6 変異検出・検証・フィルタリング計算に対する最適化

遺伝子変異の検出および検出結果の統計検証とフィルタリングを行う Task6A「identify_mutations」は、連続して実行される以下の 6 つの Python スクリプトからなる。

- fisher comparison
- mutfilter realignment
- mutfilter indel
- mutfilter breakpoint
- mutfilter simplerepeat
- EBFilter

このタスクは、Genomon2 の京移植当時のプログラムでは、塩基座標上に分布している 30 の変異検出対象領域に対応して、30 並列の VGE ジョブとして実行され、京において 18 時間程度の計算時間を必要とした。詳細設計（3）では、これに対して以下の最適化を実施した。

1. 最も実行時間内訳の大きかった EBFilter のノード内並列化
2. EBFilter に次いで実行時間内訳の大きかった fisher comparison のノード内並列化
3. 入力サンプルデータに、本来の解析対象である遺伝子変異とは無関係な異常が見られたため、depth 値（ある位置にマッピングされたリード数）が 1000 を超えた塩基位置については、解析をスキップ
4. 変異検出対象領域を見直し、それをさらに細分化し、タスク全体を 275 並列の VGE ジョブに変更

これらの最適化により、18 時間程度かかっていた Task6A 全体の計算が 4.5 時間程度で実行可能となった。

詳細設計（4）では、さらに以下の最適化を実施した。

1. realignment に対して、ノード内並列化、外部プログラム Blat 呼び出し回数の削減、depth 値計算の回避、その他の Python コードチューニング
2. indel では、Darshan による調査により、アライメント結果を格納した BAM ファイルにランダムアクセスするために必要な BAM インデックスファイルの読み込みが（ファイルサイズは 9MB と小さいが、延べ約 5,000 回の全ファイル読み込みがあるため）ボトルネックとなっていることが判明した。Task6A は、リファレンス配列群をさらに細分化した変異検出対象領域単位で並列化されている。そこで、予めリファレンス配列ごとに、BAM インデックスファイルの該当リファレンス配列外のインデックスを 0 クリアした「分割 BAM インデックスファイル」を作成しておき、これを読み込むように改良し、ファイル読み込み時間を削減した。

これらの最適化により、京での現問題に対する Task6A 全体の実行時間をさらに 2.2 時間程度まで短縮できた。

4.3.7 ファイルシステムの負荷削減

Genomon2 では、パイプライン実行が Task4A「map_dna_sequence」から Task5A「markdup」に移る時点で、並列化の軸が入力リードデータファイルの分割から主に染色体を単位としたリファレンス配列を基準とした分割に変わるため、MPI_Alltoall 通信に類似したデータ交換をファイルシステムの共用領域上で行っている。詳細設計（3）終了時のコードでは、京上での現在の問題規模の解析実行において、このタイミングで 1 つのディレクトリ上に約 47 万個のファイルが存在していた。この事が原因でファイルシステムが過負荷となり、ある程度の頻度で、ノードダウンやディレクトリエントリ取得の失敗によるプログラム異常終了が発生していた。

そこで、詳細設計（4）では以下の 2 点の改良を行い、ファイルシステムの負荷を削減し、Genomon2 パイプライン実行の安定性を向上させた。

サブディレクトリの導入　Task4A「map_dna_sequence」から Task5A「markdup」へ受け渡すデータファイルを格納する作業ディレクトリを、サブディレクトリを導入して細分化した。これにより、この時点でのディレクトリあたりのファイル数の最大値を約 3,100 まで削減した。

ディレクトリエントリ取得手続きの削除　Task5A「markdup」では、ファイル結合操作および不要となった一時ファイルの削除操作において、対象となるファイルのリストをシェルのグロブ展開により取得していた。ファイルシステム過負荷時には、稀に、このグロブ展開の結果にファイルの重複や欠損などが発生していた。そこで、結合操作や削除操作の対象となるファイルは予め分かっていたので、グロブ展開などのディレクトリエントリ取得手続き使用しない実装に改めた。

4.3.8 アライメント後処理計算の並列化

Task5A「markdup」では、アライメント結果に対する後処理を行っている。このタスクは、VGE レベルで見ると、2 ステージの VGE ジョブ投入からなる。

Task5A_stage1 リファレンス配列単位で並列化され、tumor と normal の 2 系統の入力に対して、それぞれ 87VGE ジョブとして実行される。各 VGE ジョブでは、アライメント結果ファイルを結合した後、その内容をリファレンス配列内の塩基位置でソートし、BAM ファイルとして出力する。その後、BAM ファイルに対して、重複リードデータのマーク付けを行う。

Task5A_stage2 tumor と normal の 2 系統の入力それぞれで 1VGE ジョブにより逐次実行される。stage1 で得られた BAM ファイルを結合した後、ランダムアクセスを可能にするためのインデックス（BAM インデックスファイル）を作成する。また、結合された BAM ファイルの MD5 チェックサムも計算する。

詳細設計（3）終了時では、京での現問題に対する計算において、逐次実行される Task5A_sgage2 に 2 時間以上かかり、Genomon2 パイプライン実行全体に対するボトルネックとなっていた。

詳細設計（4）において、BAM ファイルおよび BAM インデックスファイルの仕様[7]を検討したところ、リファレンス配列単位で分割された BAM ファイルに対して BAM インデックスファイ

[7] https://samtools.github.io/hts-specs/SAMv1.pdf

ルを作成した後、それらを結合することにより、結合 BAM ファイルに対するインデックスファイルを作成可能である事が判明した。そこで、インデックスファイルの作成も Task5A_stage1 においてリファレンス配列単位で並列に行い、Task5A_stage2 において、新たに開発した専用プログラムによりそれらを結合するように変更した。この修正と、BAM ファイル自身の結合プログラムの高速化を合わせて、Task5A_stage2 の実行時間を 40 分程度まで短縮できた。

4.3.9 ワークフローレベルでの高速化

詳細設計（4）では、ワークフローレベルでの高速化も実施した。図 4.7 に、詳細設計（3）終了時点での Ruffus タスクレベルで見た Genomon2 パイプラインのワークフローを示す。ワークフローは基本的には図 4.2 赤枠内と同様であるが、ここでは、Task1A「bam2fastq」から Task5A「markdup」までの処理が 2 系統の入力データ tumor と normal に対して並列に行われることを明示的に記している。

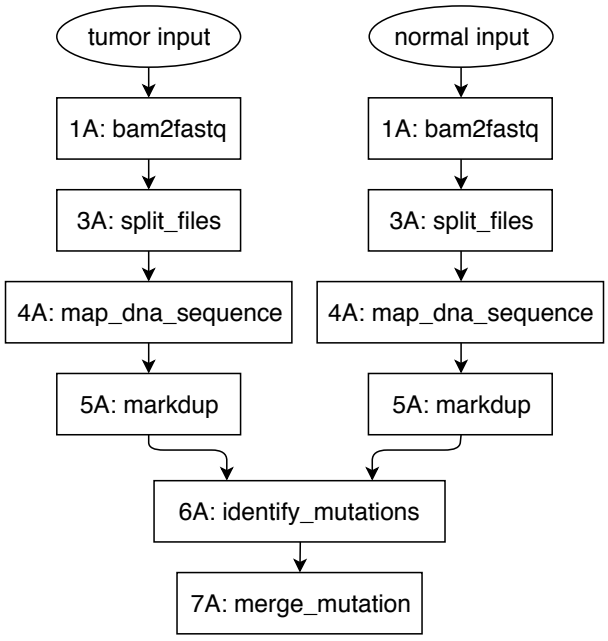


図 4.7: Genomon2 ワークフロー: 詳細設計（3）終了時

詳細設計（4）では、このワークフローに対して 2 つの改良を検討した。

ワークフロー改良 1　（図 4.8）
Task5A「markdup」逐次実行部 Task5A_stage2 の最後に行われる BAM ファイル MD5 チェックサム計算は、京での現問題規模の計算では 20 分程度かかるが、その結果は継続する Task6A「identify_mutations」以降では使用されない。そこで、MD5 チェックサム計算を Task5A_sgage2 から、新設した Task8A「post_markdup」移動させ、Task6A と 2 つの Task8A を並列に処理することにした。Task6A 処理時には使用ノード数に余裕があるため、Task6A と Task8A の実行は完全にオーバラップされ、この改良により BAM ファイルの MD5 チェックサム計算時間が隠蔽されることが期待される。

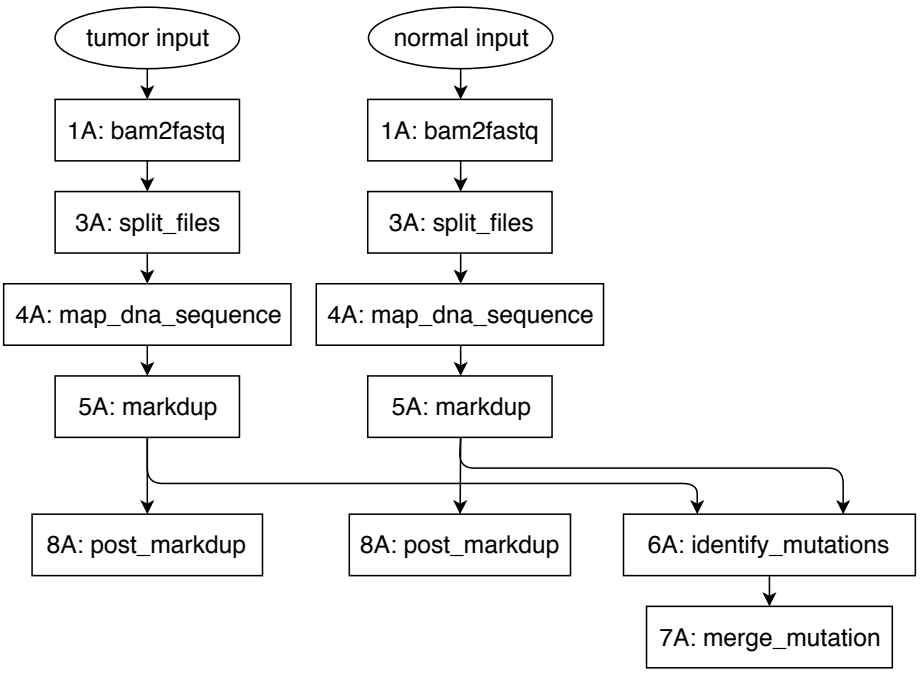


図 4.8: Genomon2 ワークフロー: 改良 1

ワークフロー改良 2　（図 4.9）

4.3.8 節で述べたように、アライメント結果データは Task5A「markdup」の並列計算部 Task5A_stage1 でリファレンス配列単位で後処理された後（分割 BAM ファイル、分割 BAM インデックスファイル）、逐次計算部 Task5A_stage2 においてこれらを結合している（結合 BAM ファイル、結合 BAM インデックスファイル）。実際には、Genomon2 の解析機能を富岳ターゲット問題の範囲（図 4.2 赤枠内）に限定すると、後続する Task6A「identify_mutations」以降の処理には結合 BAM ファイルと結合 BAM インデックスファイルは不要で、分割 BAM ファイルと分割 BAM インデックスファイルが準備されていれば十分である。そこで、Task5A「markdup」の逐次実行部 Task5A_stage2 を、改良 1 で分離した MD5 チェックサム計算部も含めて、新設した Task8A「merge_bam」移動させ、Task6A と 2 つの Task8A を並列に処理することにした。合わせて、一時ファイルである分割 BAM ファイルと分割 BAM インデックスファイルを消去するための Task9A「cleanup_bam」をワークフローの末尾に追加した。改良 1 と同様に、Task6A 処理時には使用ノード数に余裕があるため、Task6A と Task8A の実行は完全にオーバラップされ、この改良により旧 Task5A_stage2 の計算時間が隠蔽され、全パイプライン実行において改良 1 よりさらに 20 分程度の計算時間短縮が期待された。

これらの改良の結果、京における現問題規模の計算では、改良 1 においては期待された計算時間短縮が実現できた。しかし、改良 2 においては、全パイプライン実行計算時間は改良 1 と同程度で、実際には、Task5A_stage2 の計算時間が Task8A として隠蔽された代わりに、オーバラップされた Task6A の計算時間が増加していた。Task6A の計算時間増加の原因としては、Task8A 内のファイル結合処理での I/O 量が比較的大きいため、ディスクアクセス競合の発生により Task6A 内の I/O 時間が増加してしまったためと考えられる。

詳細設計（4）での富岳性能推定においては、改良 1 のワークフローを実装したコードを用いた。

ソフトウェア調整（1）では、改良 2 のワークフローを実装したコードを用いた。

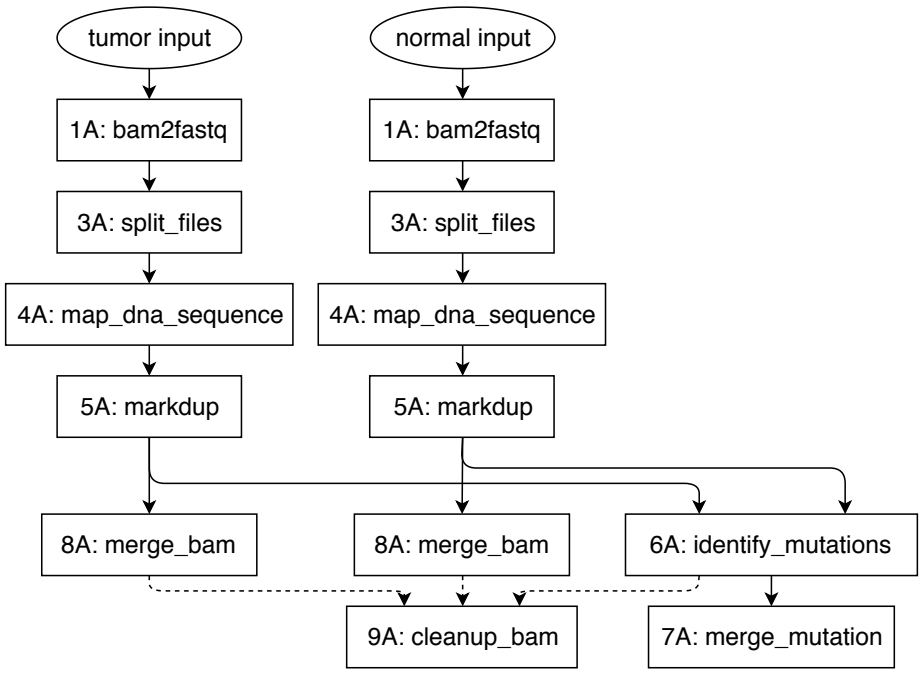


図 4.9: Genomon2 ワークフロー: 改良 2

4.3.10　コデザインによるシステムへの実装要請項目

本コデザインを通してシステムソフトウェア、プログラミング環境へフィードバックした実装要請・改善項目を以下に示す。

1. Genomon2 ワークフローにおけるファイル I/O
   富岳ターゲット問題解析時におけるファイル I/O 量の推定を行い（表 4.8)、富岳ファイルシステム設計への資料として提出した。

2. グリッドエンジン型バッチジョブシステム
   現在、Genomon をはじめとするバイオインフォマティクス分野での大規模解析アプリケーションの多くは、グリッドエンジンの使用を前提に並列化がなされている。富岳上でのこれらより多くのアプリケーションの実行を実現するためには、富岳のシステムジョブスケジューラ側において、グリッドエンジンと互換性のある機能とインタフェースを提供することが望まれる。富岳において、このような機能の提供が可能かどうか、運用系ソフトウェア開発側に検討を依頼した。

3. ページキャシュサイズを超えた I/O 長でのファイル読み込み時の遅延解消
   京のファイルシステムでは、C プログラムからページキャシュサイズ 128MB を超える I/O 長によるファイル読み込みを行うと遅延が発生する。Genomon2 ではアライメントプログラム BWA においてこの現象が発生し、約 750MB のファイル読み込みに 30 分以上かかった。現在は、ファイル読み込み時の遅延を避けるには、128MB 以下の I/O 長でファイル読み込みを実施するようシステム利用者側にプログラム修正による対処が必要とされている。富岳ではシステム側で対応するように要請した。

4. Python プログラミング環境の整備
   現在、バイオインフォマティクス分野におけるスクリプト言語の主流は、かつての Perl から Python へと移行が進んでいる。Genomon2 においても Python を、ワークフロー管理、ジョブスケジューラ、パイプラインを構成する個々のプログラムと、全ての階層で利用している。そのため、Genomon2 の京移植時には、Python と Python ライブラリのインストールおよび動作確認に非常に多くの時間が費やされた。富岳では、プログラミング環境として Python（2 系および 3 系）と主要 Python ライブラリ、特に、富岳に最適化された BLAS、LAPACK をリンクした NumPy、SciPy の提供を要請した。

5. Tofu 座標を取得する API
   Genomon2 において、富岳の第 1 階層ストレージを有効活用するためには、各ノードが使用する SSD（各ノードがどの BoB に所属するか）を知る必要がある。また、SIO ノードの特定ができれば、タスクのノードへの配置のさらなる最適化の可能性もある。これらは、各ノードの Tofu 座標が取得できれば可能であるため、富岳においても京と同程度の API の提供を要請した。

表 4.11: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (Genomon)

実装要請課題	対応
プロセスが効果的に充填されて計算コアの空きを減らすジョブスケジューリングの実現	Grid Engine 相当機能の対応方針を検討。階層ファイルシステムによるファイル共有について検討。
SIMD 化による文字列比較処理の高速化	標準関数の富岳向け高速化として開発する
zlib などのシステム付属ライブラリの性能改善	zlib などのシステムライブラリは gcc で翻訳し提供する。実機での性能検証で問題を検出した場合再検討
Python プログラミング環境の整備	オープンソースソフトウエアのサポートの一環として整備を実施
Genomon2 ワークフローを効果的に実現するファイル I/O の検討	富岳ターゲット問題解析時におけるファイル I/O 量の推定を行い、富岳ファイル設計への資料として提出。
ページキャシュサイズを超えた I/O 長でのファイル読み込み時の遅延解消	第 1 階層で計算ノード内キャッシュサイズ以上の先読みをしない等の対策、第 2 階層で階層間データ転送時に最適な IO 長を選択したり、キャッシュを使わない DirectI/O の使用等の対策を検討
グリッドエンジン型バッチジョブシステムのグリッドエンジンに相当する機能の提供	ジョブスケジューラ自体ではなく、通常 MPI ジョブとして VGE を実行することにより対応すると決定された。
Tofu 座標を取得する API の提供	Tofu 座標を取得する API を提供する (採用)

4.3.11 富岳向け性能推定

4.3.11.1 GFK パイプライン全体性能に対する性能推定

GFK は GFKalign、GFKdedup、GFKdetect の 3 つの処理部分からなる。まず、各処理部分で実行される外部プログラムについて、個別に性能推定を行った。

- アプリケーション全体のホットスポットとなるアライメントプログラム Blat は 5 区間、もうひとつのアライメントプログラム Bowtie は 2 区間、主要 Perl スクリプトについてはスク

リプト全体に対して FX100 上での精密 PA データ計測を行い、性能電力予測ツールにより実行性能を推定した。

- ただし、I/O 中心の計測区間、I/O 律速となる Perl スクリプトについては、富岳の I/O 性能を京と同程度と仮定し、京での実行時間をそのまま採用した。
- その他の実行時間の短いプログラムについては、FX100 での実測実行時間に対して動作周波数補正のみ行い、それを推定値とした。

以上は GFK で実行可能な RNA シークエンス解析の実測性能をベースにした性能推定である。富岳で想定している全ゲノムシークエンス解析における予測性能として、GFKalign, GFKdedup, GFKdetect の実行時間推定値に対して以下の補正を行った (以下の補正値は旧問題規模 (リード長 200、39 億リード) でのもの)。

- 解析対象が RNA シークエンスから全ゲノムシークエンスに変わることによる計算処理コストの増加はおよそ 10 倍と予想
- リード（入力データとなる塩基配列の断片）長が現在の 2 倍程度に、読み取り深度 (depth) が 5 倍程度増加すると予想
- GFKalign と GFKdetect の実行時間増加は 100 (= 10 × 2 × 5) 倍と評価
- リード長に依らない GFKdedup の実行時間増加は 50 (= 10 × 5) 倍と評価

GFK パイプライン全体性能は、システム全ノードを用いた複数検体同時パイプライン計算を想定し、4.3.2.4 節の方法を用いて、1 日あたりの解析可能検体数により評価した（表 4.12)。

表 4.12: Genomon(GFK) の性能評価結果 (詳細設計（２）旧問題規模)

機種	京	富岳			
		エコモード無効		エコモード有効	
		通常	ブースト	通常	ブースト
1 検体あたりの問題規模	リード長 200、39 億リード（ペアードエンド）				
全ノード数	82944	165888			
1 検体あたりの使用ノード数	162	145	141	145	141
1 日あたりの解析検体数	35.5	469	506	469	506
システム性能向上比	1	13	14	13	14
平均／最大システム消費電力 (MW)	-	23/25	30/32	18/20	22/24

詳細設計（２）の GFK 性能推定では、GFKdedup の前処理として GFKalign と同数ノード数で実行されていた部分を、GFKalign の一部として評価するように改めた。

4.3.11.2　Genomon2 アライメントプログラム BWA に対する性能推定

詳細設計（２）では、Genomon2 の京上での全ゲノムシークエンス解析における全ワークフローの完全な動作確認が完了していなかったため、先行して、Genomon2 パイプライン全体に対するホットスポットとなることが確実なアライメントプログラム BWA のみを対象とした富岳性能推定を行った（表 4.13)。

バージョン bwa-0.7.15 をベースに、ビッグエンディアン対応、データ読み込み時の遅延対策を施し、4SIMD 版 ksw_i32 関数（京でのベースライン計測には非 SIMD 版 ksw_i32 関数）を組み込

み、さらに OpenMP 化したコードを対象とした。プログラム全体を、ksw_i32 関数を 1 演算区間とし、他に 2 つの演算区間と 7 つの I/O 区間に分割し、FX100 上での精密 PA データ計測を行い、性能電力予測ツールにより実行性能を推定した。ただし、I/O 区間については、富岳の I/O 性能を京と同程度と仮定し、京での実行時間をそのまま採用した。

表 4.13: Genomon2 アライメントプログラム BWA の性能評価結果 (詳細設計（2）)

機種	京	FX100	富岳			
			エコモード無効		エコモード有効	
			通常	ブースト	通常	ブースト
問題規模	リード長 100、250000 リード（ペアードエンド）					
使用ノード数	1					
実行時間 (秒)	188.4	55.0	69.4	64.2	69.4	64.2

オリジナル BWA-MEM は 8 ビット整数に対する 16SIMD 演算を行っているが、詳細設計（2）での性能推定は、富岳においては 32 ビット整数に対する 8SIMD 演算を想定した場合の推定値である。富岳のコア命令セットでは 16 ビット整数、8 ビット整数に対する SIMD 拡張もサポートされる。また、詳細設計（2）の性能推定はリード長 100 塩基対の入力データに対して行ったが、富岳ターゲット問題ではリード長は 150 塩基対に増加する。これらをふまえて、32 ビットより小さい整数データに対する SIMD 演算において、実行性能が SIMD 同時演算数に対して、オリジナル BWA-MEM の 16SIMD を超えて、どこまでスケールするかについての検証を行なった。

4.3.11.3　Genomon2 パイプライン全体性能に対する性能推定

Genomon2 においても、GFK と同様に、富岳全系を用いた解析における、1 日あたりの解析可能検体数により性能を評価する。現在の VGE の実装では複数検体同時解析に対応していないため、1 検体ごとの解析を独立したジョブとして、それを複数同時実行することを想定している。

4.3.11.4　使用ノード数の決定

GFK では 1 検体解析計算は 3 つの MPI プログラムに分割され、それぞれの MPI プロセス数は 4.3.2.4 節に示した方法により最適な値が決定できた。一方、Genomon2 においては、1 検体解析実行時の MPI プロセス数 (=VGE ワーカプロセス数 +1)、あるいは使用ノード数を天下り的に決定する必要がある。この使用ノード数は、性能評価値に大きく影響することに注意する必要がある。使用ノード数を少なくとると (極端な場合には 1 ノードなど)、スループット性能としての 1 日あたりの解析可能検体数は増大するが、1 検体に対する解析開始から終了までの時間で評価されるレイテンシ性能は悪化してしまう。Genomon2 の実際の応用時を想定すると、スループット性能をある程度まで犠牲にしても、1 検体あたりの使用ノード数を増加させることにより、1 検体解析にかかる時間を現実的に許容可能な値（我々は 1 日を想定）に近づける必要がある。

予備調査段階での見積り[8]では、富岳ターゲット問題解析時のおける 1 日あたりの解析可能検体数および 1 検体解析にかかる計算時間は、表 4.14 のように評価された。

この予備調査の結果をふまえ、性能評価に用いる実行条件として、1 検体解析に使用するノード数を以下のように決定した。

[8] I/O ノード非占用での計測結果をもとにした評価。富岳ターゲット問題への換算係数なども、この後に見直しを行っている。

表 4.14: Genomon2 予備調査により得られた予想実行性能と使用ノード数の関係

機種	京ベースライン		ポスト京 (通常モード)	
1 検体計算ノード数	36	192	48	96
解析検体数/日	308.8	69.0	2158.4	1415.4
計算時間 (日)	7.5	6.3	1.6	1.2

京ベースライン: 36 ノード

富岳: 96 ノード (384MPI プロセス)

予備調査による推定値であるが、1 日あたりの解析可能検体数をもとにしたシステム性能向上比が、「京」36 ノード→「ポスト京」96 ノードで 4.5 倍、「京」192 ノード→「ポスト京」48 ノードで 34.7 倍と、1 検体解析に使用するノード数の選択に大きく依存することに注意が必要である。

4.3.11.5　性能推定方法

富岳ターゲット問題 (リード長 150 塩基対、リード数 14 億) に対する入力データはまだ存在しないため、現在入手可能なサンプル入力データ (リード長 100 塩基対、平均リード数 6.78 億) に対する京上での実行性能をもとに、各タスク内で実行されるコマンド毎に以下の換算係数を乗じて実行時間を推定した。

- 計算量がリード長に比例する処理は 1.5(= 150/100) 倍
- 計算量がリード数または読み取り深度 (depth 値) に比例する処理は 2.06(= 14/6.78) 倍
- 計算量がリード長およびリード数に比例する処理は 3.09(= 1.5 × 2.06) 倍
- さらに富岳での実行時間については、京との CPU 動作周波数比倍

入力リードファイルの分割については、現在のサンプル入力データに対する計算時と同様に、Fastq ファイル 100 万行 (25 万リード) を 1 分割ファイルとするものとした。

4.3.11.6　性能評価結果

表 4.15 に、富岳ターゲット問題における実行性能推定値をまとめる。この表では、参考のために、問題サイズ変更後の GFK 性能値とともに、システム性能向上比として GFK の京ベースライン性能を基準にした値も掲載した。

表 4.15: Genomon2 性能推定

Genomon バージョン	v1(GFK)	v2(Genomon2)				
		詳細設計 (4)			ソフトウェア調整 (1)	
機種	京	京	ポスト京		ポスト京	
	ベースライン	ベースライン	通常	ブースト	通常	ブースト
1 検体計算使用ノード数	128	36	96	96	96	96
解析検体数/日	125	261	2025	2227	2040	2243
1 検体計算時間 (日)	15.57	8.84	0.82	0.74	0.81	0.74
システム性能向上比 (v2 基準)	-	1.0	7.8	8.5	7.8	8.6
システム性能向上比 (v1 基準)	1.0	2.1	16.2	17.8	16.3	18.0

性能目標としていた「1 日に 1,000 検体の解析」は詳細設計（3）により達成されていたが、詳細設計（4）での最適化により、1 日あたりの解析検体数は 2,000 を超え、また、もうひとつの性能目標「1 検体の解析を 1 日以内に完了」も達成された。

京でノード単位で、富岳では CMG 単位で MPI を使用した際、全系を用いた計算では、富岳での MPI 並列数は京の 7.7 倍になる。このことから、京と同一 CPU クロック周波数の富岳通常モードでの Genomon2 全系計算では、システム性能向上比 7.7 倍をひとつの基準値として考えることができる。詳細設計（4）では、Genomon2 基準での通常モードのシステム性能向上比が 7.8 倍となり、1 検体解析にかかる実行時間を 1 日以内とするというノード配置に課せられた制限下においても、この基準値 7.7 倍を上回る性能向上比が得られた。

ソフトウェア調整（1）では、「ワークフロー改良 2」のコードに対して性能推定を行った。「ワークフロー改良 2」では、Ruffus タスクレベルおよび VGE ジョブレベルでの並列度が上がったが、ディスクアクセスの競合によりファイルシステムへの負荷が増加する傾向がある。そのため、ファイル I/O 時間については京での実測値をもとにしている本推定では、詳細設計（4）からの性能向上は僅かであった。

4.3.11.7　Genomon2 メモリ消費量の推定

京での、現問題のパイプライン実行時、および、富岳ターゲット問題と同じリード長 150 塩基対の疑似リードデータに対するパイプライン実行（原理的に実行が不可能な Task7A, Task8A を除く）では、BWA 実行中に消費メモリ量が最大となった。この傾向が富岳での富岳ターゲット問題解析時でも同様であると仮定し、Genomon2 のノードあたりのメモリ消費量を以下のように推定することとした。

$$\begin{aligned} \text{富岳メモリ消費量} = &(\text{VGE ワーカープロセスメモリ消費量} + \text{BWA メモリ消費量}) \times 4 \\ &+ \text{MPI ライブラリメモリ消費量} \end{aligned} \tag{4.4}$$

ここで、右辺第 1 項の 4 倍は、CMG を単位として 1 ノードに 4 つの VGE ワーカープロセスが実行されるためである。

BWA のメモリ消費量について、詳細設計（4）時点では富岳ターゲット問題と同等なリード長 150 塩基対の入力データが得られなかったが、事前の調査により、現問題のリード長 100 塩基対リードデータでの実行時よりメモリ消費量が減少することが予想された。そこで、上限値として、ラウンドロビン分割した現問題リードデータに対する、FX100 での 12 スレッド実行時の実測メモリ消費量 6571.3MiB を採用した。また、VGE ワーカープロセスメモリ消費量には京での現問題実行時の計測データからの推定値 106.8MiB、MPI ライブラリ消費メモリ量には富岳での推定値 2.8MiB を用いて、以下のように見積もった。

$$\text{富岳メモリ消費量} = (106.8\text{MiB} + 6571.3\text{MiB}) \times 4 + 2.8\text{MiB} = 26.1\text{GiB} \tag{4.5}$$

ソフトウェア調整（1）では、期間中に入手できた 150 塩基対リードデータに対して富岳試作機上での BWA 実行時のメモリ消費量の計測を実施した。予備調査として、ターゲット問題と同様に、150 塩基対リードデータを 25 万リードごとに分割し、それらに対して京上で 8 スレッド実行を行った。試作機での計測では、1 ノード上で mpiexec から直接 4 つの BWA コマンドを実行し、その時のノードあたりの消費メモリ量を計測した。各 BWA プロセスとも京上で最も消費メモリ量の多かった分割を入力データとした。BWA プログラムは 32SIMD 版コードを使用し、実行時にベクトルレジスタ長を 256bit に縮小している。富士通社内の 2019 年 12 月末現在の試作機環境での

計測結果では、ディフォルトのラージページパラメータ設定では、メモリ不足により実行できなかった。スレッドヒープ獲得パラメータの調整を行い、環境変数 XOS_MMM_L_HEAP_SIZE_MB を 64 に、MALLOC_MMAP_THRESHOLD_を 33554432(=32MiB) に指定することにより実行可能となり、ノードあたりの消費メモリ量 27.8GiB という結果を得た。

4.3.12　富岳上でのチューニングと性能評価

ここでは、共用前評価環境にて富岳上で行った、Genomon2 のチューニングと、100 塩基対リードデータに対する実行性能、消費メモリ、消費電力、また、それを元に推定したターゲット問題の実行性能について報告する。

4.3.12.1　変異検証計算のロードバランス改善

富岳上での性能評価では、京と比較して Python インタプリタ自身の実行性能が向上していたため、今まで見えていなかった Task6A「identify_mutations」の EBFilter コマンドのロードインバランスが顕著となり、Task6A の新たなボトルネックとなっていることが判明した。そこで、EBFilter のデータ分割方法を動的なものに改め、ロードバランスの改善を図った。この修正により、第 1 階層ファイルシステム未使用時での Task6A 実行時間を、46 分から 30 分まで短縮できた。

4.3.12.2　LLIO ファイルシステム利用方法の再検討

第 1 階層ストレージ上の LLIO ファイルシステムは、基本的には 4.3.4.2 節の方針に従い使用した。

ノード内テンポラリ Task5A「markdup」以外では、タスク内プログラム間でのデータ受け渡しに使用。

共有テンポラリ タスク間でのデータ受け渡しに使用。また、ファイルサイズが大きい Task5A「markdup」では、タスク内プログラム間でのデータ受け渡し、およびプログラム内テンポラリファイルにも使用。

第 2 階層キャッシュ（複製なし） 入力 FASTQ ファイルと計算結果ファイル、および、Genomon で使用するシステム提供以外のプログラム、共有ライブラリファイル、Python モジュール類もキャッシングの対象とした。

第 2 階層キャッシュ（複製あり） Task3A「map_dna_sequence」および Task6A「identify_mutations」で参照されるインデックスファイルおよび変異 DB ファイルは、Genomon 実行前に llio_transfer コマンドにより、第 2 階層キャッシュ領域に SSD 単位で複製をコピーしておく。

富岳実機上での性能計測の結果、第 2 階層キャッシュ（複製なし）の利用方法を以下のように改めた [54]。

Task3A「split_files」では，count_lines コマンドにより第 2 階層ファイルシステム上の入力 FASTQ ファイルの行数をカウントし、split_file コマンドにより同ファイルを 4 行単位でラウンドロビン分割する処理が、normal と tumor で 2 つずつ、計 4 つの VGE ジョブが同時実行される。表 4.16、図 4.10 に検討した各ケースについて、最も時間のかかった VGE ジョブでの実行時間内訳を示す。

表 4.16: Task3A「split_files」最も時間のかかった VGE ジョブの実行時間内訳（秒）

ケース	A	B	C	D
共有テンポラリ		✓	✓	✓
第 2 階層キャッシュ			✓	✓
一部キャッシュ停止				✓
split_file 改良				✓
count_lines	352	368	690	347
split_file	2173	1905	1511	1146

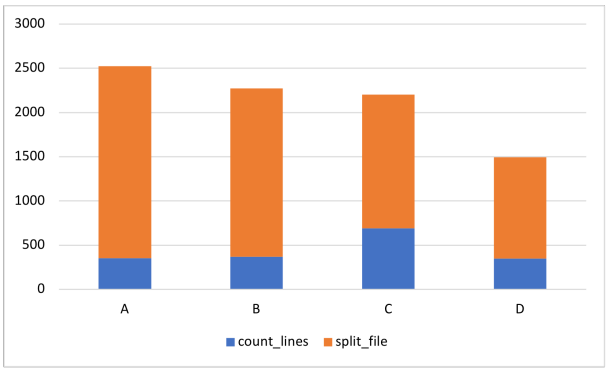


図 4.10: Task3A「split_files」最も時間のかかった VGE ジョブの実行時間内訳（秒）

ケース (A) が LLIO 未使用時の実行時間。ケース (B) が分割後ファイルの出力先を共有テンポラリ領域にしたもの、ケース (C) は count_lines コマンド実行時に FASTQ ファイルをキャッシュに載せ、split_file コマンドはキャッシュ上のデータに対して実行するようにしたものである。第 2 階層キャッシュの使用により、split_file は 400 秒ほど高速化されたが、逆に count_lines は 320 秒ほど遅くなっている。これは、キャッシュに載せるためのオーバヘッドが原因と考えられる。そこで、ケース (D) では FASTQ ファイルのみ第 2 階層キャッシュの使用は止め、同時に split_file コマンドのファイル入力部分を、マルチスレッドによる並列読み出し化により高速化を図った。改良後の split_file コマンドは、改良前のキャッシュ上データ読み込み時より高速であったため、ケース (D) を採用することとした。

Task6A「identify_mutations」の 275VGE ジョブと Task8A「merge_bam」の 2VGE のジョブが同時実行される。表 4.17、図 4.11 に検討した各ケースについて、最も時間のかかったジョブでの実行時間内訳を示す。ケース (A) は LLIO 未使用時の実行時間内訳である。ケース (B) では、共有テンポラリ領域上のマッピング結果ファイルを参照するように、また、タスク identify_mutations 内の一時ファイルをノード内テンポラリ領域に置くようにしたもので、素直に LLIO の効果が出ている。ケース (C) は、第 2 階層キャッシュを有効にしたものだが、タスク identify_mutations では若干の、タスク merge_bam では samtools cat コマンドで顕著な速度低下が見られる。samtools cat は、マッピング結果ファイルを結合する、ほぼ I/O 処理だけのコマンドである。第 2 階層キャッシュの有効化により、SSD のアクセス競合が増加し、I/O 性能が低下しているものと考えられる。ケース (D) は、タスク identify_mutations において、コントロールパネルファイルをキャッシングの対象から外したものである。コントロールパネルファイルは、ランダムアクセスのためキャッシングの効果は低く、キャシング無効化の影響はほとんどないものと思わる。逆に、SSD のアク

セス競合が減少したことにより、タスク identify_mutations では若干の、タスク merge_bam では samtools cat コマンドで顕著な計算速度の回復が見られた。このことから、ケース (D) を採用した。

表 4.17: Task6A「identify_mutations」Task8A「merge_bam」最も時間のかかった VGE ジョブの実行時間内訳（秒）

ケース	A	B	C	D
ノード内テンポラリ		✓	✓	✓
共有テンポラリ		✓	✓	✓
第 2 階層キャッシュ			✓	✓
一部キャッシュ停止				✓
identify_mutations				
fisher	911	727	767	621
realignment	182	104	154	97
indel	208	144	274	134
breakpoint	134	120	128	110
simplerepeat	7	9	9	6
ebfilter	372	292	297	279
merge_bam				
samtools cat	1002	164	1089	191
bai_merge	7	5	7	5
md5sum	959	634	591	635

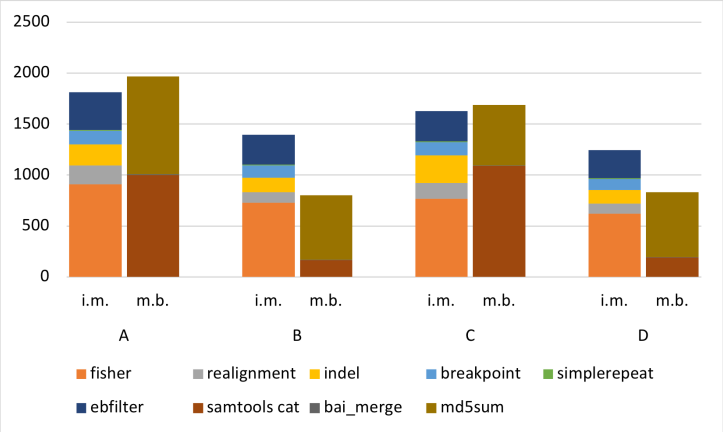


図 4.11: Task6A「identify_mutations(i.b.)」Task8A「merge_bam(m.b.)」最も時間のかかった VGE ジョブの実行時間内訳（秒）

4.3.12.3　100 塩基長データに対する計算性能

共用前評価環境期間においても、ターゲット問題の対象である 150 塩基長データの入手が不可能であったため、今までと同様に 100 塩基長データを用いて計算性能を計測した。計測時のノード割り付けには「4x3x8:strict-io」を指定し、ストライピングの設定は、第 2 階層上の入出力データファイルはストライプサイズ 2MiB、ストライプ数 32，LLIO の共有テンポラリおよび第 2 階層キャッシュ領域はストライプサイズ 2MiB、ストライプ数は 96 ノード使用時に可能な最大数である 6 としている。なお、Genomon では入力ファイルを全て実行開始前に LLIO 領域に移動させることができないため、実行中に他ユーザと共用している第 2 階層ファイルシステムへのアクセスが必須となる。そのため、実行時の第 2 階層ファイルシステムの負荷状況が、性能計測結果の再現性に影響する。計測は、通常モード/ブーストモード、エコ無効/有効の組み合わせ 4 ケースについて行ったが、各ケースで複数回計測して最速実行時の値を採用することとした。

表 4.18 に、得られた計算性能を示す。表中の平均消費電力と最大消費電力は、ジョブ実行情報ファイルのそれぞれ “AVG POWER CONSUMPTION OF NODE (MEASURED)” と “MAX POWER CONSUMPTION OF NODE (MEASURED)” のノード間平均値である。Genomon では計算速度に対するエコ設定の影響より外部要因による実行時間のばらつきの方が大きいため、今回の計測では、エコ有効ブーストモード時がエコ無効ブーストモード時よりわずかに高速となり、最も計算性能が高いケースとなった。

表 4.18: 富岳での 100 塩基長データ解析時の計算性能

	エコ無効		エコ有効	
	通常	ブースト	通常	ブースト
実行時間 (s)	8918	8552	9038	8532
最大メモリ/ノード (GiB)	24.8	24.8	24.7	24.8
平均消費電力/ノード (W)	103	116	76	84
最大消費電力/ノード (W)	110	123	83	92

Genomon では消費電力がノードの種類毎に大きく異る。図 4.12 にノード種別に平均した消費電力値を示す。

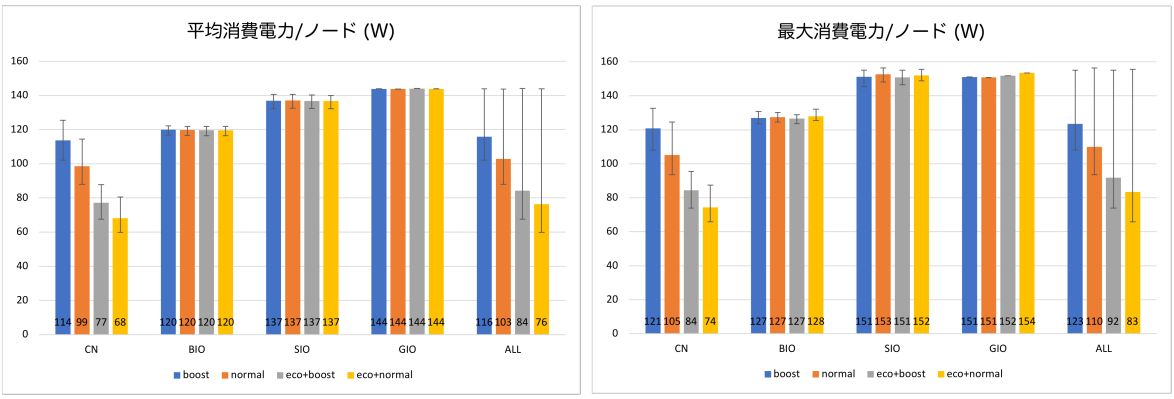


図 4.12: 平均消費電力 (左図) と最大消費電力 (右図) のノード種別の平均: CN=IO ノード非兼任な計算ノード (83 個)、BIO=BIO ノード (6 個)、SIO=SIO ノード (6 個)、GIO=GIO ノード (1 個)、ALL=全計算ノード (96 個)。棒クラフ長と数値はノード間平均値、エラーバーはノード間での最小値〜最大値を表す。

4.3.12.4　ターゲット問題計算性能

前節での 100 塩基長データに対する計算性能実測値から、ターゲット問題（150 塩基長）での計算性能を 4.3.11.3 節と同様な手法により推定した。

表 4.19 に、得られたターゲット問題計算性能を示す。表中の最大消費メモリ、消費電力は、100 塩基長データに対する実行時の実測値であり、ターゲット問題（150 塩基長）への換算は行っていない。

表 4.19: 富岳での実測値から推定したターゲット問題計算性能

	京	ポスト京			
	ベースライン	エコ無効		エコ有効	
		通常	ブースト	通常	ブースト
1 検体計算使用ノード数	36	96	96	96	96
解析検体数/日	261	5888	6046	5732	6091
1 検体計算時間 (日)	8.84	0.28	0.27	0.29	0.27
システム性能向上比	1	22.6	23.2	22.0	23.4
最大メモリ/ノード (GiB)	-	24.8	24.8	24.7	24.8
平均消費電力/ノード (W)	-	103	116	76	84
最大消費電力/ノード (W)	-	110	123	83	92

表 4.20 に、富岳で達成された京に対する性能倍率および消費電力をまとめる。

表 4.20: Genomon の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
Genomon	23 倍	20 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

表 4.21: Genomon の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
Genomon	23 倍		23 倍		23 倍		22 倍	
	18 MW	20 MW	16 MW	17 MW	13 MW	15 MW	12 MW	13 MW

富岳で達成された性能倍率 23 倍は、ソフトウェア調整（1）での推定性能倍率 8 倍を大きく上回った。以下に、その理由を考察する。

過去の性能推定時には、以下の方針をとっていた。

- Genomon パイプラインを構成するプログラム群は、いわゆる HPC 的なプログラムと違い、ハードウェア資源を考慮しての数値的な性能推定が困難であったため、多くのものは CPU クロック周波数比による換算のみとしていた。
- また、富岳のディスク I/O 性能についても、京と同程度として評価していた。

図 4.13 に、ソフトウェア調整（1）での性能推定時の京、LLIO 未使用時の富岳、LLIO 使用時の富岳について、100 塩基対データに対する Genomon2 パイプライン実行中の主要タスクの実行

時間を示す。CPU クロック周波数は、いずれも 2GHz である。この図から読み取れるように、過去の性能推定時に考慮していなかった以下の点が、性能倍率の大幅向上の要因となった。

- Task3A では、LLIO 未使用時でも、ファイルシステムの性能向上とプログラムの I/O チューニングにより 3.6 倍高速化した。
- Task4A では、アライメントプログラム BWA の SVE 移植が効果をあげ、京→ポスト京 (LLIO 無) で 2 倍高速化している。
- Task6A では、京→ポスト京 (LLIO 無) で 3.7 倍高速化している。これは、MPI プロセス内の並列度が 8 から 12 に増加した点や EBFilter コマンドのロードバランス改善の効果もあるが、Python インタプリタ自身の実行性能が向上していることが主因と思われる。
- パイプライン全体実行時間は、ポスト京 (LLIO 無) で 3 時間 32 分、ポスト京 (LLIO 有) で 2 時間 28 分であった。LLIO を使用することにより実行時間は約 1 時間短縮され、計算性能は 1.4 倍高速化された。

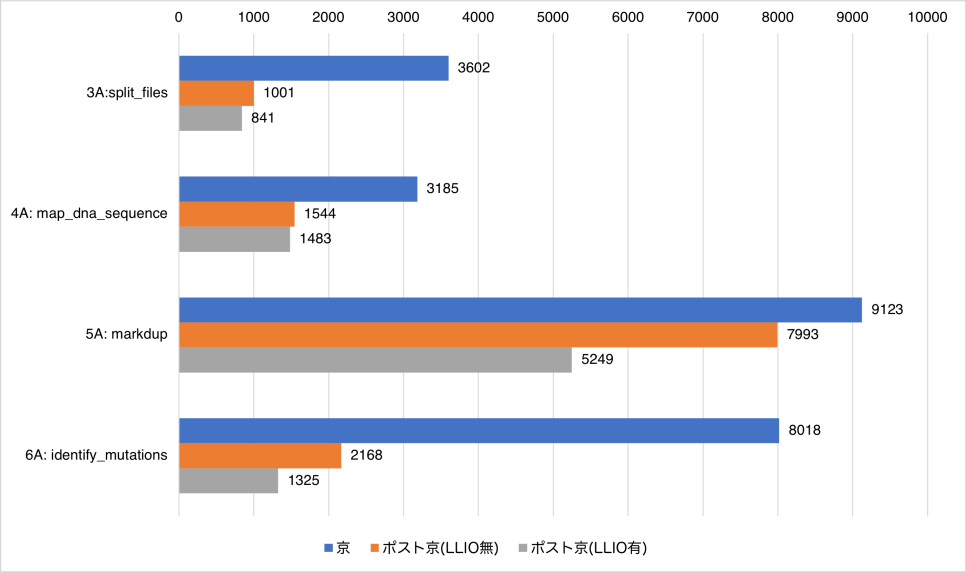


図 4.13: 100 塩基対データに対する Genomon2 主要タスク実行時間（秒）：「京」はソフトウェア調整（1）での性能推定時のコード、「ポスト京 (LLIO 無)」「ポスト京 (LLIO 有)」では最終版コードを使用。

4.4 重点課題 3 GAMERA のコデザイン

4.4.1 アプリケーションの特徴とコデザインの方針

GAMERA は地盤地震動シミュレーションのための陰解法・非構造格子有限要素法プログラムである。このプログラムでは共役勾配法をベースに精度混合演算・マルチグリッドを組み合わせたソルバーが各時間ステップにおける大規模な連立一次方程式の求解に使われており、共役勾配法の行列ベクトル積には Element-by-Element(EBE) 法が使われている。通常の有限要素法では疎な全体行列

$$K = \sum_i P_i K_i P_i^T, \tag{4.6}$$

をメモリ上に格納し、行列ベクトル積

$$f \Leftarrow Ku, \tag{4.7}$$

を計算する度にメモリから K を読み出すのに対し、EBE 法では式 (4.7) を

$$f \Leftarrow \sum_i (P_i(K_i(P_i^T u))), \tag{4.8}$$

として計算する。ここで、添え字は要素番号 i に関する値を示し、K_i は要素行列、P_i は全体節点番号と要素節点番号のマッピング行列である。行列ベクトル積に式 (4.8) を使った場合は、K_i を行列ベクトル積を実行する度に計算する必要があるものの、K_i を計算するためのデータ量と P_i のデータ量は全体行列 K のデータ量より少なくなるため、式 (4.7) を使う場合に比べてメモリアクセス量を減らすことができる。その一方で、K_i を計算するために多くの変数を使ったオンキャッシュ計算が必要となり、また、u のランダムロードと f へのランダムアクセスを介した足しこみが必要になる。ソルバーのうち、マルチグリッド前処理に使われる四面体二次要素メッシュにおける単精度 EBE カーネルが計算時間の大部分を占める。

このように GAMERA は、オンキャッシュ・非構造計算として、インダイレクト SIMD ロード・低キャッシュレイテンシなどのランダムアクセス性能を必要とするアプリケーションとしてコデザインに寄与する。

以下、GAMERA の単精度 EBE カーネルの最適化とその計算科学的観点からのコデザインについて述べる。

4.4.2 コデザイン前の EBE カーネル

図 4.14 にコデザイン前の EBE カーネルの概要を示す。式 (4.8) を要素方向 (i) に並列計算する際には、節点を共有する要素間で f への足しこみ競合が起こる。そこで、コデザイン前のカーネルでは、スレッド毎にテンポラリ配列 ftmp を確保し (図中区間 I)、それらに各スレッドが $(P_i(K_i(P_i^T u)))$ を足しこみ (図中区間 II)、最後にスレッド毎の計算結果を全体右辺ベクトル f に足しこむ (図中区間 III) 方式をとっていた。ここでは、要素番号と節点番号を並び替えることで、節点値 u 及び f のキャッシュ上の再利用を促している。テンポラリ配列のうち各スレッドがアクセスする部分は限られているため、その部分だけ 0 初期化 (区間 I) と足しこみ (区間 III) をすることでスレッド並列化の追加コストを減らしている。その一方で、区間 II において ftmp への ie 方向のスレッド内の足しこみ競合が避けられないため、主要計算部となる区間 II において SIMD 演算を適用することができなかった。

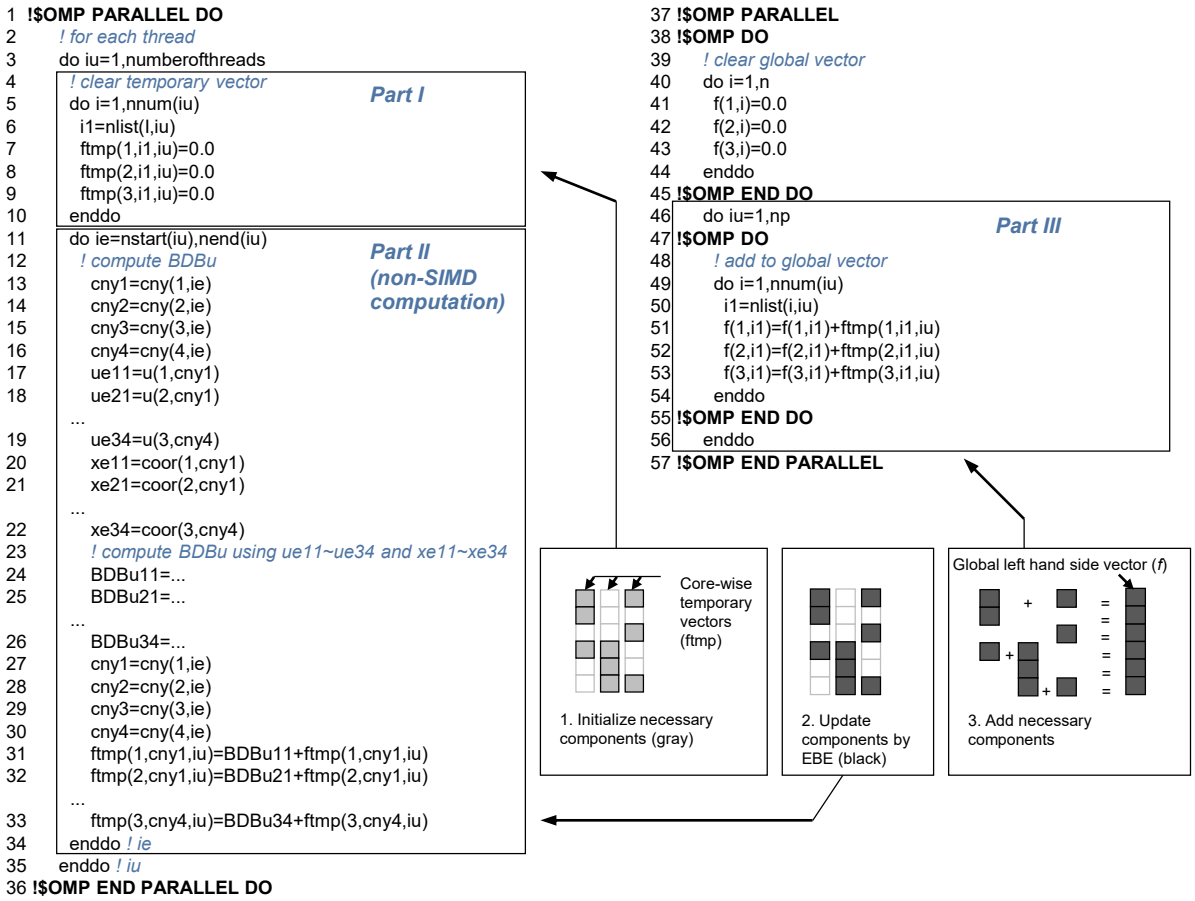


図 4.14: コデザイン前の EBE カーネル

4.4.3 EBE における SIMD 演算器の利用

そこでまず富岳において SIMD 幅が増えることを念頭に、区間 II の SIMD 並列化に取り組んだ。ここでは、区間 II をデータリカレンスのある部分とそうでない部分の二つに分けることで (図 4.15 区間 II-A と II-B)、前半部において SIMD による並列計算ができるようにした。区間 II-A において EBE カーネルの主要計算である $(P_i(K_i(P_i^T u)))$ が実施されるため、この部分の SIMD 化によりカーネルの浮動小数点計算のほとんどを SIMD 演算器を使って実行できるようになる。区間 II を二つに分けるためには区間 II-A と II-B をつなぐテンポラリバッファ(BDBu11~BDBu34) が必要になるが、計算機の SIMD 幅に合わせたループブロッキング (ブロック長 NL=SIMD 幅) をあわせて実装することでこれらのテンポラリバッファの大きさを抑えることができ、主要計算を L1 キャッシュ上に載せたまま SIMD 計算できることとなる。

4.4.4 EBE におけるカラリングによるリストアクセスの削減

SIMD 演算器の利用により区間 II が速くなったため、相対的に区間 I 及び区間 III におけるランダムアクセスコストが見えるようになった。特に富岳においては京よりも SIMD 幅が増える分これらのランダムアクセスのコストが相対的に増えるため、これらのランダムアクセスコストを減らすことが重要となる。そこで富岳に向けたコデザインとして、要素ループのカラリングを実施

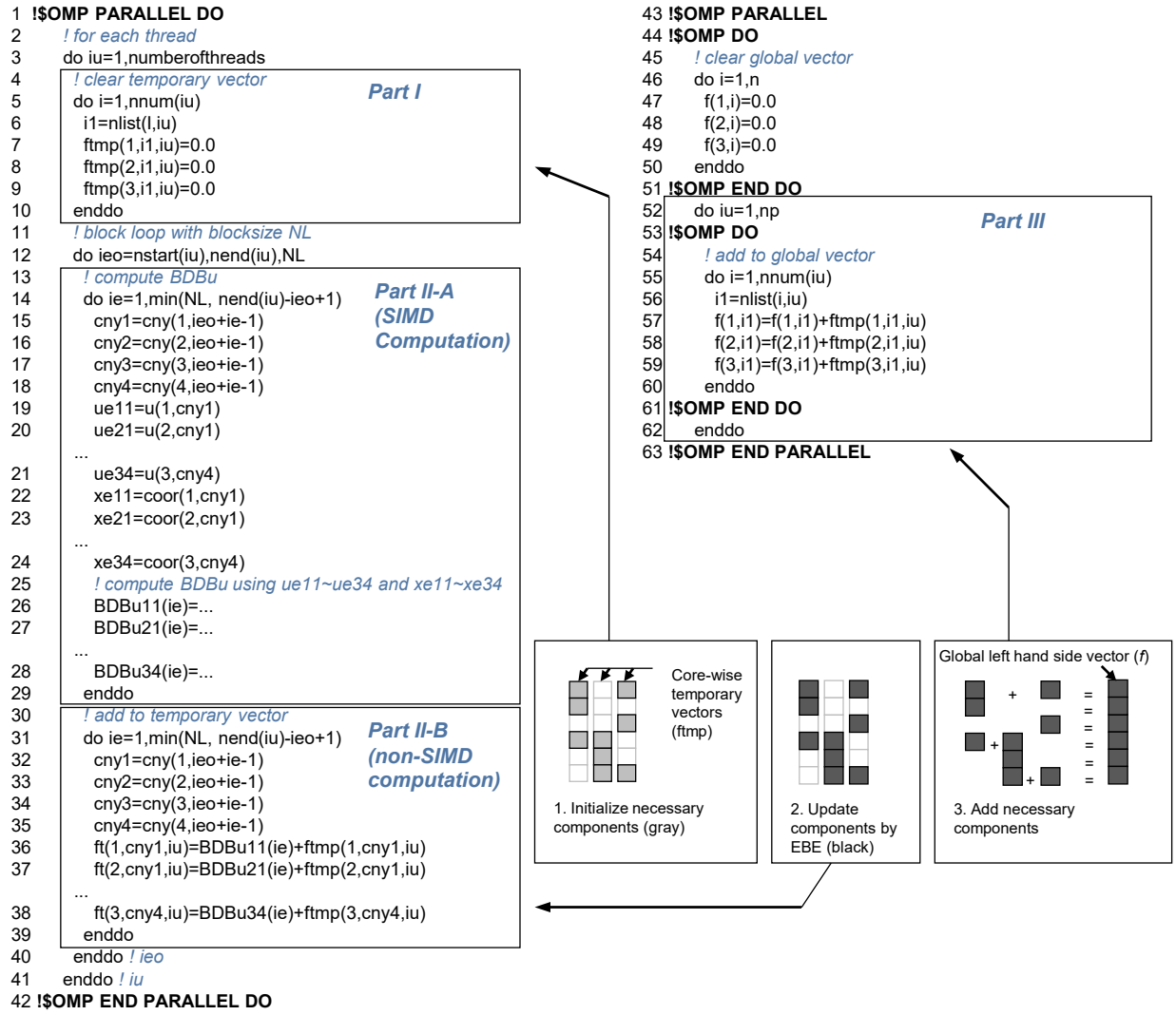


図 4.15: SIMD 演算器を利用した EBE カーネル

した (図 4.16)。通常の要素カラリングでは、要素群のうち節点を共有しないサブセットを取り出し、それを一つのカラーとする (図 4.17)。この方法により、各スレッドで計算した節点値は f に直接足しこめるようになる一方で、カラー内の要素の u と f はキャッシュ上で再利用できなくなるため、カラリング前に比べてメモリアクセス量が増えてしまう。そこで、カラリングをする際にグラフ分割手法を使って、全体メッシュをスレッド数に分割する方法をとった (図 4.18)。これにより、各スレッド内において u と f がキャッシュ上で再利用できるようになり、かつカラリング適用前と比較して区間 I と III が不要となった。この効果により、カラリング適用前と比べて性能電力予測ツールによるこの区間の実行時間の予測値は 33%短縮された。

4.4.5 EBE におけるループ分割によるスピルの削減

EBE 法の主要計算部分である区間 II-A (図 4.16) は、ソースコード上で 500 行以上の大きいループボディをもつ。京においては多数のレジスタを活用することでキャッシュアクセスを減らした高速計算が可能であったが、富岳ではレジスタ数が減るためにこの区間においてスピルが多発する

ことが予想された。実際、基本設計でコデザインしたコードの該当部分を SPARC8SIMD 性能シミュレータ用にコンパイルすると多数のレジスタスピルが発生することがわかった。そこで、該当のループのループ分割数を基本設計時の 6 個から、18 個のループに再分割した。これにより、アセンブリコードにおけるスピル数を基本設計時の 6692 個から 1149 個へと約 1/5 に減らすことができた。

4.4.6 コデザインによるシステムへの実装要請項目

本コデザインを通してシステムへフィードバックした実装要請・改善項目を以下に示す。

1. インダイレクト SIMD ロードの高速化のための L1 アクセスまとめ上げ

   EBE の演算カーネルの主要演算ループに SIMD 演算が使えるようになり、また、メニーコアに対するスレッド足しこみコストが低減したことから、本カーネルの主なボトルネックは要素右辺ベクトルのロード (図 4.16 の 18~27 行目) と、要素左辺ベクトルの全体左辺ベクトルへの足しこみ (図 4.16 の 39~41 行目) となる。要素右辺ベクトルのロードはインダイレクト SIMD ロード、要素左辺ベクトル足しこみはデータのリカレンスのため、非 SIMD ロード・ストアとなる。そこで、これらの部分の高速化をコデザインとして要望してきた。具体的には、インダイレクト SIMD ロードの高速化のための L1 アクセスまとめ上げである。通常、インダイレクト SIMD ロード命令における複数の要求先が同一のキャッシュラインに含まれていたとしても、L1 へのアクセス要求が要求先の個数分発行され L1 エンジン律速となってしまう。そこで本システムにおいては、インダイレクト SIMD ロード命令における要求アドレスが同一のキャッシュラインに含まれている場合には L1 へのアクセス要求回数を減らすハードウェア機構が導入されることになった。なお、システムへ要望するだけでなく、アルゴリズム側でもこの部分のコストを下げる工夫も行っている。例えば、ベクトル成分は節点順が非構造となっているものの、各節点の x, y, z 成分は連続にメモリ上に配置されるため、これら 3 成分をまとめてロード・ストアするようにすることでコストを削減する方法を検討した。

2. コンパイラによるループ分割機能の適用と処理の最適化

   EBE の演算カーネルはループボディが長く、ループボディ内の全ての変数をレジスタ上に載せることができないため、スピル・フィルが発生する。本検討においては手動でループ分割をすることで性能劣化を抑えてきたが、分割位置の決定などに多くの手間がかかっていた。そこで、これらのループ分割をコンパイラにて自動で高性能で実施できるようシステム側に要望し、採用された。

表 4.22: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (GAMERA)

実装要請課題	対応
コンパイラによるループ分割機能の提供と処理の最適化	GAMERA 区間 B のコンパイラによるループ分割の効率化。富岳向けコンパイラの機能として開発
間接 SIMD ロード命令実行時の L1 キャッシュに対するアクセスリクエスト削減	間接 SIMD ロード命令実行時に、2 要素単位に同一 128Byte ブロックへのアクセスであるかをチェックし、同一の場合は 2 つのアクセスリクエストを 1 つに纏め上げてリクエスト数を削減する

表 4.23: GAMERA の京・富岳におけるインバランス係数の推定結果

構成	ノード数	プロセス数/ノード	プロセス数	インバランス係数 α
京	82944	1	82944	1.33
富岳	165888	4	663552	1.45

4.4.7 富岳向け性能推定

富岳向け性能推定は以下の手順により実施した。

1. 実行時間のほぼすべてを占めるタイムステップループ部分全体を 15 個の演算区間、及び、7 個の通信区間に分け、3 タイムステップ分に対してそれぞれの区間の実行時間を以下の手順で推定した。

   演算区間については、ターゲット問題を富岳全ノードで実行した場合に対する 1 プロセスのウイークスケーリング問題を対象に、富岳における実行時間を推定した。ここでは、FX100 1 ノード 1 プロセス (1 プロセス/CMG) を用いて区間毎の詳細プロファイラ情報を取得し、性能電力予測ツールを用いて、富岳における実行時間を推定した。

   通信区間については、ターゲット問題を富岳全ノードで実行した場合に対する 18432 プロセスのウイークスケーリング問題を対象に、富岳における実行時間を推定した。ここでは、メッセージ長により区間をレイテンシネック・スループットネックに分類した上で、レイテンシネック区間に関しては京 2304 ノード (8 プロセス/ノード) による通信時間の実測値をそのまま使用し、スループットネック区間に関しては京と富岳のネットワークバンド幅の比を乗じることで富岳における通信時間を推定した。

2. 非構造メッシュをグラフ分割手法により領域分割しているため、各小領域の間に計算負荷のばらつきが生じる。そのため、ウイークスケーリング問題実行時と、全ノード実行時の実行時間・性能に差が生じる。そこで、全プロセスの平均実行時間と全プロセス中の最大実行時間の比であるインバランス係数 α を、ウイークスケーリング問題の実行時間に対して乗じることで全ノード使用時の経過時間を推定した。要素数・節点数だけでなくキャッシュミス特性も小領域毎に変化するため、モデルパラメータからこのインバランス係数を見積もることは難しい。そこで、プロセスあたり問題サイズ、及び、プロセス数に対するインバランス係数を京を用いて実測することで、富岳におけるプロセス数に対するインバランス係数を見積もった。図 4.19 にインバランス係数の推定方法を、表 4.23 においてインバランス係数の推定結果を記す。

3. 最後に、3 タイムステップの推定値をターゲット問題 12 万ステップに対して線形外挿した。

なお、ノードマッピングに関してはいずれの計測においても 1 次元のノードマッピングを使用した。上記推定の結果、2019 年度までの推定で京全系比で 45 倍の性能となると予測された。

4.4.8 富岳で達成された性能倍率

上述したコデザインの成果を反映して富岳で達成された京に対する性能倍率及び消費電力を表 4.24 に示す。表より、対京比で 63 倍の性能となっていることがわかる。2019 年における性能推定値である 45 倍よりも高い性能が達成されていることがわかる。推定値よりも高い性能が出た理由として、インバランス係数が想定していた値よりも小さかったこと、及び、アーキテクチャに即

したさらなるチューニングを実施したことが挙げられる。性能推定時にはインバランス係数 (最大プロセス実行時間/平均プロセス実行時間の比) を 1.45 と見積もっていたが、実際には 1.24 程度であり、また、アーキテクチャに即したさらなるチューニングとして structure load/structure store など ACLE による詳細チューニングを実施してきた。また、性能推定時においてはソルバーの一部 (四面体一次要素行列ベクトル積カーネル) に省メモリ型計算法を使っていたが、メモリ容量に余裕があったためメモリをより多く使うが実行時間が短くなる方法に変更したためと考えられる。

表 4.24: GAMERA の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
GAMERA	63 倍	21 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

表 4.25: GAMERA の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
GAMERA	63 倍		58 倍		56 倍		52 倍	
	22 MW	23 MW	20 MW	21 MW	17 MW	17 MW	16 MW	16 MW

以下計測結果の詳細を示す。まず、アプリケーションのスケーラビリティを確認するため、ターゲット問題と近い計算ノード当たり問題規模の設定で、ウィークスケーリングモデルセット A-1 から A-9 までの性能を測定した（表 4.26)。288 ノードにおいて最小のモデル (A-1) は 4.13 s で実行できたのに対し、147456 ノードにおいてターゲット問題となるモデル A-ポスト京 (1.08 兆自由度) は 5.54 s (49.6 TFLOPS, FP64 ピーク性能比 9.97%) で実行できた（図 4.20)。このように、288 ノードから 147456 ノードまで高いウィークスケーリング性能が達成されたことが確認できた。なお図 4.20 では、コデザイン前の京用の as is コードを富岳で実行した場合の性能も示している。ターゲット問題においては 7.0 倍の高速化を達成しているなど、各 weak scaling モデルにおいてコデザインにおける検討により大幅な高速化が実現していることがわかる。本検討で開発したランダムアクセス主体の計算の効率的な計算方法は他のアプリケーションにも適用できると考えられ、それらのアプリケーションの性能向上につながると期待される。本検討の詳細は以下の文献に記載されている：Kohei Fujita, Kentaro Koyama, Kazuo Minami, Hikaru Inoue, Seiya Nishizawa, Miwako Tsuji, Tatsuo Nishiki, Tsuyoshi Ichimura, Muneo Hori, Lalith Maddegedara, High-fidelity nonlinear low-order unstructured implicit finite-element seismic simulation of important structures by accelerated element-by-element method, Journal of Computational Science, 2020, https://doi.org/10.1016/j.jocs.2020.1012.

4.4.9　さらなる高速化に向けた検討：時間並列アルゴリズム

さらなる高速化に向け、以下に示す時間並列アルゴリズムを開発・検討した。なお、本時間並列積分手法を利用せずとも目標性能を達成できたため、前節までの性能推定・性能計測においては以下の検討を反映していない結果を用いている。今後以下の時間並列積分の検討をポスト京上で進めることで、さらなる高速化を達成できる可能性がある。

表 4.26: GAMERA の富岳における性能測定に用いたモデルセット。A-ポスト京はターゲット問題、A-京はターゲット問題を京全系で計測した際のモデル分割を示す。

モデル	ノード数	MPI プロセス数	自由度	プロセスあたり自由度	四面体要素数
A-1	288	1152	1888114923	1638988	469413888
A-2	576	2304	3775253451	1638564	938821632
A-3	1152	4608	7548554667	1638141	1877630976
A-4	2304	9216	15095157099	1637929	3755249664
A-5	4608	18432	30186410283	1637718	7510474752
A-6	9216	36864	60368916651	1637612	15020924928
A-7	18432	73728	120730026027	1637505	30041800704
A-8	36864	147456	241452244779	1637452	60083552256
A-9	73728	294912	482888875563	1637399	120167006208
A-ポスト京	147456	589824	1086476549163	1842035	270375616512
A-京	82944	82944	1086476549163	13098916	270375616512

ポスト京などの SIMD 幅の大きい計算機では、連続データ領域への SIMD アクセスに比べてランダムアクセスのコストが相対的に大きくなる傾向にある。GAMERA においては EBE カーネルの u のランダム読み込みと f のランダム足しこみコストがカーネル全体において大きな比率を占めるようになり、この部分の高速化がアプリケーション全体の高速化において重要となる。そこで、時間方向に複数タイムステップを同時に解くことでランダムアクセスを減らす方法を開発した。

Algorithm 1 が従来の計算アルゴリズムである。空間方向に問題を分割して並列計算し、時間ステップ方向には逐次計算をしている。5 行目の反復ソルバーでは行列に対してベクトル一本を掛け合わせる SpMV (Sparse Matrix-Vector product, GAMERA では EBE カーネル部分に該当) が使われており、このカーネルがアプリケーション全体のコストの大部分を占めることとなる。

これに対して提案アルゴリズム (Algorithm 2) では、m 時間ステップをまとめて並列に計算する。8 行目の反復ソルバーにおいては、行列に m 本のベクトルを掛け合わせる GSpMV(Generalized SpMV) を使う。一反復当たりの演算数は SpMV の m 倍となるが、ソルバーで解いた先のステップの近似解を反復法ソルバーの高精度な初期解として使えるため、総反復数を減らすことができる。結果としてアプリケーション全体の総演算数は Algorithm 1 と同等となる。その一方で GSpMV カーネルにおいてのデータアクセスは m 個の時間方向のデータに関して連続となるため、ランダムアクセス回数を SpMV のおよそ $1/m$ に減らすことができる。これにより、GSpMV カーネルは SpMV に比べて高い効率で計算が可能となり、同程度の演算数を高効率なカーネルで計算できるようになるためにアプリケーション全体の実行時間が短縮される。

GAMERA は非線形問題に対応したマルチグリッドソルバーだが、2017 年度は一番単純な例として、線形問題用の 3×3 ブロックヤコビ前処理を使った共役勾配法ソルバーに Algorithm 2 を組み込んだ。四面体二次要素 3,820,608 個 (5,183,270 節点) の線形地盤における地震動伝播を 25 時間ステップ解く際に必要な合計反復数は Algorithm 1 の場合で 9,068 回だったのに対して、$m = 4$ とした Algorithm 2 を使うことで 1/2.56 にあたる 3,541 回に減少した。GSpMV カーネル一回の実行時間は SpMV カーネルの 1.2 倍程度で済むため、アプリケーション全体の実行時間において、Skylake-SP Xeon Gold cluster (512 bit SIMD) で 1.42 倍の高速化、 Haswell Xeon E5 cluster (256 bit SIMD) で 1.40 倍の高速化、Oakforest-PACS (512 bit SIMD) で 1.69 倍の高速化、京コンピュータ (128 bit SIMD) で 1.50 倍の高速化を達成した。本検討の詳細は以下の文献に記載されている：Kohei Fujita, Keisuke Katsushima, Tsuyoshi Ichimura, Masashi Horikoshi, Kengo Nakajima, Muneo Hori, Lalith Maddegedara, Wave Propagation Simulation of Complex Multi-

Algorithm 1 従来の計算アルゴリズム。時間方向に逐次計算する。

set $x_{-1} = 0$
for $(i = 0; i < n; i = i + 1)$ **do**
  guess $\bar{x}_i$ using standard predictor
  set b_i using x_{i-1}
  solve $x_i \Leftarrow A^{-1}b_i$ using initial solution $\bar{x}_i$
end for

Algorithm 2 時間方向の並列計算アルゴリズム。ϵ はソルバーの閾値、m は並列で計算する時間ステップ数である。

set $x_{-1} = 0$ and $\bar{x}_i (i = 0, ..., m-2)$
for $(i = 0; i < n; i = i + 1)$ **do**
  set b_i using x_{i-1}
  $\bar{b}_i = b_i$
  guess $\bar{x}_{i+m-1}$ using standard predictor
  while $(|A\bar{x}_i - b_i|/|b_i| > \epsilon)$ **do**
    guess $\bar{b}_j$ using $\bar{x}_{j-1} (j = i+1, .., i+m-1)$
    refine solution $\{\bar{x}_j \Leftarrow A^{-1}\bar{b}_j\}$ using initial solution $\bar{x}_j (j = i, ..., i+m-1)$
  end while
  $x_i = \bar{x}_i$
end for

Material Problems with Fast Low-Order Unstructured Finite-Element Meshing and Analysis, Proceedings of HPC Asia 2018, 2018.

2018 年度においてはこの計算手法を非線形問題に拡張したうえでマルチグリッドソルバーに適用した。図 4.21 に開発したカーネルの概略を示す。図のように、時間並列積分なしの場合 (図 4.16) では non-SIMD 計算としていた Part II-B 部においても幅 m の SIMD 計算ができるようになっている。これにより、GAMERA の実行性能の向上・実行時間の短縮につながると期待される。実際、SC14 版 GAMERA 比で、提案アルゴリズムを使った解析手法は京において 1.7 倍、512 bit SIMD 演算器を搭載する Intel Xeon Phi (Knights Landing) ベースの Oakforest-PACS においては 5.02 倍の高速化を達成している。ポスト京においても本アルゴリズムを使うことで SIMD 演算の比率が増え、演算効率の向上と実行時間の短縮が見込める。本検討の詳細は以下の文献に記載されている：Tsuyoshi Ichimura, Kohei Fujita, Masashi Horikoshi, Larry Meadows, Kengo Nakajima, Takuma Yamaguchi, Kentaro Koyama, Hikaru Inoue, Akira Naruse, Keisuke Katsushima, Muneo Hori, Maddegedara Lalith, A Fast Scalable Implicit Solver with Concentrated Computation for Nonlinear Time-evolution Problems on Low-order Unstructured Finite Elements, 32nd IEEE International Parallel and Distributed Processing Symposium, 2018.

```
1  !$OMP PARALLEL
2  !$OMP DO
3        ! clear global vector
4        do i=1,n
5          f(1,i1)=0.0
6          f(2,i1)=0.0
7          f(3,i1)=0.0
8        enddo
9  !$OMP END DO
10       do icolor=1,ncolor
11 !$OMP DO
12       ! for each thread
13       do iu=1,numberofthreads
14       ! block loop with blocksize NL
15         do ieo=nstart(iu,icolor),nend(iu,icolor),NL
16       ! compute BDBu                                  Part II-A
17         do ie=1,min(NL, nend(iu)-ieo+1)               (SIMD
18           cny1=cny(1,ieo+ie-1)                        computation)
19           cny2=cny(2,ieo+ie-1)
20           cny3=cny(3,ieo+ie-1)
21           cny4=cny(4,ieo+ie-1)
22           ue11=u(1,cny1)
23           ue21=u(2,cny1)
         ...
24           ue34=u(3,cny4)
25           xe11=coor(1,cny1)
26           xe21=coor(2,cny1)
         ...
27           xe34=coor(3,cny4)
28           ! compute BDBu using ue11~ue34 and xe11~xe34
29           BDBu11(ie)=...
30           BDBu21(ie)=...
         ...
31           BDBu34(ie)=...
32         enddo
33       ! add to temporary vector                      Part II-B
34         do ie=1,min(NL, nend(iu)-ieo+1)               (non-SIMD
35           cny1=cny(1,ieo+ie-1)                        computation)
36           cny2=cny(2,ieo+ie-1)
37           cny3=cny(3,ieo+ie-1)
38           cny4=cny(4,ieo+ie-1)
39           ft(1,cny1)=BDBu11(ie)+f(1,cny1)
40           ft(2,cny1)=BDBu21(ie)+f(2,cny1)
         ...
41           ft(3,cny4)=BDBu34(ie)+f(3,cny4)
42         enddo
43       enddo ! ieo
44     enddo ! iu
45 !$OMP END DO
46   enddo ! icolor
47 !$OMP END PARALLEL
```

図 4.16: EBE カーネルにおけるカラリングによるリストアクセスの削減

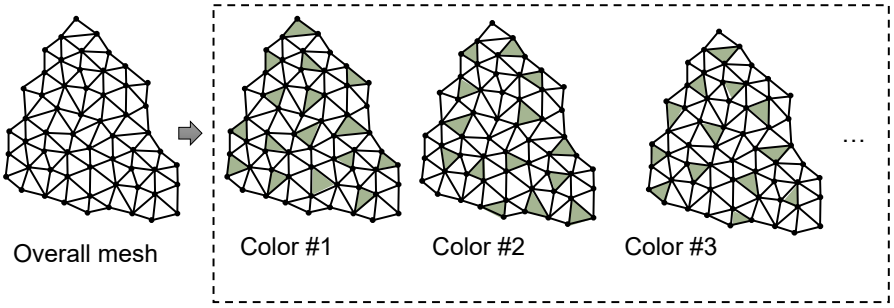


図 4.17: 一般的なカラリング手法

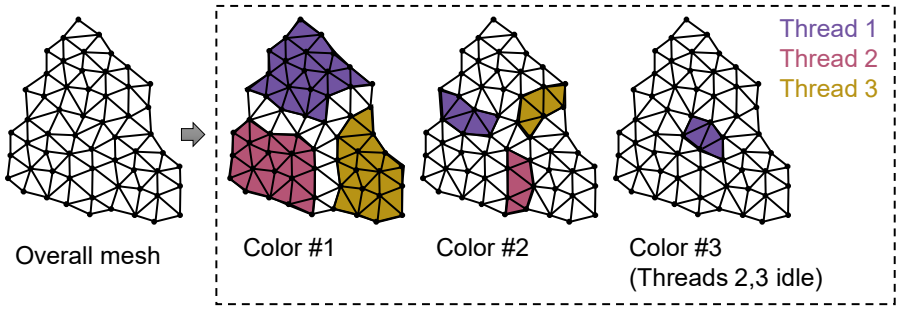


図 4.18: コデザインにおいて開発したカラリング手法

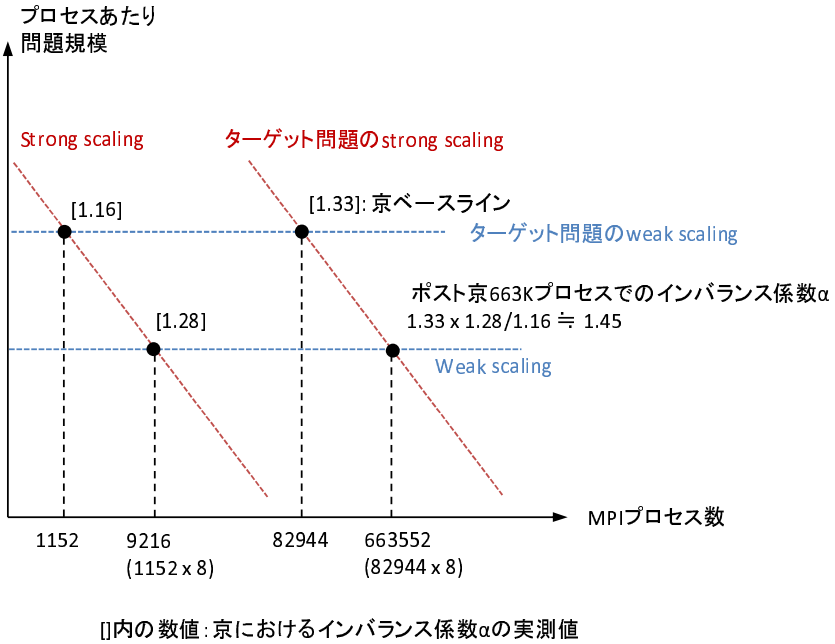


図 4.19: GAMERA におけるインバランス係数の推定方法

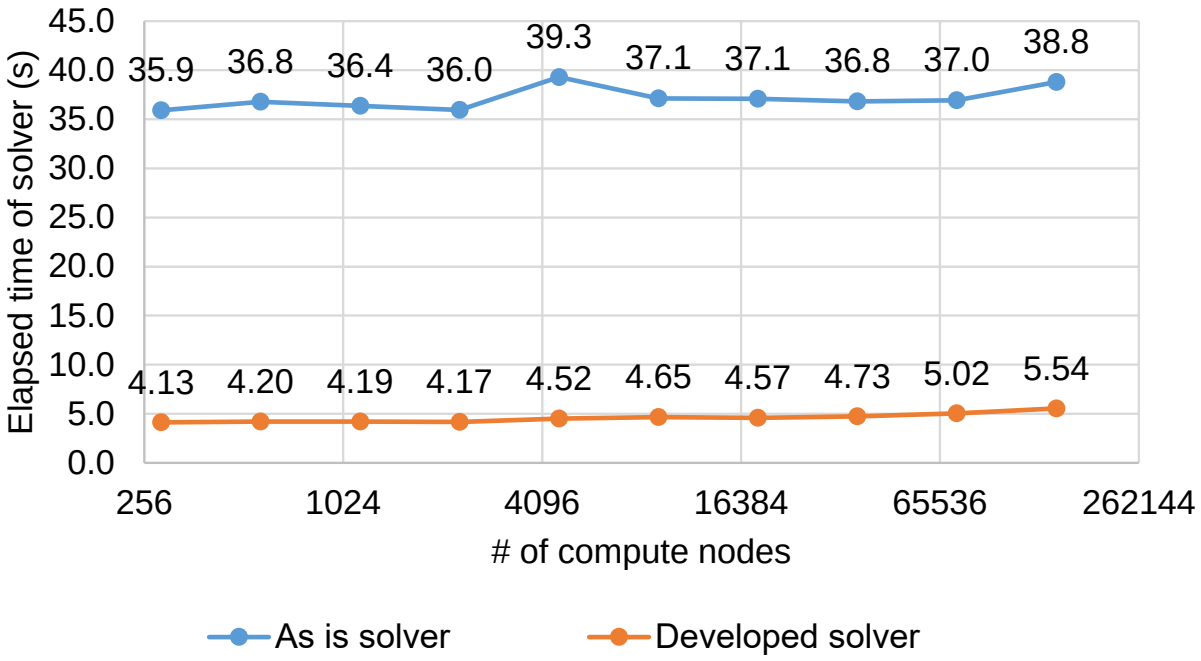


図 4.20: GAMERA の富岳におけるウィークスケーリング性能の測定結果

```
1  !$OMP PARALLEL DO
2      ! clear global vector
3      do i=1,n
4        do im=1,m
5          f(im,1,i)=0.0
6          f(im,2,i)=0.0
7          f(im,3,i)=0.0
8        enddo
9      enddo
10 !$OMP END PARALLEL DO
11     do icolor=1,ncolor ! for each color or element set
12 !$OMP PARALLEL DO
13     do iu=1, numberofthreads
14       ! block loop with blocksize NL/m
15       do ieo=npl(icolor,iu)+1,npl(icolor,iu+1),NL/m
16         ! load ue, xe
17         do ie=1,min(NL/m,npl(icolor,iu+1)-ieo+1)
18           cny1=cny(1,ieo+ie-1)
19           cny2=cny(2,ieo+ie-1)
20           cny3=cny(3,ieo+ie-1)
21           cny4=cny(4,ieo+ie-1)
22           do im=1,m
23             ue11(im+(ie-1)*m)=u(im,1,cny1)
24             ue21(im+(ie-1)*m)=u(im,2,cny1)
25         ...
26             ue34(im+(ie-1)*m)=u(im,3,cny4)
27             xe11(im+(ie-1)*m)=x(1,cny1)
28             xe21(im+(ie-1)*m)=x(2,cny1)
           ...
29             xe34(im+(ie-1)*m)=x(3,cny4)
30           enddo
31         enddo
32         ! compute BDBu
33         do i=1,NL
34           BDBu11(ie)=...
35           BDBu21(ie)=...
           ...
36           BDBu34(ie)=...
37         enddo
38         ! add to global vector
39         do ie=1,min(NL/m, npl(icolor,iu+1)-ieo+1)
40           cny1=cny(1,ieo+ie-1)
41           cny2=cny(2,ieo+ie-1)
42           cny3=cny(3,ieo+ie-1)
43           cny4=cny(4,ieo+ie-1)
44           do im=1,m
45             f(im,1,cny1)=BDBu11(im+(ie-1)*m)+f(im,1,cny1)
46             f(im,2,cny1)=BDBu21(im+(ie-1)*m)+f(im,2,cny1)
           ...
47             f(im,3,cny4)=BDBu34(im+(ie-1)*m)+f(im,3,cny4)
48           enddo
49         enddo
50       enddo ! ieo
51     enddo ! iu
52 !$OMP END PARALLEL DO
53     enddo ! icolor
```

SIMD (width=m) computation

SIMD computation

SIMD (width=m) computation

図 4.21: 時間並列積分による EBE カーネルにおける non-SIMD 計算の削減

4.5 重点課題 4 NICAM+LETKF のコデザイン

4.5.1 アプリケーションの特徴とコデザインの方針

NICAM+LETKF は全球高解像度気象モデル NICAM とアンサンブルデータ同化システム LETKF から構成され、地球大気の物理モデルシミュレーションと観測データを融合しながら時間発展を解く複合アプリケーションである。NICAM は構造格子系を用いたステンシル計算を代表するアプリケーションとして、主にメモリ・キャッシュの転送性能やレイテンシの評価、スレッド並列やスケジューリング、アウトオブオーダー実行の性能評価に寄与する。また同時に NICAM のステンシル計算以外の部分では、ループボディが大きく条件分岐が頻出するプログラムコードの最適化手法について知見を蓄積し、コンパイラの性能向上に寄与することが期待されている。LETKF については、主要演算部分では単一ノードに収まる小規模な固有値計算の性能評価に寄与し、同化システムのフレームワーク部分では大気モデルと大量のデータをやり取りする際のファイル I/O と大域通信性能の評価に寄与する。気象・気候モデルは数値流体計算を含むアプリケーションであるが、一般的にあまり高次精度の移流スキームや時間発展スキームは使われない傾向にある。これは水蒸気の相変化や放射による大気の加熱・冷却、地表面境界との熱・水蒸気の交換など大気の物理諸過程が 1 次精度程度しか持ち合わせていないため、流体計算を高精度にしたところで計算時間に見合うシミュレーション性能を得られないからである。そのため、気象・気候モデルのソースコードには大部分の計算時間と演算量を占める演算区間（ホットスポット）が存在しない。また、アプリケーションのソースコード量は数十から百万行以上に達し、たくさんのコンポーネントを実験設定に合わせて切り替えて使うという特徴がある。富岳での性能見積もりにおいても、ソースコードの一部をカーネル化して最適化と見積もりを行なっても、カーネルそれ自体が全体に占める割合は数%以下である。アプリケーション全体の性能見積もりを行なうためには、カーネルで得られた知見をそれ以外の区間にも適用することを前提に、代表性のあるカーネルの見積もり結果を全体にマッピングするという作業が必要になる。NICAM における演算区間の見積もりには、データ依存性や条件分岐の有無などの特徴が異なる構造格子ステンシルカーネルから 6 つ、またループボディの大きい諸物理過程のコンポーネントから 3 つのカーネルを切り出して評価・検討を行なった。詳細設計（2）では特に混合精度浮動小数点演算でのアプリケーション実行について、演算性能の評価とシミュレーション性能の両面からさらなる最適化と評価を行ない、性能電力予測ツールを用いた見積もりを更新した。詳細設計（3）では、性能見積もりマッピングを精緻化することで全体見積もりを更新し、上位ルーチンの構造変更によるメモリ転送量削減、時間積分方法の変更による演算量変化の評価、単プロセス・小問題で高速に動作する固有値ソルバの評価を行なった。詳細設計（4）では、諸物理過程カーネルの演算最適化を進めると同時に、富岳試作機でのカーネル演算性能評価を行なった。ソフトウェア調整（1）では、富岳試作機での NICAM フルアプリケーションの演算性能評価と総合的な最適化を行ない、アンサンブル数と観測データ数の増加に耐えうる LETKF のメモリ使用量削減を行なった。ソフトウェア調整（2）では、富岳実機での NICAM およびデータ同化システムのフルアプリケーションの演算性能評価と総合的な最適化を行ない、最終的な性能結果をまとめた。

以下に NICAM および LETKF のコデザインの具体的な内容を述べる。

4.5.2 NICAM におけるコデザイン

4.5.2.0.1 自動並列化から OpenMP への移行 複数の多重ループをもつカーネルでは、キャッシュブロッキングを促進する目的で中間配列の次元を積極的に削減し、外側のループを共有するようなコード最適化を行なった。この場合、自動並列化では共有される外側ループではなく、その内側の多重ループでスレッド並列を行なう必要がある。しかし、L2 キャッシュに載った状態で計算が行われ、SIMD 幅も京よりも増加していることから、内側ループの計算は高速化しており、多重ループそれぞれでスレッド並列を立ち上げていてはスレッド同期の時間コストが無視できなくなることがわかった。そこで自動並列化から OpenMP を利用する方法に変更した。OpenMP を用いることにより、キャッシュブロッキングを行なう外側ループでパラレルリージョンを設定し、内側にある複数の多重ループを可能な限り非同期で実行することが可能になる。これによりスレッドの同期処理の回数を削減し、高速化を達成した。

4.5.2.0.2 間接 SIMD ロードの抑止とモジュール配列の引数化 ステンシルカーネルの中には、間接 SIMD ロードが多量に発行され、L1 キャッシュのビジー率が高くなるために経過時間が伸びるものがあった。該当するコードではストライド SIMD ロードが発行されることを意図してサイズの小さいデータ次元を内側に配置していたが、実際には間接 SIMD ロードが発行されていた。この現象は主に最内のループ回転数がコンパイル時に予測できないことに起因していた。この問題を回避するには、コンパイル時に配列サイズを固定してしまうことが望ましい。しかしこの方法ではコードの柔軟性が失われてしまい、実験設定を変える毎にバイナリを別々に用意しなければいけない。allocatable なモジュール配列を用いていてもストライド SIMD ロードを発行させる最適化として、モジュール配列を use 文を通して利用せず、引数として渡すという方法を採用した。allocatable なモジュール配列は、コンパイル時にその配列がどのようなサイズで確保されるのか予測できないため、use 文を用いて引用した場合にストライド SIMD ロードを発行できない。しかし、この配列を引数として subroutine に渡した場合、部分的に配列のサイズ情報をコンパイラに知らせることが出来る。例えば、モジュール配列の最内データ次元のサイズが x,y,z 方向を表す 3 であるような場合、引数として配列を渡せば外側のループの回転数が未確定であっても最内が 3 であることは確定するので、ストライド SIMD ロードが発行可能になる。

4.5.2.0.3 ループボディが巨大で複雑なカーネルの最適化 詳細設計（2）ではコンパイラの最適化に関する特性について知見を得るため、NICAM の物理過程の中から、ループボディが巨大でこれまで最適化されていなかった区間を追加選定し、カーネル化を行なった。選定した区間は雲微物理過程を扱うコンポーネント NDW6 であり、これまで切り出し評価を行なってきた NSW6 よりも扱う変数が多く、より精緻化されたコンポーネントである。富岳時代において、この新しいコンポーネントの積極的な利用が見込まれている。切り出しカーネルの最適化の中で得られた知見は以下である。

- 巨大なループは出来る限り分割して、ループ内の命令数を抑える必要性がある
- 配列サイズや最適化範囲のスコープを分かりやすくし、コンパイラによる最適化を促すためには、主要なループの単位で更にサブルーチン化する方策が有効である
- Fortran の配列記述による書き方は、一般的に SIMD 化やソフトウェアパイプラインが適用されやすいという利点がある。さらに最適化を進めるには、そこから融合すべきループを適切に選択するのが最も高速化する

- 雲微物理過程の場合、各雲カテゴリごとに変数をまとめて手動でループ融合を行なうのが最も性能が高い結果となった

4.5.2.0.4　単精度実数を用いた場合の力学カーネル性能評価　単精度実数を用いる場合に、計算高速化に資する利点は大きく 2 つある。ひとつは要求される Byte/FLOP 比が小さくなることにより、特にメモリの転送性能で律速されている演算区間での高速化が見込めるということである。NICAM は演算密度が低くメモリバンド幅律速の区間が多いため、この効果は大きいと考えられる。もうひとつは単精度演算において最大 16SIMD で実行できる点である。これにより、演算待ちで律速される区間やメモリ・キャッシュビジー率にまだ余裕があるような区間において、より高速化が見込める。詳細設計（2）では FX100 上で、まず倍精度で力学カーネルのコード最適化を行ない、これを起点に単精度での演算性能を評価した。必要に応じて、単精度実行下でのコード最適化を行なった。さらに、FX100 単精度最適化済ソースコードを用いて性能電力予測ツール及びポスト京 CPU 性能シミュレータでの評価を行なった。表 4.27 に倍精度・単精度でのそれぞれ最適化前（ASIS）と最適化後（TUNE）の演算性能をまとめた図を示す。上段より、経過時間 [s]、倍精度ピーク性能比 [%]、メモリスループット [GB/s] を表す。以下、詳細について述べる。

表 4.27: NICAM 力学カーネルの倍精度・単精度での性能

	倍精度ASIS	単精度ASIS	倍精度TUNE	単精度TUNE
OPRT_diffusion	0.22 11.3% 14	0.14 17.3% 11	0.14 16.8% 22	0.07 35.6% 28
OPRT3D_divdamp	0.75 9.3% 103	0.49 14.1% 57	0.65 10.5% 120	0.38 18.2% 73
horizontal_adv_flux	0.07 8.9% 64	0.04 12.3% 49	0.05 12.2% 160	0.03 18.2% 126
horizontal_adv_limiter	8.15 6.9% 76	4.84 11.1% 64	6.46 8.6% 99	4.06 13.2% 80
vertical_adv_limiter	4.06 14.0% 120	1.88 27.9% 98	1.86 29.2% 85	1.79 28.0% 44
vi_rhow_solver	0.42 6.4% 95	0.25 9.0% 74	0.23 11.8% 115	0.11 19.7% 97

4.5.2.0.5　力学カーネルのコード最適化　詳細設計（2）では構造格子ステンシルカーネル 6 本について、基本設計及び詳細設計（1）で行なったコード最適化をさらに進めた。効果のあった最適化を大まかに分類すると次の 2 つが挙げられる。ひとつは中間配列の次元削減によるデータ転送量の削減である。出来るだけ L2 キャッシュに載るよう配列サイズを削減することでメモリへのアクセスを減らし、高速化を行なった。もうひとつはスレッド同期の回数削減とプリフェッチチューニングである。スレッド並列を自動並列化ではなく OpenMP を用いて実装し、パラレルリージョンの拡大によって同期にかかるオーバーヘッド時間を削減した。また、ソフトウェアプリフェッチ

命令を挿入することで、ハードウェアプリフェッチが効かないケースを回避し、データ転送の待ち時間を削減した。これらの最適化では演算量やデータ転送量は変化せず、レイテンシが削減される事によって高速化された。表 4.27 において、倍精度 ASIS と倍精度 TUNE の間でメモリスループットが減少しているものは前者の最適化、増加しているものは後者の最適化による効果が大きかったと結論づけることが出来る。

4.5.2.0.6　単精度化した力学カーネルの性能評価　引き続き、力学カーネル 6 本の単精度計算による性能評価を行なった。このうち OPRT_diffusion では、単精度化によるメモリフットプリントの減少に伴い、スレッド並列軸をより外側に移動させてもメモリ律速にならなくなったので、倍精度とは異なるコード最適化を行なうことで高速化した。また horizontal_adv_limiter では、追加したプリフェッチ命令が特定のループで性能劣化させる要因となっていたことを発見したため、これを削除した。この劣化はメモリ律速であるにもかかわらず、L1 キャッシュにのみソフトウェアプリフェッチ命令を発行させていたことが原因であり、単精度計算でその影響がより明確に現れた。単精度計算の倍精度計算に対する高速化倍率は 6 本の平均で 1.7 倍程度であり、これは ASIS 同士の比較でも TUNE 同士の比較でも 6 本平均では同程度であった。しかしそれぞれのカーネルの高速化傾向は異なり、また最適化前後でも異なる。メモリ律速傾向が強いカーネルは約 1.7 倍程度の高速化であり、これはメモリ転送量の削減による効果が主であった。一方でキャッシュ律速傾向にあるカーネルは 2 倍程度の高速化を達成したもの（OPRT_diffusion）とほとんど高速化しないもの（vertical_adv_limiter）があった。これは単精度 8SIMD が適用されたかされなかったかの違いが大きく影響していた。FX100 ではループ内に除算や max、min、sign、abs 等の組み込み関数が存在する場合に、単精度 8SIMD ではなく単精度 4SIMD が適用される。またメモリアクセスが連続ではないロード・ストアについても単精度 8SIMD が適用されない。そのため、FX100 を用いた評価の時点では単精度化に伴う SIMD での同時実行数倍増の効果が小さく見積もられている。

4.5.2.0.7　混合精度計算におけるアプリケーション全体への性能マッピングの見直し　最適化を施し、単精度で計算したカーネルの多くは、FX100 上ではまだメモリバンド幅律速であったが、富岳性能電力予測ツールでの見積もりでは L1・L2 キャッシュのビジー率がほとんどメモリのビジー率と同等になり、明確にメモリバンド幅律速とはいえない性質を持つようになった。またキャッシュ律速であったカーネルは演算律速へと移行したものもあった。これらカーネルの特徴の違いを表 4.28 にまとめる。最適化と単精度化によるカーネルの性質の変化は、アプリケーション全体の性能を見積もる上で非常に重要な変化である。これまでの見積もりで代表カーネルとして用いてきた OPRT3D_divdamp は、メモリ律速からキャッシュ律速のカーネルへと変化した。カーネル以外の区間の多くは、演算密度が低くメモリ律速傾向にあるため、OPRT3D_divdamp は代表カーネルとして適切ではなくなった。そこでこれら 6 本のカーネルの性質より、新たに horizontal_adv_flux を代表カーネルとして選定し直し、全体性能のマッピングに用いることとした。

4.5.2.0.8　ポスト京 CPU 性能シミュレータを用いた力学カーネルの性能評価　更なる見積もり精度の向上に向けて、力学カーネル 6 本のポスト京 CPU 性能シミュレータでの測定・評価を開始した。その中で、FX100 での測定を元にした性能電力予測ツールとシミュレータとの間に大きな見積もり性能の隔たりが見られるカーネルがあった。シミュレータでの性能が向上したケースは主に前述の単精度 SIMD に関連していた。すなわち、シミュレータでは FX100 よりも多くのループに同時実行数が倍増する単精度 SIMD が適用され、高速化した場合があった。一方で、FX100 と富岳の間でストライドアクセス幅が異なることに起因して、シミュレータでの性能が劣化する

表 4.28: 最適化後の NICAM 力学カーネルの性質のまとめ

	メモリ律速	キャッシュ律速	ポスト京でキャッシュ律速に変わる	ポスト京で演算律速に変わる	FX100では単精度8SIMD適用不可	要求Byte/FLOP（倍精度）	要求Byte/FLOP（単精度）
OPRT_diffusion		●				0.34	0.21
OPRT3D_divdamp	●		●		●	2.99	1.07
horizontal_adv_flux	●				●	3.45	1.82
horizontal_adv_limiter	●		●		●	3.02	1.61
vertical_adv_limiter		●		●	●	0.77	0.42
vi_rhow_solver	●					2.57	1.29

ケースもあった。この場合、FX100 のストライドアクセス幅に合わせたコード最適化で達成した高速化は富岳では見込めないことになり、見積もりを過大評価すると考えられる。同様にレジスタ数の違いに起因するコード最適化方法の違いも、見積もり精度に影響すると予想される。そこで、富岳の CPU パラメータを想定してハードウェア資源に制限をかけた FX100 コンパイラを整備し、最適化作業に役立てるよう提案を行なった。

詳細設計（３）ではシミュレータ結果改善のためのチューニングを行なった。具体的に効果のあったものは、配列次元順の変更による連続データアクセスの促進 (OPRT3D_divdamp, horizontal_adv_flux, horizontal_adv_limiter)、ループ分割によるソフトウェアパイプライニングの促進 (OPRT3D_divdamp, horizontal_adv_flux)、FX100 でのチューニングで追加したプリフェッチオプションの指定を削除 (OPRT_diffusion)、中間配列の局所化とループブロッキング (horizontal_adv_flux) 等であった。これらに加え、コンパイラのアドレッシングの改善も適用された (OPRT_diffusion)。また、レジスタ数が 10%多いと仮定してスケジューリングを行うオプションを用いることで高速化する場合もあった (vertical_adv_limiter)。vi_rhow_solver に関しては FX100 の時点でも単精度 SIMD が適用されており、ストライド幅などの問題も無いがシミュレータの結果の方が 10%程度高速であった。これはツールで推定されたメモリ・キャッシュビジー率がシミュレータよりも大きいことが要因であった。最終的に全ての力学カーネルについて、ポスト京 CPU 性能シミュレータでの経過時間が性能電力予測ツールで推定した経過時間を上回る結果を得た。

4.5.2.0.9　モデル全体に関わる配列データ構造、ループ構造に関する考察　NICAM はもともとベクトル型コンピュータでの性能最適化を行なってきた過去があり、ベクトル長を長くとるために最内にある水平 2 次元のデータを 1 次元にマージしていた。この (ij,k) というデータ構造は京での最適化においても有効に働き、連続メモリアクセス下での十分長いループの演算は多くの箇所において最適な性能を示した。しかし、鉛直軸 k に依存関係のあるループでは、SIMD とソフトウェアパイプライニングをかける軸とスレッド並列をかける軸を、鉛直軸 k 以外に用意する必要がある。この場合、ij を分離して (i,k,j) とした方がより最適化が進むことが多い。ただしこの変更は最内ループの回転数が分割されて小さくなってしまうため、単精度化によって SIMD による同時演算数が 16 個になった場合に、並列度が足りなくなって性能が劣化するケースもある。特に分割した i,j のサイズがそれぞれ 50 回転を切るような場合、ソフトウェアパイプラインが適用されなくなり、演算待ちの時間が現れる傾向にあった。また、ij を結合してループを回すことで、ハードウェアプリフェッチがより有効に働くケースも存在する（特に、一部の袖領域格子での計算をスキップしない場合）。このように、配列のデータ次元とループネストはカーネルによって最適な順番が異なっており、6 つの力学カーネルの最適化の過程においていくつかの事例を抽出することで、一般的な最適化方針を立てる上での知見を得た。巨大なアプリケーションコードの中でデータ次元を頻繁に入れ替えるとデータコピーの大きなコストが生じるため、区間毎に最適なコード

チューニングを採用することは必ずしもできない。データ構造を変更した場合の全体性能の見積もりをどう予測するかについては将来の課題である。その際に根拠となるカーネルの見積もりは、新たなアーキテクチャの場合新しい機能を利用可能な状態で行うべきであり、富岳で行ったようなシミュレータを用いた評価は必要不可欠である。

4.5.2.0.10　単精度実数を用いた場合のシミュレーション性能評価　単精度を用いたシミュレーションを行なうには、シミュレーション結果が単精度浮動小数点を用いても大きく劣化しないことを示す必要がある。シミュレーション結果の評価には大気の力学コアコンポーネントの性能評価に広く用いられている傾圧不安定波のテストケースを用いた。詳細設計（1）では比較的格子の水平解像度が粗い 224km から 56km メッシュでの実験を行ない、単精度と倍精度での結果比較を行なった。224km メッシュでの実験ではすべて倍精度とすべて単精度で行なった実験結果にほとんど差が見られなかったが、56km メッシュ解像度の結果では明らかに不自然な流れ場が計算されることがわかった。計算結果に大きな影響を与える区間を特定するため、部分的な倍精度化を行なって複数の実験を行なったところ、計算初期化部分での計算を一部倍精度にすることで、シミュレーション結果の精度が高く保たれることがわかった。計算結果を劣化させた原因は主要なステンシル係数で用いる係数の値の精度低下であり、これは球面上の 2 点間の距離計算やメトリクス項の計算精度が単精度化によって著しく劣化したことの影響を受けていた。係数準備のための計算は初期化部分でのみ 1 回行われる。この時の計算に用いる変数の精度を高く保つことで、係数そのものは単精度であっても結果は劣化しないことが明らかになった。

詳細設計（2）ではターゲット解像度である水平 3.5km メッシュまで実験を行なった。結果として、先行研究で報告されている複数モデル間の結果の差異と比べて、同一モデルの倍精度・単精度の結果の差異はリファレンス実験からのノルムで 1 桁以上小さいことが示された。しかし、最高解像度である 3.5km メッシュでは計算が進むにつれて 7km までには見えていなかった倍精度・単精度間の差が大気擾乱に沿ったパターンとして出現することが明らかになった。結果を図 4.22 に示す。図は 3.5km メッシュ傾圧波理想実験における 9 日後の倍精度実験ー単精度実験の差を表し、上から、コントロール実験、気圧傾度力を倍精度で計算した場合、数値粘性を倍精度で計算した場合、単精度実験の代わりに倍精度で毎ステップ精度丸め込みフィルターを適用した実験を用いた場合の結果である。このノイズパターンは十分小さいため現時点でも十分シミュレーションに耐えうるが、ここでは生成要因をより詳細に探るために、部分的に気圧傾度力や数値粘性の計算部分を倍精度に高めて実験を行なった。しかし、これらの区間を倍精度にしてもシミュレーション精度は改善しないことが明らかになった。一方で、すべて倍精度で計算しメインループの最後に時間発展する変数の浮動小数点精度を単精度に落としてすぐ倍精度に戻す操作（丸め込みフィルター）を適用したところ、倍精度シミュレーションとの差は顕著に大きくなることが明らかになった。

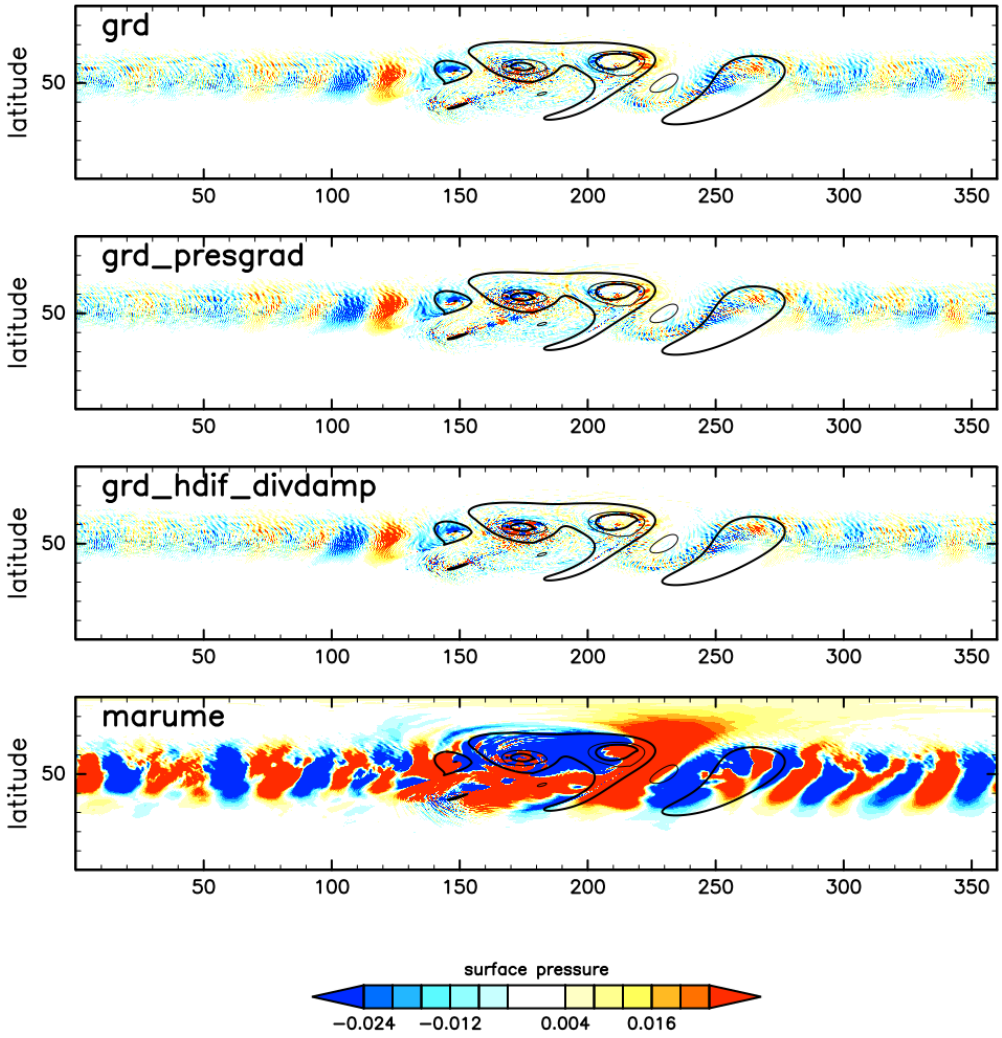


図 4.22: NICAM3.5km メッシュ傾圧波理想実験における倍精度ー単精度実験の差

4.5.2.0.11　カーネル評価結果に基づく性能マッピング　NICAM の性能見積もりの不確定性は、カーネルの性能測定の精度と、それをアプリケーション全体にマッピングした時の精度によって決まる。表 4.29 は FX100 で実測された各カーネル区間の代表性を示したもので、左よりそれぞれ実行時間、演算量、データ転送量が NICAM 全体の何%を占めるかを表す。表より、力学カーネル 6 本は足し合わせても全体実行時間の 10%程度しかないことがわかる。

表 4.29: NICAM カーネル区間のアプリケーション全体に対する代表性

全体に対する比率	実行時間	演算量	データ転送量
OPRT_diffusion	2%	7%	0%
OPRT3D_divdamp	3%	4%	2%
vi_rhow_solver	1%	1%	1%
horizontal_adv_flux	1%	1%	0%
horizontal_adv_limiter	4%	3%	4%
vertical_adv_limiter	1%	3%	1%
Cloud_Microphysics	20%	17%	14%
Radiation	13%	19%	10%
カーネル区間以外 (Horiz.Adv.Fluxを適用)	32%	31%	26%
カーネル区間以外 (Streamlikeを適用)	24%	15%	40%

詳細設計（3）ではまず、全体計測の基礎データをとるためのフルアプリケーションの更新を行なった。これまでのカーネル最適化の知見をカーネル区間以外の部分にも適用し、力学コード全体を更新した。また計測条件の見直しを行い、基本設計では雲微物理過程の call 回数を半分に減らすことを見込んでいたものを元に戻した。これは雲微物理過程の時間幅刻みの変更について、まだシミュレーション性能の観点からの検証が不十分であるためである。この更新されたソースコードと実験設定の元に、これまでよりも計測区間を詳細に分割し、FX100 を用いて倍精度での全体計測を行なった。その結果、カーネル区間以外の計測区間の中に、カーネルと似たような演算量・データ転送量を示すものだけでなく、より単純で STREAM 計算に近いものが少なからず存在することが明瞭になった。この STREAM 的区間は演算数が小さく最適化を施すような余地が少ししかないため、見積もりにおいて異なる性質を示す。この影響を評価するために、FX100 でカーネルを全体にマッピングして積み上げた実行時間と、実測時間との比較を行った。マッピングの方法は以下の通りである。

- カーネル計測区間そのものは、カーネルの性能見積もり結果を用いる

- カーネル計測区間以外は、代表カーネルとして採用した horizontal_adv_flux カーネルの性能見積もり結果を代用する
- データ転送量を特徴量とし、代表カーネルの実行時間とデータ転送量の関係を用いて実行時間を見積もる

上記の方法でマッピングした計測時間は、実測よりもおよそ 37%長い結果となった。このずれは大部分が STREAM 的区間の見積もりを長く見積もってしまうことによる。このままでは見積もりの不確定性が大きいため、新たに NICAM のソースコードを参考に、STREAM 的区間を代表するようなデータアクセスと演算パターンを持った streamlike カーネルを作成し、ポスト京 CPU 性能シミュレータで最適化と測定を行なった。この結果を用いて、マッピングのルールを以下のように修正した。

- カーネル計測区間以外のうち、FX100 倍精度実測時にメモリ転送性能が 120GB/s 以上の区間に対して、streamlike カーネルの性能見積もり結果を代用する。メモリ転送性能が 120GB/s 以下の区間に対しては、horizontal_adv_flux カーネルの性能見積もり結果を代用する

適用するカーネルを切り替えるしきい値として 120GB/s を採用した理由は、実測した NICAM の各区間について、概ねこの値を超える区間がメモリ転送量と浮動小数点数演算量の観点で streamlike カーネルに近い（要求 B/F が低い）性質を持っているためである。修正されたマッピング方法を用いて推定した計測時間は、実測よりも 1.8%長い結果となり、これまでよりも精度の高い見積もりを得た。表 4.30 に、基本設計から詳細設計（4）に至るまでの NICAM の計算時間見積もり内訳の推移について示す。力学過程が高速化した理由はマッピングの改善と、カーネルの知見を反映したことによるリファクタリングの成果である。雲微物理過程（Cloud_Microphysics）の見積もり時間が大きくなった理由は、上述した雲微物理過程の時間幅刻み変更の過程を一旦元に戻したためである。放射過程（Radiation）については、ここ 2 年の精緻化によりカーネルの演算量が増えたことに起因する。境界層乱流過程（BoundaryLayer）及びその他の物理過程は詳細設計（2）の見積もりと比べて大幅に時間が増えている。これは詳細設計（2）において、これらの区間の経過時間を代表カーネルの経過時間向上率でスケールする方法をとっていたことが原因であった。実際にはこれらの区間は代表カーネルほど最適化される余地も演算量もないため、高速化率を過大に評価していた。マッピング方式の見直しの中でこれらの区間をデータ転送量によるスケーリングに変更したことで、より現実的で精度の高い見積もりとなった。詳細設計（3）から詳細設計（4）で高速化した部分は、この期間において行われた最適化や富岳試作機での評価結果を反映した結果である。ただし、詳細設計（4）ではノード分割数の見直しによってプロセスあたりの問題サイズが増加していることに注意されたい。ソフトウェア調整 (1) での見積もりは、富岳試作機でのフルアプリ実測結果と通信性能評価が反映されている。ソフトウェア調整 (2) では富岳実機での実測を行った結果である。実機での計測では実際に 256 メンバーの同時実行を行った結果、メンバー毎に異なる演算量を反映して、一部の区間において計算時間が増加した。そのため、直前の見積もり結果よりも総経過時間が大きくなる結果となった。

4.5.2.0.12　コード構造の見直しによるメモリ転送量削減の評価　NICAM の物理過程は袖通信を伴わないため、物理過程のドライバモジュール内では各変数を格納するコンテナモジュールから一部の袖を取り除いた形で配列を受け取り、それぞれの過程を解く下位のサブルーチンに配列を受け渡している。この袖を取り除く操作は単純なメモリコピーであるため、余計な経過時間がかかっている。そこで、メモリコピーを廃止し、袖領域を外さないままポインタで配列を受け渡

表 4.30: NICAM の実行時間見積もりとその内訳

	京	基本設計（2）	詳細設計（2）	詳細設計（3）	詳細設計(4)	S調整(1)	S調整(2)
ステップ数	2,700	2,700	2,700	2,700	2,700	2,700	2,700
MPIプロセスあたり水平格子数	16,900	16,900	16,900	16,900	21,780	21,780	21,780
同時実行アンサンブルメンバー数	32	256	256	256	256	256	256
実行段数	32	4	4	4	4	4	4
実行時間内訳[sec]							
力学過程 Pre_Post	10,661			230	201	188	231
力学過程 Large_Step	40,475	7,702	5,194	595	548	566	845
力学過程 Small_Step	105,979			2,470	1,884	2,173	2,362
力学過程 Tracer_Advection	52,338	2,143	1,548	1,296	1,040	933	1,073
物理過程 Cloud_Microphysics	35,887	771	816	1,557	1,354	1,483	1,257
物理過程 Radiation	38,705	1,249	1,354	1,605	1,591	1,100	1,158
物理過程 BoundaryLayer	31,152	2,779	232	1,133	754	728	706
物理過程 その他	60,666		470	623	395	926	1,449
力学過程演算部分	209,454	9,845	6,742	4,592	3,673	3,860	4,511
物理過程演算部分	1,66,409	4,799	2,872	4,918	4,094	4,237	4,570
通信部分	1,243	283	291	114	229	116	-
NICAM全体	377,106	14,927	9,905	9,510	7,995	8,213	9,081

す方式を実装し、評価を行なった。その結果、コピーのコストよりもその後の計算で袖領域の分だけ増えたデータ転送、演算量が無視できず、高速化しないことが明らかになった。そのためこの変更については採用を見送った。ただし、この変更に際してループ構造やデータ構造を見直したことによるメモリ消費量の削減や高速化については一定の効果が見られた。

4.5.2.0.13　力学コアの時間積分方法を変えた場合の経過時間・シミュレーション性能の評価　時間積分方法の組み合わせの見直しは時間刻み幅を伸ばしたり、演算量や通信回数を削減することで実行時間短縮に有効であるが、一方で強引な組み合わせは計算不安定を引き起こし、正常なシミュレーションが出来なくなる危険性がある。NICAM の力学過程の時間積分方法について、Runge-Kutta の段数と時間刻み幅の変更、また速いモードの力学を解くための fractional step の廃止と時間刻み幅の変更を行なった場合について評価を行なった。これらの変更の組み合わせによるパラメータスイープを行なった結果、モデルが正常に長期積分できる範囲内では、現行の組み合わせよりも高速化するものはなかった。ソフトウェア調整 (2) では、時間積分方法は変えないまま、計算不安定を起こさずに計算を継続できる時間刻み幅を調査した結果、水平 3.5km メッシュのシミュレーションにおいて 6[sec] から 8[sec] まで延ばすことが可能であった。ただし、最終性能評価の際にはこの時間刻み幅は用いなかった。

4.5.2.0.14　物理カーネルの性能評価　詳細設計（3）では、物理過程カーネルのコードを更新し、改めて性能評価を行なった。物理過程のカーネルは行数が多く、全体をポスト京 CPU 性能シミュレータにかけることはできなかったため、性能評価では主要な部分を抽出して性能電力予測ツールと見積もりに差がないかを詳細に解析した。更新されたコードは全面的なリファクタリングを施し、コード可搬性を高めるために自動並列化から OpenMP でのスレッド並列化に変更した。また、ループ内のローカル配列のメモリフットプリントが大きい場合は、配列サイズを削減するためにループブロッキングを適用した。

まず、FX100 を用いてスレッドスケーラビリティの調査を行なった。自動並列化と OpenMP での並列化を比較し、自動並列化でより高速化している部分を改善した。具体的にはループの一重化 ($omp collapse) による高速化や、逆にループの一重化によって性能劣化していた部分を改善した。この一重化するループの範囲によって性能が劣化する問題への対応については、システムへの実装要請項目として要請を行った。

シミュレータを用いた計測では、多くの区間でツール推定結果よりも短い実行時間を得た。これは力学カーネルの時と同じく単精度 SIMD が適用されるループが増えたことによる。加えて富岳は SIMD 幅の違いによってキャッシュアクセスレイテンシが短くなる場合があり、これが高速化に寄与していた。シミュレータ評価結果を踏まえたコード最適化では、力学コードの最適化の時と同じく、ループ内の命令数を少なく抑えるためのループ分割が有効であった。特に雲微物理過程で最も行数の長いループでは、ソフトウェアパイプライニングが有効になるまでループ分割を進め、最内ループを 54 に分割することで高速化を計った。このループの一部を用いて、さらに最適な分割数について調査を行なったところ、レジスタ不足が解消するまで分割を進めた場合が最も高速化した。さらに分割を進めると、必要となる一時配列の増加によりメモリアクセス待ちが増加し性能は劣化した。この知見はコンパイラ開発へフィードバックされ、コンパイラによる自動ループ分割機能が強化された。これは富岳システムソフトウェアコデザインにおける大きな成果のひとつである。

詳細設計（４）では、具体的な最適化をさらに進め、中間配列の次元削減や OpenMP を用いたスレッド並列の最適化を行った。特に鉛直層のループごとにサブルーチンを呼び出し、水平方向でしかスレッド並列を適用できていない部分については、多数の omp parallel do が起動され、スレッドの fork/join のオーバヘッドやキャッシュアクセス待ちのばらつきに起因するスレッドインバランスが蓄積し、経過時間が伸びてしまう傾向にあった。これを解決するために並列化領域を拡大し、スレッドの fork/join の回数を減らすような最適化を行った。

ソフトウェア調整 (1) では詳細設計（４）で問題となった、粒度が細かくスレッド並列をサブルーチンの外で行わざるを得ないような区間のリファクタリングを行った。また、主要な３つの物理過程コンポーネントの単精度化を行ない、それぞれ 1.6 倍程度の高速化を達成した。ただし、単精度化が行われていないその他のコンポーネントとの間で新たに配列の詰替え時間が発生した。単精度化を行う過程では計算結果が丸め誤差よりも大きく変化しないよう確認しながら進める必要があるため、単精度化区間を拡げるには多くの作業が必要となる。例えば放射過程では、exp の計算と引き算の組み合わせにより、単精度ではどうしても結果が大きく変わってしまう演算があったため、一部のスカラー変数は倍精度で固定された。

4.5.2.0.15　富岳試作機を用いた力学カーネルの性能評価　力学カーネルおよび物理カーネルの測定を富岳試作機上で行ない、これまでの見積もりとの差異とその原因について調査を行った。力学カーネルでは OPRT_diffution の実行時間が 70 %程度増加したが、これはコンパイラがループ分割を行ったことに起因しており、ループ分割を抑制することで対処した。また、素カーネルでの調査より、キャッシュミスのデマンドミス率が高い場合にポスト京 CPU 性能シミュレータと実機の性能に差が現れる（実機の方が遅くなる）ことが明らかになった。NICAM のカーネルはデマンドミス率が 30%を越えるものがいくつかあり、そのためポスト京 CPU 性能シミュレータよりも性能が１割程度劣化する傾向にあった。ハードウェアプリフェッチ距離の調整によって改善が見込めるものも見られたが、一部は FX100 上での実行の時点でデマンドミス率が高いため、これを改善するチューニングを進めた。

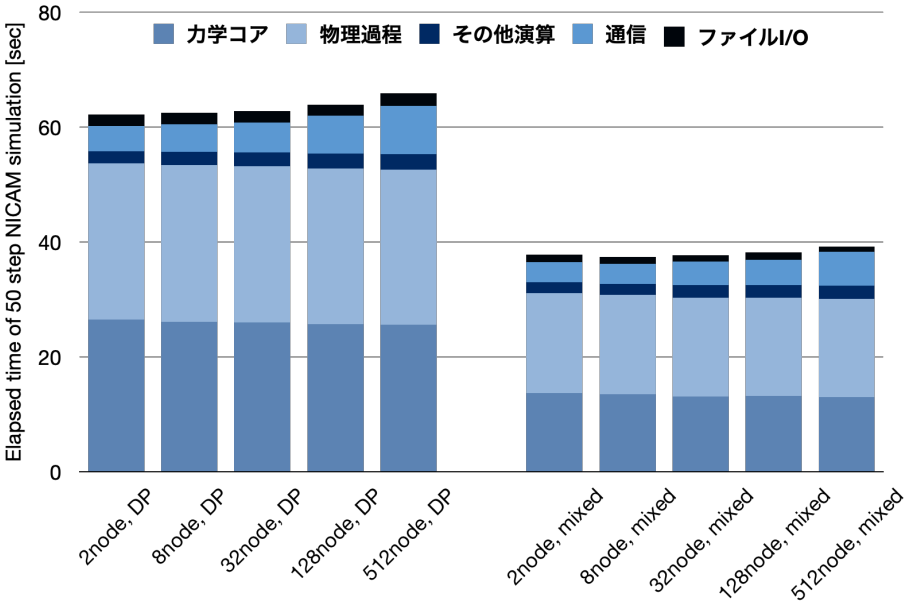
4.5.2.0.16　富岳試作機を用いたフルアプリケーションの性能評価　ソフトウェア調整 (1) では詳細設計（4）までに行われたカーネル最適化コードおよび最適化の知見を NICAM アプリケーションコード全体に適用し、富岳試作機上でのフルアプリケーションの実行を行い、性能電力予測ツールやポスト京 CPU 性能シミュレータ結果との比較を行った。評価開始時点では富岳試作機上でかなり遅い結果となったが、全体から該当区間へと細分化していく評価の流れをこれまでに確立してきたこともあり、低速化の原因となる区間を素早く特定し、最適化阻害要因をこれまでの知見に照らし合わせて評価し、短期間でのフルアプリケーションを進めた。その結果、性能電力予測ツールでの全体見積もり時間に対してのズレは 10 %程度まで抑えられた（富岳試作機での結果の方が低速)。伸びた時間の大部分はカーネルでは評価しておらず、またリファクタリングが進んでいない物理過程のコンポーネントに由来しており、これらの区間では性能マッピングでの高速化率を過大評価していた。

4.5.2.0.17　富岳実機を用いた NICAM の性能評価　ソフトウェア調整 (2) では、富岳実機を用いた NICAM 単体でのウィークスケーリング性能評価を行った。図 4.23 に 1 メンバーでの実行時間を示す。鉛直層数は 94 層、時間刻みは 6 秒に統一し、倍精度と混合精度でそれぞれ 50 ステップ計算を行った。ウィークスケーリングは 2 ノード 8 プロセスから 512 ノード 2048 プロセスまで 4 倍ずつノードを増加させて行い、ノードあたりの格子点数を一定にするために水平解像度を 56km, 28km, 14km, 7km, 3.5km と変化させている。結果より、倍精度、混合精度の両方において良好なウィークスケーリング性能を得た。特に演算部分はノード平均の経過時間でみるとほぼ一定となっている。通信区間だけが、ノード数が増えるにつれて経過時間が伸びる傾向にあった。この要因はノード間の通信時間のインバランスと、演算時間のインバランスに起因する。待ち時間に関しては、ノードごとの演算量の違いのばらつきが、使用ノード数を増やすにつれて大きくなったことと関連していると考えられる。雲の相変化などは、地球上の地点によって計算量が大きく変動する。使用ノード数を増やす際に水平解像度を高めていった結果、雲が多く計算時間の長い格子点群と、雲がなく計算時間が短い格子点群とのコントラストが強くなり、演算時間のインバランスが拡大したと考えられる。ファイル IO 時間は非常によくスケールしている。この計算はファイル入出力を第一階層の SSD を用いて行っており、第二階層の GFS へ出力した場合はノード間の IO 時間に大きなばらつきが発生し、さらに大きなインバランスとなって現れた。ここでは図示しないが、256 メンバーを同時実行する場合の実行時間の評価も行った。256 メンバーはそれぞれ異なる初期値から計算されるため、計算時間にはばらつきが表れる。このメンバー間のインバランスにより、経過時間は 1 メンバーの時よりも 10 %程度増加する傾向にあった。

4.5.3　LETKF におけるコデザイン

4.5.3.0.1　複数の固有値計算ライブラリの評価　LETKF の主要演算区間について、経過時間の大部分を占める固有値の計算にはライブラリを用いている。これまでの評価によって、1000 を越える大きなアンサンブル数（固有値ベクトルのサイズに等しい）の場合は EigenExa を用いた計算が高速であることが示されているが、より小さいベクトルサイズではあまり高速化しない結果となった。これについて数値計算ライブラリ WG と連携し、問題サイズによって最適なライブラリを選択できるよう、複数のライブラリを用いた性能比較を京を用いて行なった。用いたライブラリは EISPACK、EigenExa、LINPACK と京に最適化した特製版 Kevd の 4 つである。このうち Kevd の結果が、ベクトル長 10 から 1000 の範囲で最も性能が高かった。このライブラリは京の L1 キャッシュライン幅やレジスタの大きさに特化して最適化されたものである。本調査によっ

図 4.23: 富岳での NICAM のウィークスケーリング性能計測結果



て、富岳においてノード間通信を伴わない、小さな問題サイズを高速に解くことに特化した固有値ソルバを開発し提供することの有効性について示した。

4.5.3.0.2　単プロセス・小問題で高速に動作する固有値ソルバの評価　上記の固有値計算ライブラリの評価を踏まえ、数値計算ライブラリ WG 内で富岳向けの小規模問題用固有値ソルバが検討され、候補となるライブラリコード Kevd の提供を受けた。このコードに対し FX100 では性能電力予測ツールを用いた見積もりを行なうための HPC-ACE2 命令化、富岳向けにはアルゴリズム変更と ARM-SVE 化を行ない、NICAM+LETKF で用いる問題サイズを用いた評価を行なった。その結果、性能電力予測ツールを用いた見積もりでは、1 プロセス利用での EigenExa の計算時間に対しておよそ 6 倍高速化することがわかった。

4.5.3.0.3　富岳試作機を用いた DGEMM の性能評価　各重点アプリケーションのうち DGEMM を用いるものについて、それぞれのターゲット問題サイズでの性能評価を行う中で、LETKF で用いる問題サイズでの性能が劣化していることがわかった。これは整合寸法が 1024 と 2 のべき乗になっているために、キャッシュライン競合が発生していることに起因していた。この問題を回避するために追加する配列要素のサイズを何パターンか試して評価を行ったところ、キャッシュライン 2 ライン分=64 要素を増やすことで演算効率を 4-6%改善できることを確認した。

4.5.3.0.4　使用メモリの削減　LETKF ではデータ同化に用いるモデルシミュレーションのアンサンブル数、あるいは観測データ数が増加するほど、計算時のメモリ使用量が増加する。使用量を出来るだけ少なく抑え、より大きいアンサンブルサイズかつより多くの観測データを利用可能にするために、プログラムの見直しを行った。まず、NICAM の単精度化に合わせて、NICAM のからの入力データの単精度化を行った。次にモデル計算値を観測値と同じ地点・同じ変数に変換したデータについて、各プロセスが担当する空間領域で必要となるデータを前もって見積もり、データ数のふるい落としを行うことで一時的に確保するメモリ領域を節約した。さらにアルゴリズムフローを見直し中間配列の削減を進めた。これにより、1000 アンサンブルかつ 1 プロセスあたり

50 万観測点の同化を試みるような場合においても、メモリ使用量が富岳 1 ノードの上限に収まるまで改善した。

4.5.3.0.5　LETKF 計算コアの単精度化　ソフトウェア調整 (2) では、LETKF の計算コアである、GEMM と固有値計算を格子点の数だけ繰り返す区間を単精度化し、高速化を図った。LETKF では全固有値と固有ベクトルを利用するため、単精度化に伴って小さい値の固有値が桁落ちすることを回避しなければいけない。そこで、LETKF の計算アルゴリズムを見直し、桁落ちを防ぐような計算順に変更した。この変更により、安定したデータ同化サイクルを継続することが可能になったことを 56km メッシュの長期実験で確認した。

4.5.4　NICAM+LETKF ワークフローにおけるコデザイン

4.5.4.0.1　最適化したワークフローの大規模実行での性能検証　NICAM と LETKF 間のデータ受け渡しの高速化のため、ファイル I/O と大域通信のスループットに着目して開発したモデルワークフローについて、京でより大規模な実験設定を用いて検証を行なった。1 万ノードまでのストロングスケールにおいて、新しい方法がスケーラビリティを大幅に改善することを実証した。この成果は論文の形でまとめられ、国際ジャーナルにおいて出版された [49]。

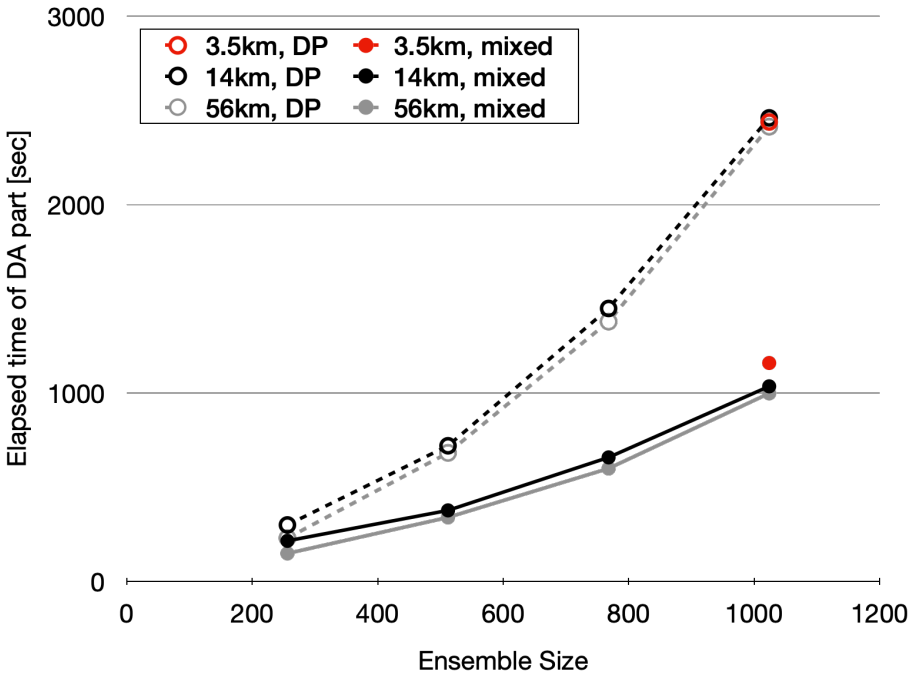
4.5.4.0.2　富岳実機を用いたデータ同化システムの性能評価　ソフトウェア調整 (2) では、富岳実機を用いた LETKF を含むデータ同化システム全体の性能評価を行った。図 4.24 は、アンサンブルメンバー数、水平解像度、浮動小数点精度を変えて実験を行った結果である。アンサンブルメンバー数を増加させた場合、ノード数は同一のためプロセスあたりの演算量が増加する。特にアンサンブルメンバー数 x アンサンブルメンバー数の大きさの配列に対して行われる固有値分解の計算量が増加する。これは図 4.25 に示す水平解像度 14km メッシュのケースにおいて示した経過時間の内訳においても明らかであり、倍精度を用いた場合、アンサンブルメンバー数の増加に対して非線形に固有値計算を行う区間の経過時間が増加する。一方で混合精度を用いた場合、固有値計算は単精度で実行され、アンサンブルメンバー数の増加に対してほぼ線形で経過時間が増加する。このように経過時間の増加が大きく抑えられた要因は、単精度計算を用いた際の deflation の効果であると考えられる。水平解像度を 56km, 14km, 3.5km と増加させた際にはプロセスあたりの格子点数が同じくなるようにノード数を増加させた。そのため同じアンサンブルメンバー数の場合はウィークスケーリングとなる。このときの経過時間の伸びは小さく、良好なスケーリング性能を示した。特筆すべきは IO 時間であり、図 4.25 においてもアンサンブルメンバー数の増加に対して IO 時間はほとんど増加しておらず、水平解像度を変えて利用ノード数を増加させてもほとんど影響が見られなかった。これは最適化した NICAM+LETKF ワークフローによってスケーラブルなファイル IO を実現した結果であり、また IO の読み書きを第一階層 FS である SSD に対して行い、スループットを大幅に改善することが出来たためである。

4.5.5　コデザインによるシステムへの実装要請項目

本コデザインを通してシステムへフィードバックした実装要請・改善項目を以下に示す。

1. L1 キャッシュビジー率の調査と改善

図 4.24: 富岳での LETKF の性能計測結果



NICAM の力学カーネルのうちキャッシュ律速となるよう最適化されたものについて、L1 キャッシュのビジー率が想定していた値よりも高く、早くに性能が頭打ちになっているという傾向が見られた。この原因の一つは上で述べた、想定しない間接 SIMD ロードの発行である。間接ロード命令が発行されると、リストアクセスが連続であっても容易に L1 ビジー状態になるため、出来る限り不必要な間接ロード命令を回避すべきである。もう一つの要因として、L1D キャッシュミス時の cache fill/eviction 処理に関わる L1 ビジーサイクル数が大きいという問題が挙げられる。これらのサイクル数を削減することを、ハードウェア要求事項とした。

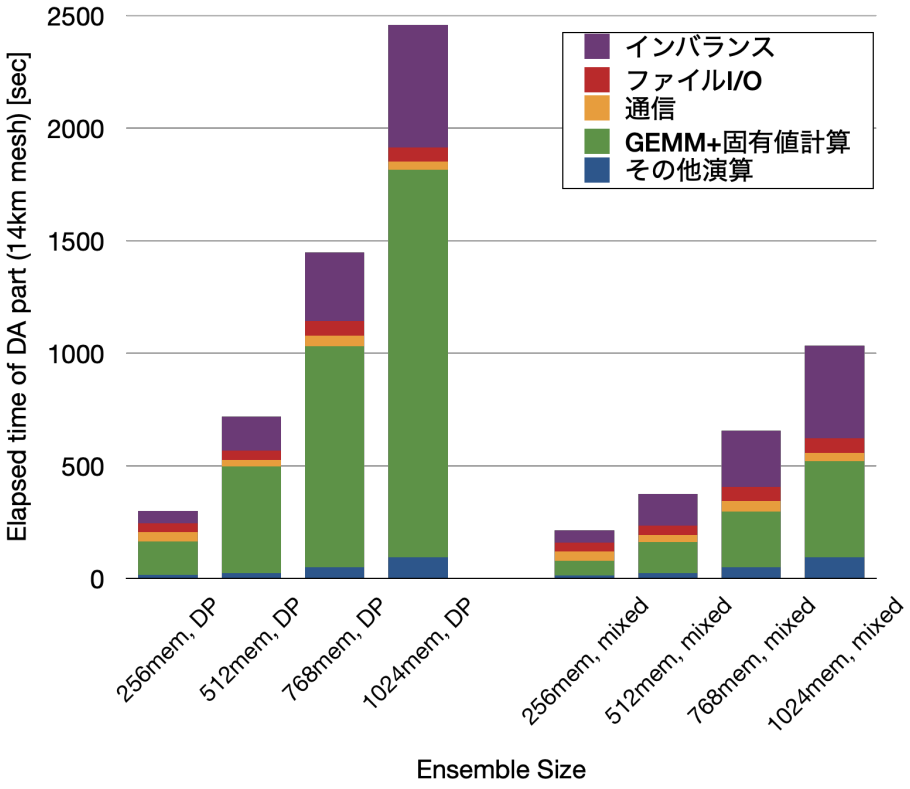
2. ハードウェアプリフェッチとソフトウェアプリフェッチの有効性に関する調査と改善

   ソフトウェアプリフェッチはハードウェアによるプリフェッチが動作しない場合に有効であるが、命令数を増やしてしまうため、なるべくハードウェアに任せたプリフェッチの発行を促したいと考える。ステンシル計算の場合、袖領域を飛ばして計算する場合にメモリアクセスの連続性が失われ、ハードウェアプリフェッチが動作しなくなる場合が非常に多い。これらについては、可能であればコード修正を行ない、出来なければソフトウェアプリフェッチを積極的に発行するようにする。ハードウェアプリフェッチが動作しなくなる条件としては、最適化の中でも特にソフトウェアパイプライニングが適用されることでキャッシュラインへのアクセスが連続性を失うケースがあることが明らかになった。この場合、ソフトウェアプリフェッチによる補完を必要とする。これについて、出来る限りディレクティブ行を入れずにコンパイラによる自動的な最適化処理が行われるような改善を、要求事項とした。

3. べき乗計算の混合精度利用に関する調査

   気象モデルのコードの中ではべき乗の計算が頻繁に出てくる（例えば、液体や固体の水の粒径分布近似）。そのため、べき乗計算の精度を落とすことで演算の高速化が出来ないか検討

図 4.25: 富岳での LETKF(14km メッシュ) の計測時間内訳



を行なった。評価では、底（Base）と指数（Exponent）について、単精度と倍精度の組み合わせによる演算時間の比較を行なった。京での評価ではどの組み合わせも実行時間にほとんど差は見られなかった。組み合わせによっては単精度を部分的に用いていても演算数がすべて倍精度の時と変わらないものもあった。

4. FX100 を用いたコード最適化を行なう際のコンパイラ環境整備

   FX100 と富岳ではスレッド数、L2 キャッシュサイズ、最大ストライドアクセス幅やレジスタ数が異なる。これらに依存したコード最適化を FX100 上で行ない、富岳性能電力予測ツールでの性能推定を行なった場合、見積もりを過大評価する可能性がある。これを避けて広範な範囲のソースコードの最適化作業を FX100 上で進めるために、富岳に見合うハードウェア資源の制限をかけてコンパイルを行なうコンパイラの提供を要請した。

5. キャッシュをバイパスしたストアの有効性に関する調査と実装の検討

   Intel の CPU には、キャッシュを通らずにメモリへ書き出しを行なう機能（streaming-store）が実装されており、この機能を操作するコンパイルオプションが存在する。一方で京や富岳では XFILL（相当）命令という、書き出し領域をメモリに問い合わせずに L2 キャッシュ上に生成し、メモリ-L2 キャッシュ間のストリームの削減を行なう命令が用意されている。両者の機能の違いと、streaming-store 機構を実装した場合の性能向上の見積もりと実装可能性について検討を依頼した。見積もりの結果、メモリ転送性能の向上率に関しては streaming-store 機構も XFILL 機構もほぼ同じ性能を発揮するということが確認された。さらにキャッシュに関しては、streaming-store 機構の方がキャッシュの動作を減らしビジー率を軽減するため、机上見積もりにおいて性能向上が見込めることがわかった。ただし、性能向上するのは L1

または L2 のビジー率がほぼ 100 %に達している場合であり、また streaming-store を使用できるループには制限がある。一方で streaming-store 機構をサポートするために追加される LSI の面積は CPU 全体の約 0.5%であり、コストを考えた場合実装は難しいという結論に至った。

6. OpenMP で collapse した時に SIMD・SWPL が抑制され性能劣化する場合について、性能劣化させない方法について検討

   連続したメモリ領域を触らない回転範囲になっている n 重ループについて、OpenMP で collapse しようとすると SIMD やソフトウェアパイプラインがキャンセルされ、性能がかなり劣化する。どのような条件で SIMD、SWPL がキャンセルされるのかについて調査を行ったところ、富士通コンパイラの場合は、ループ多重度よりも小さい次元数の配列にアクセスする場合、ループ回転範囲に昙間がある場合、配列の添字に式が指定されている場合、ディスタンスアクセスの場合、インダイレクトアクセスの場合を除くと、最内ループを collapse に含めても高速な最適化（高速な 1 重化変換）が適用されるため、性能が劣化しないことがわかった。しかし、前述の高速な 1 重化変換が適用されない条件のどれかに該当するときは、最内ループを collapse に含めないことが性能に対して最も有効であることが示された。そこで、最内ループまで含めた collapse が指定され、かつ、コンパイラが高速な 1 重化変換を行わないと判断した場合に、警告メッセージをを出力するよう要請した。また、自動的に最内ループを collapse に含めないよう指定するコンパイルオプションを準備することを要請した。

   加えて、predicate の機能を用いてマスクによりメモリアクセスを連続化し、最適化を出来るようにする方策について検討を行ったが、マスク配列の準備にかかるオーバーヘッドと、マスク処理に関わる命令数の増加によって、最内ループを collapse から外す方策より遅くなることがわかった。

7. 富岳における詳細プロファイラのオーバヘッドについての確認

   詳細プロファイラは入れ子で使うことも多いが、あまり頻繁にコールするとプロファイラにかかる時間が見えてしまい、全体の計測時間を劣化させることがわかっている。富岳での詳細プロファイラ挿入時のオーバーヘッドがどの程度であるか、京や FX100 と比較した確認を要請した。詳細プロファイラの実行に関しては通常モードと高速モードがあり、現時点ではどちらも京・FX100 に比べてオーバーヘッドが大きいことがわかった。

8. 演算結果に影響を与える最適化を排除したコンパイルオプションセットの提供

   物理カーネルを倍精度と単精度で計算した時に、最終的な結果が丸め誤差よりもかなり大きくなることがあった。この結果がアルゴリズムに起因するものであるか、強い最適化オプションによるものであるかを切り分けるために、推奨最適化コンパイルオプションセット (-Kfast) で誘導される最適化オプションから演算結果に影響を与えるものを排除し、さらにスレッド並列処理で発生する計算誤差あるいは計算順序変化を回避するオプションを追加して実験を行った。このようなデバッグ方法は他のアプリケーションにとっても有用であると考えられるため、用いたコンパイルオプションをまとめたセットを新たなオプションとして提供してもらうよう要請した。最適化レベルを最低にしスレッド並列を有効にしない方法では、カーネルの計算に時間がかかりすぎる。このセットは計算結果に影響を与えない範囲で最大限高速化を試みようとする点が重要である。

表 4.31: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (NICAM+LETKF)

実装要請課題	対応
間接ロード命令の抑制による L1 キャッシュビジー率改善	キャッシュ最適化された力学カーネルの L1 キャッシュビジー率が過剰に高く性能を律速している。コンパイラにより過剰に発行されている間接ロード命令の回避抑制を要請。
L1 キャッシュミス時のビジーサイクル数削減	L1 キャッシュミス時のキャッシュライン置き換え処理の fill/eviction に関するビジーサイクル数削減を要請。fill 動作分の削減は難しいが、clean eviction (キャッシュライン上のデータが更新されていない場合の eviction 動作) についてビジーサイクル数を削減する
HW/SW プリフェッチの協調最適化	HW プリフェッチ・SW プリフェッチ・パイプライニングをバランス良く用いてプリフェッチ機構を有効利用するコンパイラ機能の実装
単精度べき乗計算の高速化	単精度関数が倍精度関数に比して十分高速である実装を要請。京ではほとんど時間短縮されなかった。
キャッシュをバイパスするストア処理	Intel CPU の streaming-store 機構と XFILL 命令の比較精査と streaming-store 機構相当の実装検討。メモリストア性能では XFILL で streaming-store と同水準の性能を達成。L1 キャッシュビジー率緩和には有効であるが、効果が得られるケースが限られることと HW 面積増加のバランスから streaming-store 機構は見送り、XFILL のみを実装する。
OpenMP collapse 指定時に最内ループを含む/含まないの最適な実装選択と適切なメッセージの出力	OpenMP 指示行で最内ループまで含めた collapse が指定された場合、最内ループを含む 1 重化変換および最内ループを含まない 1 重化変換の 2 者命令列のうち、より高速な命令列をコンパイラが自動的に選択する機能と、適切にメッセージを出力する機能を要請
ハードウェアプリフェッチのロスト解消	FX100 においてハードウェアプリフェッチで十分なメモリバンド幅を実現できない問題を精査した結果、ハードウェアプリフェッチのロストによる影響と判明。富岳ではキャッシュラインをまたいだ SIMD load/store 命令実行時に命令再発行処理に伴うロストが発生しない設計とする
FX100 上で富岳相当のハードウェア資源制限を設けて性能評価を行うためのコンパイラ環境整備	FX100 上で富岳向けに広範な範囲のソースコードの最適化作業を進めるために、富岳相当のハードウェア資源制限を設けてコンパイルを行うコンパイラの提供を要請した
キャッシュインデックス競合時の L1 キャッシュスループット低下の解消	L1 ミス発生時に、L1 ミスキューの競合チェックの単位をキャッシュインデックスからフルアドレスに変更することで、偽の競合検出を排除する
MPI_Scatter の高速化	MPI_Scatter の性能が京の実装では非常に低く、大幅な改善が必要。MPI_Bcast と同様な実装を行うことにより性能改善が可能なことを検証済み
計算結果に影響を与えない最適化コンパイルオプションセットの提供	ソースプログラム変更後に現れる計算結果の違いが、アルゴリズム起因か、強い最適化オプション起因かを切り分けるために推奨最適化オプションセット (-Kfast) で誘導される最適化オプションから、精度打ち切りやスレッド足し合わせ順の非保証など、計算結果に与える影響が大きいオプション群を除いたものを -Kfp_precision オプションとして提供する。

4.5.6　富岳向け性能推定

4.5.6.0.1　ターゲット問題　全球 3.5km 水平メッシュ、鉛直 94 層での計算（NICAM）を初期値の異なる 1024 アンサンブルメンバーで 4.5 時間分シミュレーションし、その計算結果と数 10 万点規模の観測データを用いてデータ同化を行う。これを 1 サイクルとし、2ヶ月間分の同化シミュレーション（480 サイクル）を実行するのに必要な時間を性能評価の基準とした。

4.5.6.0.2　京でのベースライン測定　京における NICAM1 メンバーの実行時間には、プロセスあたりの問題サイズをターゲット問題と揃えた 10 ノード（10 プロセス）での実測値から、パーフェクトなウィークスケーリングを仮定して推定した。また LETKF の演算部分についても、プロセスあたりの問題サイズを揃えた演算カーネルコードの実測に加えて、通信部分については机上計算と 81920 ノードを用いた通信カーネルコードの実測値を元に推定を行った。また、ファイル入出力部分についても京の 2560 ノードを用いた実測を元に推定を行った。これらの結果を組み合わせることで、1 サイクルあたりに要する日数を 15 日、480 サイクルには 20 年を要すると見積もった。

4.5.6.0.3　富岳での測定と評価方法　富岳実機での測定に基づくターゲット問題の経過時間推定は以下の手順により実施した。

1. NICAM・LETKF 両アプリの実行時間を個別に推定または測定し性能倍率を算出。NICAM は同時実行メンバー数を 256 とし、メンバーあたり 2048 プロセス 512 ノード、計 131072 ノードを利用する。どちらのアプリケーションもデータ入出力時間を含み、第 1 階層 FS を利用する。
2. NICAM1 セットの時間方向の反復回数を削減したケースを実測し、LETKF は実測により測定。1 サイクルの時間から 480 サイクルに必要な経過時間を推定。
3. 実行モードは boost,eco なしとし、ノードマッピングに関してはいずれの計測においても指定しない。

4.5.7　富岳で達成された性能倍率

最終的な京に対する性能倍率及び消費電力を示す。測定環境の概要は以下の通りである。

1. 測定日時　2020/8/1
2. コンパイラ・MPI のバージョン：lang/tcsds-1.2.26
3. 利用ライブラリ：-SSL2BLAMP –SCALAPACK に加え、Kevd(2020/7/10 ビルド版)
4. コンパイル･リンクオプション（Fortran）:-Kfast, parallel, ocl, preex, array_private, noalias=s -Nlst=i -Nlst=t -X03 -Ncompdisp -Koptmsg=1 -Cpp -Kdynamic_iteration -Ksimd -Kauto, threadsafe -Kloop_fission_threshold=40
5. コンパイルオプション（C）：-Kfast, parallel, ocl, preex, array_private, region_extension, restp=all -Ksimd

最終的にノーマル・エコなしで 127 倍の性能向上を得た。2019 年までの性能見積もりと実測値の間で、NICAM は 50 倍から 42 倍へ倍率が低下した。これは同時実行メンバー間のワークロードの違い、各メンバーのノード形状の違いに起因する通信時間の違い等が存在し、各メンバーの経過時間のうち最大値を用いると、これまで見積もりに用いていた平均値より経過時間が 20 %程度増加したことによる。一方、LETKF は 419 倍から 802 倍へと向上し、計算時間がほぼ半減した。これは 2020 年度に行った観測データのファイル I/O の削減・分散と、行列積と固有値計算の単精度化が寄与している。全体の経過時間としては、両アプリの経過時間の変化が相殺し、2019 年までの推定値である 133 倍とほぼ同等の性能結果となった。

表 4.32: NICAM+LETKF の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
NICAM+LETKF	127 倍	22 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

表 4.33: NICAM+LETKF の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
NICAM+LETKF	127 倍		116 倍		121 倍		115 倍	
	24 MW	25 MW	22 MW	23 MW	19 MW	20 MW	18 MW	20 MW

4.5.8　主筆者・執筆協力者

4.5.8.0.1　主筆

- 八代　尚（国立環境研究所）

4.5.8.0.2　執筆協力者 （５０音順）

- 河合 佑太（理化学研究所計算科学研究センター）
- 小玉 知央（海洋研究開発機構）
- 寺崎 康児（理化学研究所計算科学研究センター）
- 富田 浩文（理化学研究所計算科学研究センター）
- 中野 満寿男（海洋研究開発機構）
- 三上 和徳（理化学研究所計算科学研究センター）
- 南 一生（理化学研究所計算科学研究センター）
- 三好 建正（理化学研究所計算科学研究センター）

4.6 重点課題 5 NTChem のコデザイン

4.6.1 アプリケーションの特徴とコデザインの方針

NTChem プログラム [41] は、京などのペタスケールスーパーコンピュータを活用し大規模分子の第一原理的電子状態計算を実現することを目的としたアプリケーションである。NTChem に実装されている代表的な計算手法として、(1) ハートリーフォック (HF) 法や密度汎関数理論 (DFT) に基づき、分子の基底状態と励起状態の電子状態を計算する機能、(2) より高精度な電子相関理論である Møller-Plesset2 次摂動 (MP2) 法、結合クラスター (CC) 法、量子モンテカルロ (QMC) 法に基づく基底状態と励起状態の電子状態計算機能、などが挙げられる。

これらの計算方法のうち、HF 法、DFT 法、MP2 法のコードについては京コンピュータでの超並列計算のために MPI/OpenMP ハイブリッド並列化が施されており、現在では京数千ノードから数万ノード規模を用いた超並列計算が可能である。富岳重点課題 5 では、太陽電池シミュレータや凝縮相内化学反応シミュレータとして NTChem を利用すべく、開発および高度化が進められている。上記で実行されるシミュレーションでは、定量性を満たす高精度電子相関理論に基づく計算が求められるが、原子軌道の数を N とすると、これらの計算における計算コストが $O(N^5)$ 以上となるため計算のボトルネックとなる。そこで、重点課題で目標とするアプリ成果を達成するためには、高精度電子相関理論の高度化とコデザインが求められている。

NTChem には MP2 法、CC 法、QMC 法など高精度な電子相関理論が実装されているが、そのうち最も計算コストが低く大規模分子の計算が可能かつ京を対象とした最適化が最も進められている Resolution-of-Identity(RI) MP2 法に基づく電子相関エネルギー計算を対象として富岳におけるコデザインを進めた。

NTChem/RI-MP2 モジュールは行列要素の計算や行列-行列積計算などの密行列演算が主要な演算処理であり、要求 B/F 値が低く演算量の多いアプリケーションとしてコデザインに寄与する。さらに、行列要素データのノード間通信に MPI 集団通信や 1 対 1 通信を多用し通信データ量も多いため、高い通信バンド幅が要求されるアプリケーションとしてコデザインに寄与する。また、行列-行列積計算などの線形代数演算は、BLAS、LAPACK などの数値計算ライブラリの基本機能を活用した実装となっているため、数値計算ライブラリのコデザインにも寄与する。

NTChem の RI-MP2 エネルギー計算モジュールを富岳で実行することを想定した、計算科学的観点における性能改善のためのコデザイン内容を次節以下に報告する。

4.6.2 RI-MP2 計算の超並列実装

コデザイン開始前の NTChem/RI-MP2 モジュールにおける MPI/OpenMP ハイブリッド並列処理においては、計算のボトルネックとなる 4 中心電子反発積分計算の並列化を行う際に、仮想分子軌道のインデックスをプロセス分割する実装が完了していた [37]。これは占有分子軌道のインデックスをプロセス分割する従来のアルゴリズムよりも並列度が高く、京において数千ノード規模での効率的な並列計算の実行を可能としていた。しかしながら、計算に用いることが可能なプロセス数が仮想分子軌道の次元数に制約されるため、数万プロセスを用いた超並列計算は不可能であることが問題となっていた。

基本設計では、まずこの問題を解決し京および富岳にて数万 MPI プロセスを用いた超並列計算を実行可能とするため、局所的な MPI communicator を活用し 2 次元的な MPI 階層並列化を導入したアルゴリズムの開発を実施した [38]。この改善では、最外ループとなる仮想軌道インデック

スの第 1 軸の MPI 並列化に加え、このループ内で実行される積分計算と行列演算に関わる行列をブロック分割し第 2 軸の並列化を行った (図 4.26、4.27、4.28)。

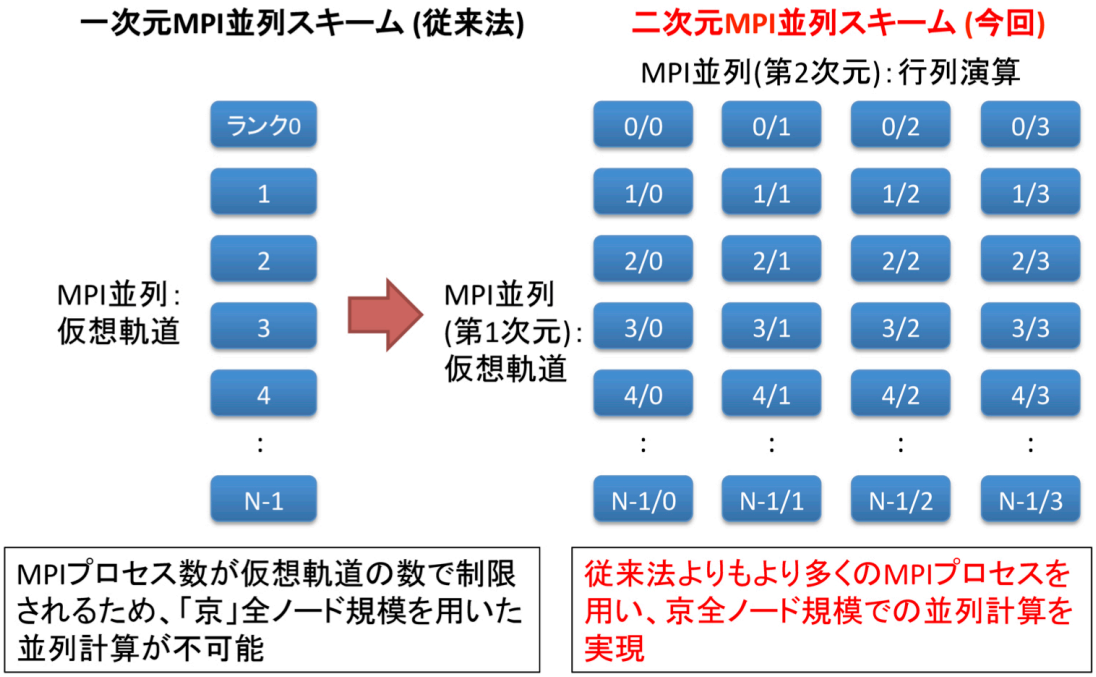


図 4.26: RI-MP2 計算の 1 次元並列化 (従来法) と 2 次元 MPI 階層並列化 (新手法)

この 2 次元 MPI 階層並列化では、ノード間通信パターンを変更し 3 中心積分の 1 対 1 通信効率の向上作業も実施した。従来の実装では 4 中心積分計算に必要な 3 中心積分データについて、データを保持するプロセスからデータを必要とする複数のプロセスへそれぞれ通信する P2P 通信法を採用していた (図 4.29 左)。この方法では、ネットワーク・トポロジーの存在するシステムでは通信相手の数が増大した場合に、ホップ数の違うプロセスとの通信で性能が低下することが、京でのベースラインプロファイルの結果より明らかとなった。通信されるデータの動きの詳細な解析を行った結果、データを隣接ノードに順次 1 対 1 通信で受け渡していくリング通信法 (図 4.29 右) を採用することで、局所的なネットワーク・トポロジーを有効に活用した通信が可能であることが明らかとなったため、リング通信法の実装を新たに行った。

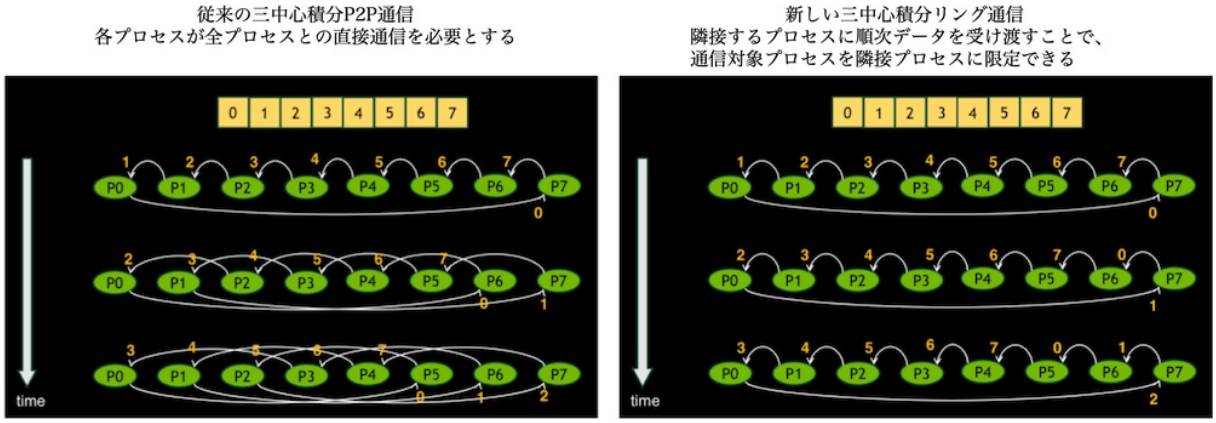


図 4.29: 3 中心積分データ通信の P2P 通信法からリング通信法への変更

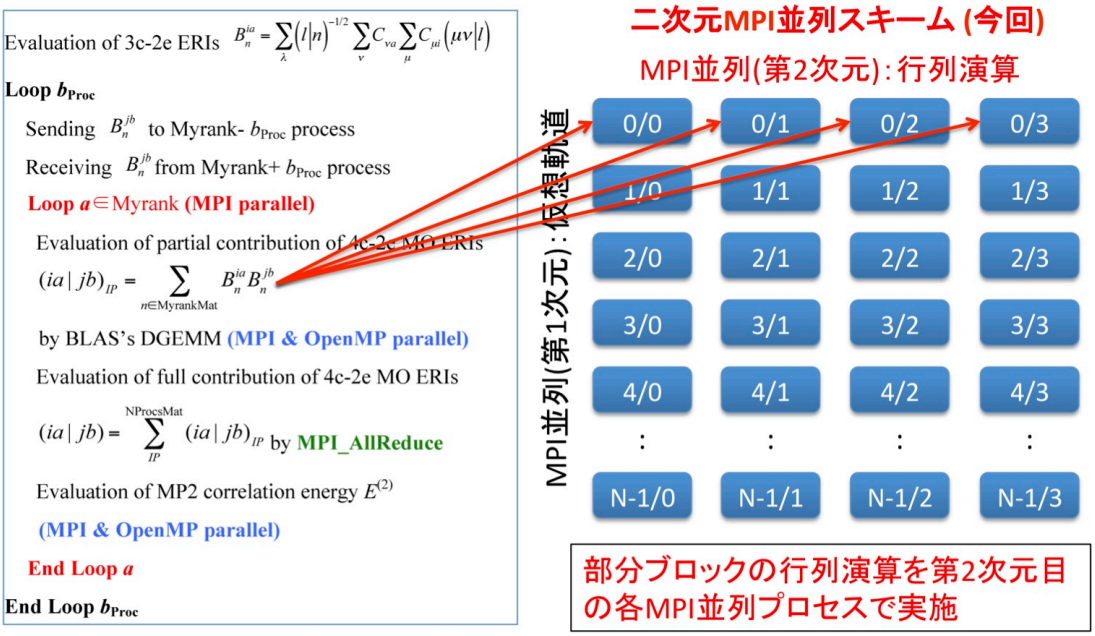


図 4.27: RI-MP2 計算の 2 次元 MPI 階層並列化時の行列データ並列化

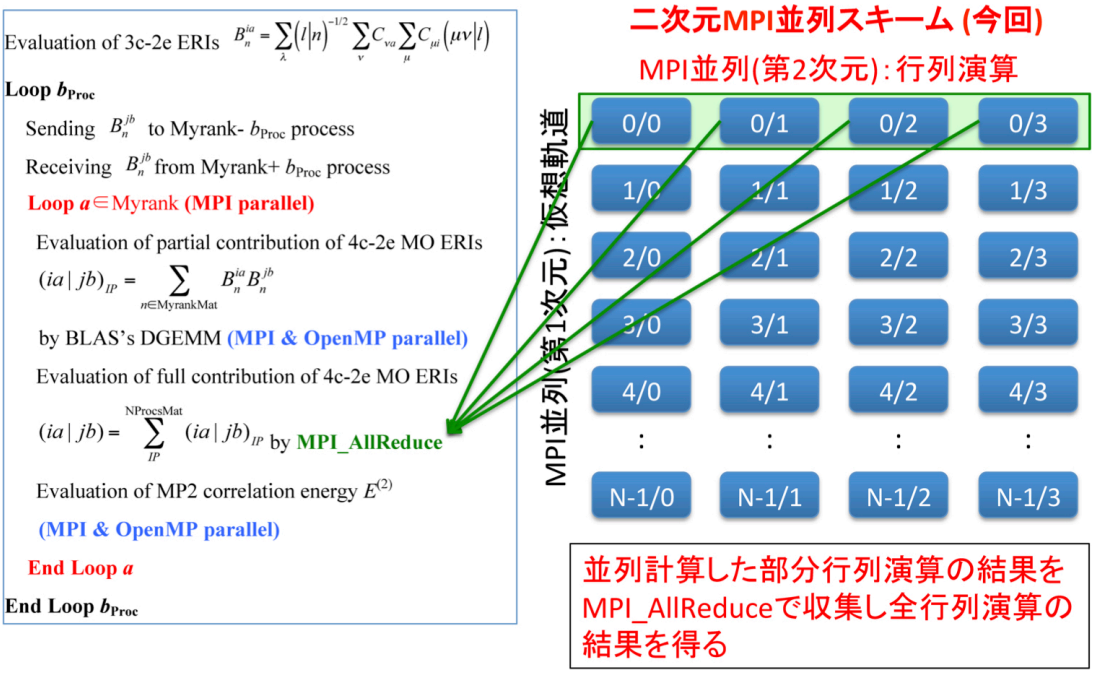


図 4.28: RI-MP2 計算の 2 次元 MPI 階層並列化時の行列ブロックデータの通信パターン

はじめに、P2P 通信法からリング通信法への変更による性能向上効果を調べるために、ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ の RI-MP2/cc-pVTZ 計算 (240 原子、6432 原子軌道、600 占有軌道、5832 仮想軌道、16992 補助基底関数) を京 2048 ノードを用いて行った。このテスト計算では 2 次元階層並列化による性能の向上分と切り分けるために、従来法である 1 次元並列化条件でのテスト計算を行った (図 4.30)。その結果、リング通信法の採用により従来の P2P 法の場合よりも通信時間の大幅な削減が達成されたため、本通信法が有効であることが明らかとなった。

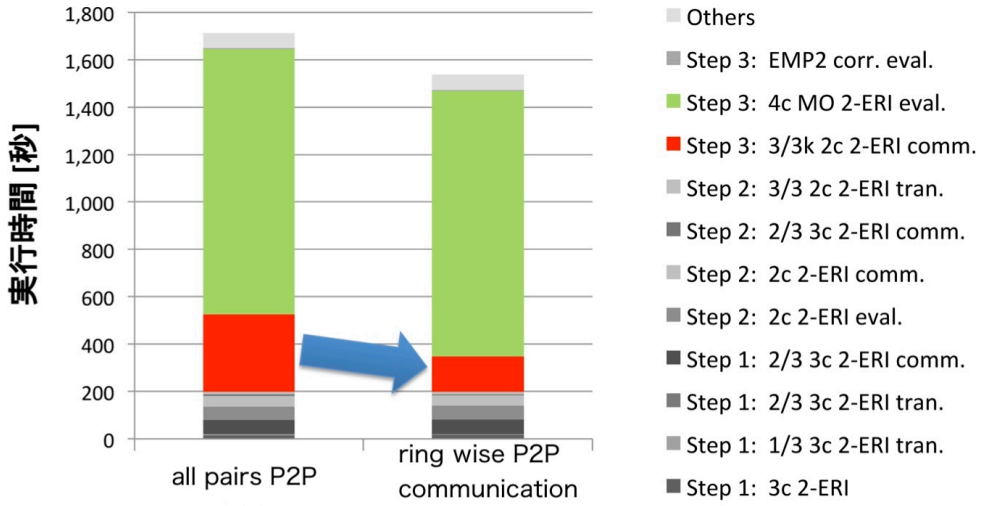


図 4.30: 3 中心積分データ通信の P2P 通信法からリング通信法への変化よる通信時間削減効果

次に、2 次元 MPI 並列化の効果を調べるために、ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ の RI-MP2/cc-pVTZ 計算 (240 原子、6432 原子軌道、600 占有軌道、5832 仮想軌道、16992 補助基底関数) を第 1 軸と第 2 軸の MPI プロセス数を変化させて実行した。実行結果を表 4.34 に示す。第 2 軸の MPI プロセス数を 1 から 16 に増やすに連れてストロングスケーリング性能が改善され、特に 16 プロセスを用いた場合には最大 65536 ノードまで良好なストロングスケーリングが達成されている。従来のアルゴリズムは第 2 軸に 1 プロセスを用いた場合に相当するが、この場合と比べて大幅な並列性能向上効果が確認できたため、本アルゴリズムの改良は数万プロセス規模での超並列計算に有効であると考えている。

表 4.34: ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ の RI-MP2/cc-pVTZ 計算 (240 原子、6432 原子軌道) の「京」での計算時間、並列性能

number of nodes	Nprocsmat=1 time [seconds]	Nprocsmat=2 time [seconds]	Nprocsmat=4 time [seconds]	Nprocsmat=8 time [seconds]	Nprocsmat=16 time [seconds]
1024	2366	2289	2211	2346	-
2048	1491	1292	1291	1315	1581
4096	1451	856	743	773	895
8192		826	532	490	541
16384			543	366	391
32768				359	298
65536					282

また、2048 ノードを用いた場合に第 2 軸の MPI プロセスを 1 から 16 に増やした際の演算時間と通信時間の内訳を図 4.31 に示す。プロセス数が 1 から 8 へと変化するにつれて計算のホットス

ポットとなる 4 中心積分計算時間 (緑色) が減少し、行列演算の並列化により計算が加速されている。また、3 中心積分のノード間通信時間 (赤色) も、プロセス間で 1 対 1 通信する 3 中心積分データのプロセス数とデータ量が減少するに従い、減少している。一方で、部分プロセスグループ内での行列ブロック集団通信量が第 2 軸のプロセス数の増加に従い増加するため、その通信時間 (青色) も増加している。これらのプロセス数の増加に伴う演算時間と通信時間の減少と増加のバランスの兼ね合いで実行時間が変化し、この場合では第 2 軸に 4 プロセスを用いた場合の実行時間が最短となる。この結果は、アプリ実行の際に高い実行性能を達成するためには、第 1 軸と第 2 軸に割り振る MPI プロセス数を最適に選択する必要があることを示唆している。

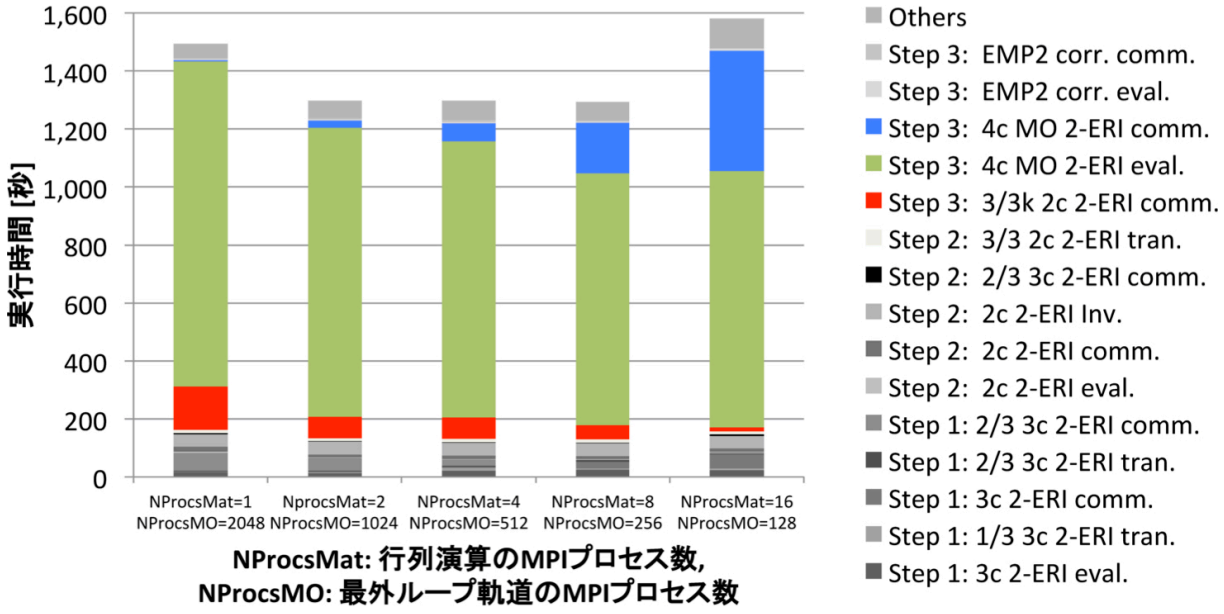


図 4.31: 2048 ノード 2 次元 MPI 階層並列化時の演算時間と通信時間の内訳

次に、360 原子、9840 原子軌道系のナノグラフェン 2 量体 $(C_{150}H_{30})_2$ の RI-MP2/cc-pVTZ 計算を最大で京 8911-80199 ノードを用いて実施した結果を表 4.35 に示す。従来のアルゴリズムでは、仮想軌道数 8911 が並列プロセス数の限界であったが、2 次元 MPI 並列化により利用可能なプロセス数を 80199 まで増加させることが可能となり、かつ並列実行による加速も達成されている。

表 4.35: ナノグラフェン 2 量体 $(C_{150}H_{30})_2$ の RI-MP2/cc-pVTZ 計算 (360 原子、9840 原子軌道) の「京」での計算時間、並列性能ならびに実行性能

ノード数	CPU コア数	計算時間 [秒]	高速化率	PFLOPS	実行効率 [%]
8911	71288	2692	71288	0.7	62
17822	142576	1634	117414	1.2	54
35644	285152	1095	175238	2.0	44
62377	499016	817	234959	2.8	36
80199	641592	759	252893	3.1	30

4.6.3 RI-MP2 計算のメモリ分割実装 (1)

4.6.2 節で示した超並列実装を行ったコードでは、一部の行列演算を行う際に行列データをレプリカとしてメモリに保持して処理を行うアルゴリズムを採用している。しかしながら、本コードを用いて富岳におけるターゲット問題の計算を行うのに必要なメモリ量の見積もりを行った所、1 プロセスあたりで 31.6GB のメモリが必要となる結果となり、本コードをそのまま用いた計算が不可能であることが明らかとなった。

基本設計時では、1CMG あたりの使用メモリ量を低減するために、行列データの分散化を進めて、各プロセスがレプリカを排した計算が可能となるようなメモリ分割アルゴリズムの検討を行った。検討したメモリ分散アルゴリズムを想定して、後述する富岳ターゲット問題規模の計算を行う際に必要なメモリ量の概算見積りを行ったところ、富岳想定アーキテクチャおよび京での計算が可能であるとの見通しを得た。

詳細設計 (1) から詳細設計 (3) までの期間に、基本設計時に検討したメモリ分割アルゴリズムをフルアプリとして実装した。具体的な実装の内容は以下である。

(1) 2 中心 2 電子積分行列計算のメモリ分割実装

(2) 2 中心 2 電子積分行列計算のコレスキー分解および分解して得られた上三角行列の逆行列の計算のメモリ分割実装

(3) 3 中心積分計算のメモリ分割実装

これらの実装では 2 電子積分計算を基底関数の全角運動量に依存したシェルブロック単位で行うため、行列の不均一な列ブロック分割に基づいた実装を行った。

(1) の 2 中心 2 電子積分行列計算のメモリ分割実装については、列方向のシェルをブロック分割し各ブロック行列要素を MPI 並列化して計算する実装とした。シェルと基底関数の対応関係の都合上、ブロック分割した行列の次元に相当する基底関数の次元は不均一となっている。

(2) の行列のコレスキー分解および上三角行列の逆行列計算のメモリ分割実装では、(1) で得られたブロック行列毎に MPI 並列化し演算を行う実装とし、演算の実装にはコレスキー分解ルーチン DPOTRF と逆行列計算ルーチン DTRTRI、BLAS のレベル 3 の行列-行列積 DGEMM を用いた。計算に必要なブロック行列データの通信については。サブコミュニケータ MPI_COMM_MAT 内で MPI_BCAST 通信を行い、行方向にデータを送信する実装とした (図 4.32、図 4.33)。

(3) の 3 中心積分計算のメモリ分割実装では、3 中心積分計算時に列方向のシェルをブロック分割し各ブロックの計算を MPI 並列化する実装とした。(1) の 2 中心積分計算と同様にブロック分割した列の次元は不均一となっている。3 中心積分計算後に係数行列を作用させる際の行列-行列積演算はそれぞれのブロックごとに BLAS の DGEMM を用いて行い、得られた行列データの部分和を必要に応じて MPI_COMM_MAT 内で MPI_ALLREDUCE で収集する実装とした。

本メモリ分割実装版の確認テストとして、ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ のの RI-MP2/cc-pVTZ 計算を京 1536 ノードを用いて実施した結果を表 4.36 に示す。メモリ分割版は行列を MPI_COMM_MAT 内でブロック分割したプロセス数を増やすにつれて高速化することが確認された。この高速化の要因として、行列のコレスキー分解と逆行列計算を MPI 並列化したことと、行列のメモリ分割によってレプリカで保持していた行列要素の総和を取るための MPI_ALLREDUCE 通信が削減されたことが挙げられる。

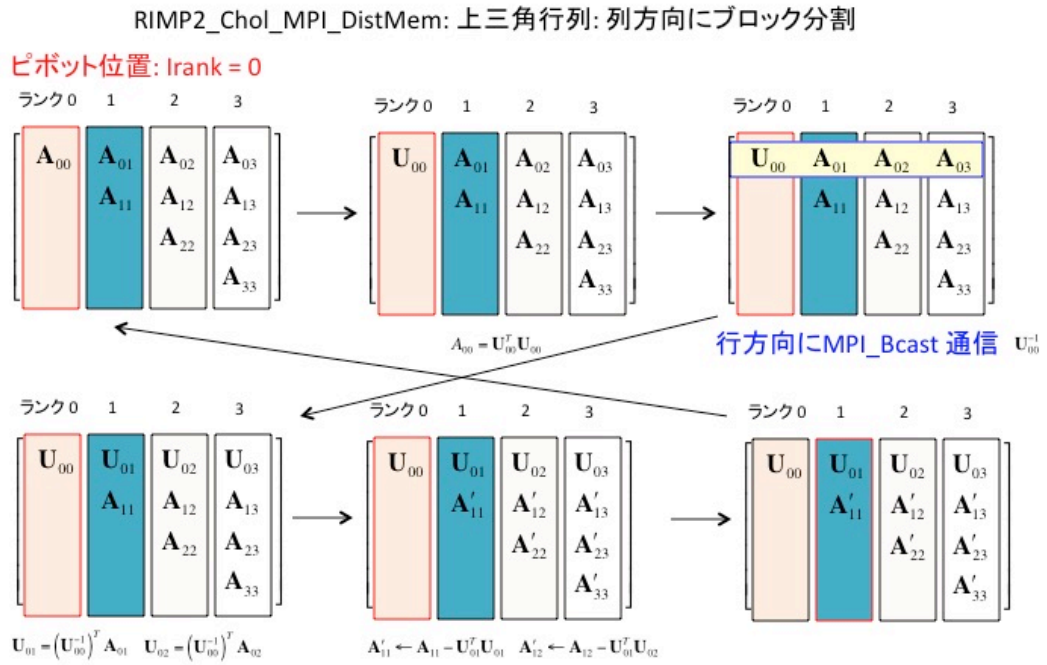


図 4.32: 行列のコレスキー分解のメモリ分割並列実装

4.6.4 RI-MP2 計算のメモリ分割実装 (2)

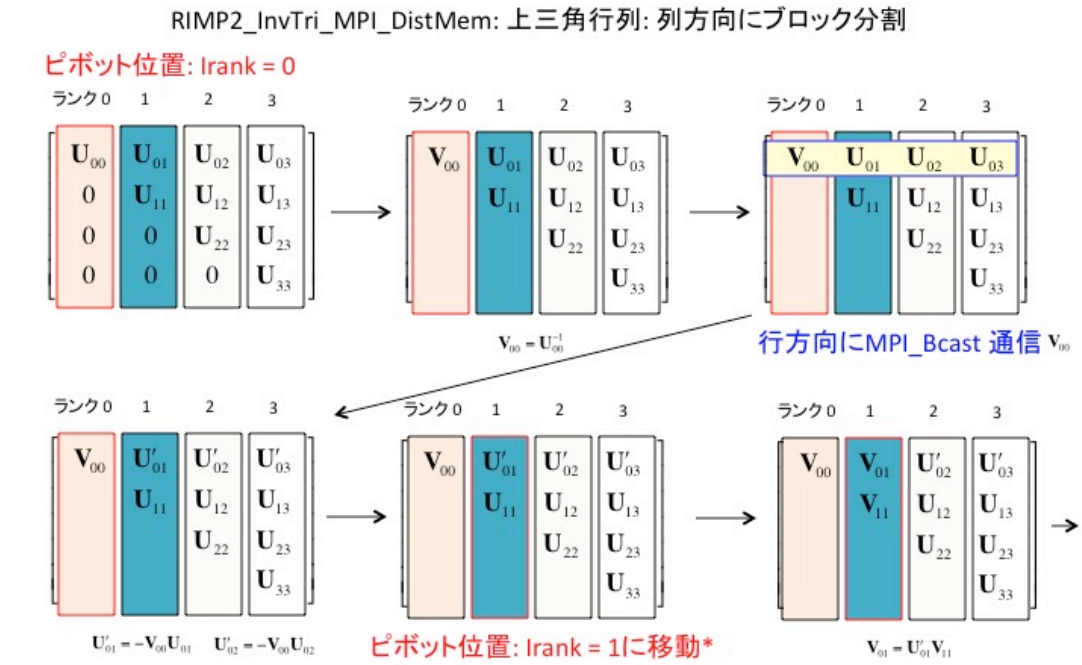
詳細設計 (4) 期間の早期にターゲット問題を実行するために必要な入力軌道データ作成が完了し、京上で実際にターゲット問題を実行して必要なメモリ量の確認を行なった。確認の結果、京上で 82928 ノードを用いて実行したジョブの統計情報ファイルに出力されるノードあたりの MAX MEMORY SIZE (USE) 値は 11.1GB となり、富岳の上でターゲット問題を実行するためには詳細設計 (3) の実装方法では不十分であることが判明した。このため詳細設計 (4) の期間においてメモリ使用量のさらなる削減の検討と実装を進めた。具体的には本節前段の配列分割に加えて以下を実施した。

(4) 行列を用いる一連の演算について行成分を一度に取り扱わず、分割して複数回に分けて行い、一時的配列についても任意にサイズを確保可能とする。

(5) 4.6.5、4.6.7、4.6.8 に説明する混合精度化アルゴリズムを導入し、倍精度データにかえて単精度データをメモリに格納することで、4 中心積分計算区間で確保する配列のメモリサイズを半減化

(6) 従来全プロセスが保持していた分子軌道係数の全データ 3.1GB の格納方法を変更し、RI-MP2 計算中は 0.3GB の占有軌道に関する係数は全プロセスが保持し、仮想軌道の係数は各プロセスが担当する部分 (データ量は (2.8/NProcsMat)GB) のみを保持する。

上記のメモリサイズ削減案を全て追加実装し、その効果を確認するために、京の上でターゲット問題の計算を実行した。当初案のプロセス数 82928 と二次元分割パラメタ 16 を用いた場合、プ



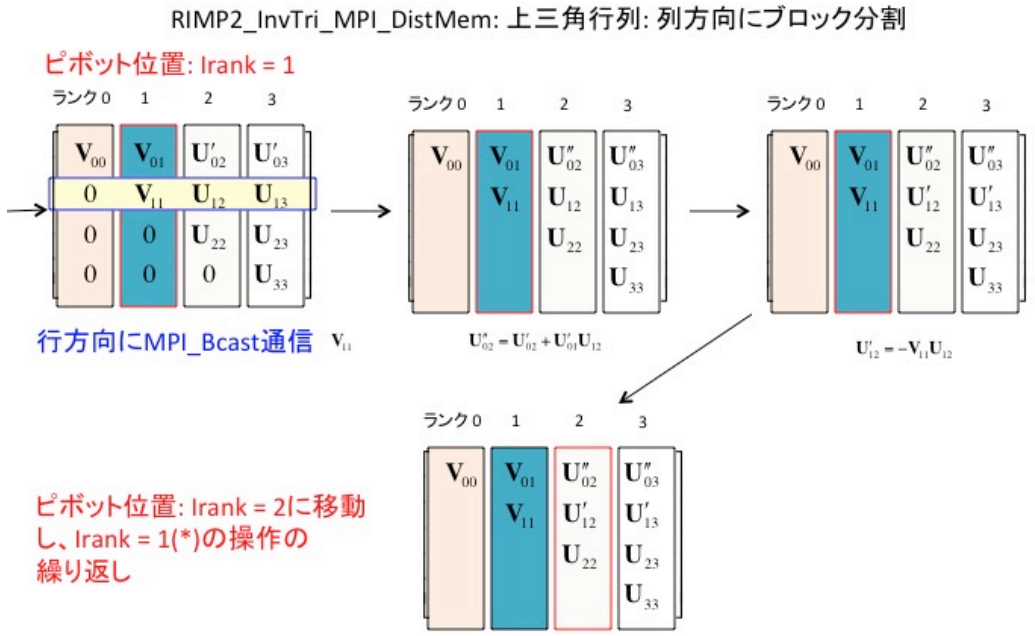


図 4.33: 上三角行列逆行列計算のメモリ分割並列実装

表 4.36: メモリ分割版と非分割版の「京」での実行性能：ナノグラフェン 2 量体 $(C_{96}H_{24})_2$ の RI-MP2/cc-pVTZ 計算 (240 原子、6432 原子軌道)、「京」1536 ノード使用

	メモリ分割版			非分割版		
行列分割プロセス数	4	12	24	4	12	24
3 中心積分計算実行時間 1	94.15	93.21	98.77	120.35	185.15	296.89
2 中心積分行列計算実行時間	60.53	34.15	34.68	62.72	63.32	70.95
3 中心積分計算実行時間 2	64.29	44.10	49.89	70.18	73.45	90.05
4 中心積分計算実行時間	1398.14	1492.51	2020.02	1417.97	1603.80	1985.63
全実行時間	1556.58	1629.82	2168.68	1608.50	1862.40	2372.57

ロセスあたりの MAX MEMORY SIZE (USE) は 7.5GB であった。富岳において 1CMG に 1 プロセス、従って 1 ノードあたり 4 プロセスを割り当てたプロセスレイアウトを想定すると、アプリケーションが必要とするメモリサイズは 30GB となり、富岳に実際に搭載されるノードあたりのメモリ容量をわずかに下回る結果となった。また、後出の節 4.6.11 で示す別のレイアウト案であるプロセス数 71280 と二次元分割パラメタ 8 を用いた場合、プロセスあたりの MAX MEMORY SIZE (USE) は 6.5GB となった。上述したメモリ分割実装 (1) および (2) の結果、富岳上でのターゲット問題の計算実行が可能なサイズとなったと評価できる。

尚、当初案に加えてプロセス数 71280 と二次元分割パラメタ 8 を用いたジョブが確認された背景は、メモリサイズ制約下で最も効率の良い計算性能を実現するプロセスレイアウトを検討する余地が生じたためである。最終的に富岳上でターゲット問題の計算を実行するジョブのプロセスレイアウトの決定経緯については、他のジョブ構成案とともに、後出の節 4.6.11 で詳しく説明される。

4.6.5 RI-MP2 計算の単精度・倍精度の混合精度化アルゴリズムの検討

RI-MP2 計算にいて最も演算量が多い区間は 4 中心積分計算区間である。後述する京におけるベースライン性能の推定においては、全演算時間の内ほとんどをこの区間の演算時間が占めると評価された。詳細設計開始時点において、この演算は倍精度密行列の行列-行列積演算として取り扱っているため BLAS Level3 の DGEMM を用いて処理を行う実装となっている。一方で、コデザインで対象としている電子相関計算法の MP2 法やより高精度な結合クラスター理論において実際に計算で得られる電子相関エネルギーは、HF エネルギーや DFT エネルギーよりも 4-5 桁程度小さい値であり、実アプリで許容される有効数字として小数点以下 3 桁 (原子単位で mHartree のオーダー) 程度となる。

一方、京では単精度と倍精度とで浮動小数点演算のコスト（演算スループット）が同じであるが、富岳においては単精度 SIMD 演算を倍精度 SIMD 演算の 2 倍の効率で実行できるマイクロアーキテクチャが採用され、計算性能・メモリ要求量の両観点から単精度計算の利点について認識が広まった。そのような背景のもと NTChem/RI-MP2 においても、アプリケーションのレベルで必要な計算精度を確保でき、かつ高効率な単精度 SIMD 演算を活用するアルゴリズムの検討が行われることになった。単純に計算全体を単精度で実行すると RI-MP2 計算に求められる精度を確保できないことは明らかであるため、電子相関エネルギー計算の実際の利用に耐えうる精度をアプリレベルで確保した上で計算を高速化することが可能な混合精度化アルゴリズムの検討を行った。

詳細設計 (1) ではまず計算に必要な行列あるいはテンソル要素のうち、絶対値の小さなものについては単精度で保持して計算を進めるアプローチで検討を実施した。具体的なアルゴリズムを以下に図（4.34）とともに示す。

- 倍精度密行列データを単精度の密行列と倍精度の疎行列 (CSR 形式) に分割
- 単精度密行列同士の行列-行列積計算を BLAS SGEMM で実施
- 倍精度疎行列と単精度密行列および倍精度疎行列同士の行列-行列積計算を密ベクトル-疎ベクトルの内積として疎行列ライブラリで実施

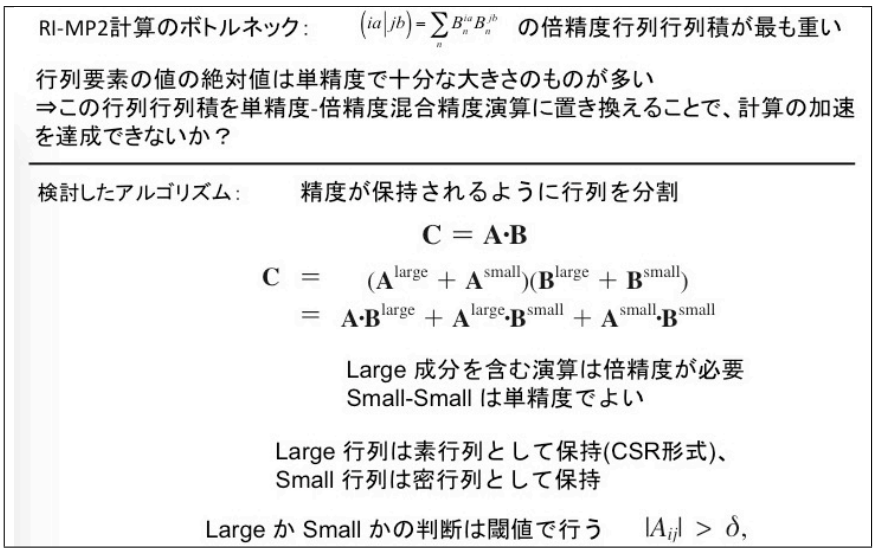


図 4.34: 密行列-密行列積演算の混合精度計算アルゴリズム

上記のアルゴリズムを実装した場合の演算性能を見積もるために、行列要素の単精度の割合とハードウェアアーキテクチャのスペックをパラメータとした性能推定モデル式を構築した。次に、本モデル式を用いて、京、FX100、富岳の各場合における性能を推定し、性能向上効果の検討を実施した。富岳の場合には、全行列要素のうち倍精度要素が 5%以下ならば、元のコードからの高速化が期待される見積もり結果となった。

この見積もりでは、行列要素の倍精度/単精度要素の比率はパラメータで与えているため、見積もりの精度が低い。そのため、実用的な計算における行列要素の絶対値の大きさと倍精度/単精度要素の比率を調査した上で、性能モデルの再検証と性能推定を行うことにより、この実装の妥当性を検証する必要がある。そこで、問題規模の異なるテスト分子に対して 3 中心積分行列要素の計算を行い、行列要素の大きさと倍精度/単精度要素の比率の調査を行った。図 4.35 および表 4.37 にコロネン 2 量体 $(C_{24}H_{12})_2$ の RI-MP2/def2-SVP 計算 (72 原子、794 原子軌道、158 占有軌道、636 仮想軌道、2640 補助基底関数) の際に倍精度疎行列で保持される行列要素の値および大きさに対する分布を示す。行列要素を単精度として取り扱うカットオフ閾値 δ を 10^{-8} にした場合は、倍精度疎行列に保存する必要がある行列要素数の割合が 0%となり、全ての行列要素を単精度密行列として取り扱うことが可能である。$\delta = 10^{-9}$ の場合でも、倍精度要素数は 0.2%となり、ほぼ全ての行列要素を単精度密行列として取り扱うことが可能である。これらの閾値に設定しても計算

結果の精度保証がされる場合、行列-行列積演算の混合精度化による高速化の効果が期待される結果を示している。

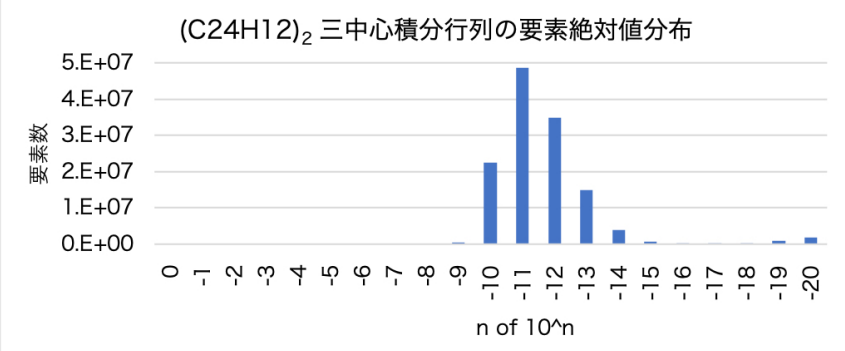


図 4.35: コロネン 2 量体 $(C_{24}H_{12})_2$ の RI-MP2/def2-SVP 計算における 3 中心積分行列の要素分布

表 4.37: コロネン 2 量体 $(C_{24}H_{12})_2$ の RI-MP2/def2-SVP 計算における 3 中心積分行列の要素分布

行列要素の大きさ	要素数	全要素素数に対する割合 [%]
0	0	0.0%
-1	0	0.0%
-2	0	0.0%
-3	0	0.0%
-4	0	0.0%
-5	0	0.0%
-6	0	0.0%
-7	0	0.0%
-8	14	0.0%
-9	523177	0.2%
-10	22400560	8.4%
-11	48597765	18.3%
-12	34873455	13.1%
-13	14946210	5.6%
-14	4025943	1.5%
-15	724666	0.3%
-16	104922	0.0%
-17	24634	0.0%
-18	166672	0.1%
-19	1026375	0.4%
-20	1854764	0.7%

4.6.6　RI-MP2 計算における密行列データ圧縮によるネットワーク通信量削減の検討

前項 4.6.5 の浮動小数点演算の混合精度化を検討する中で、計算に必要な行列データも単精度の密行列と倍精度の疎行列 (CSR 形式) に分割することにより、倍精度密行列のみを用いる時よりもデータ量を削減できることが明らかとなった。この圧縮されたデータ構造を用いた場合、通信データ量の削減も期待される。そこで通信高速化効果を見積もるための性能推定モデル式を構築し、富岳での性能を推定した。富岳では倍精度行列要素が 80%以下ならばデータ圧縮による高速化が期待される結果を得た。次に前述のテスト計算結果 (図 4.35 および表 4.37) を用いて通信量削

減効果を見積もったところ、ほぼ全ての行列要素を単精度として扱えるため、データの単精度化による通信性能削減効果が期待できることがわかった。

4.6.7 RI-MP2 計算の単精度・倍精度の混合精度化アルゴリズムのプロトタイプ実装および性能評価

前項までの検討により、行列要素の絶対値に閾値を設けて行列要素をソートして倍精度成分と単精度成分に分離して行列積計算を進める手法はその有効性が期待され興味深いものであるものの、当然ながら計算する分子モデル毎にその閾値を決定する必要があり、数理学的に妥当かつ量子化学的な見地からも適正と判断できる汎用的な閾値計算式を決定することは容易でないと考えられる。

NTChem/RI-MP2 への混合精度化の実装においては、より量子化学的な見地からの判断を重視して進めることとして、電子反発積分アルゴリズムのどの部分に対しては単精度計算を適用しても十分な計算精度が得られるか、十分な計算性能の向上効果が得られるかの判断を基にした実装を行った。具体的には倍精度計算を基本とする多数のアルゴリズムステップで構成される NTChem/RI-MP2 の主要なステップについて単精度版も各々作成し、後方から前方に向けて計算の単精度化を段階的に適用した場合に、どの段階まで単精度が可能であるかの評価を、多数の分子モデルを用いた数値実験により実施し、計算精度と計算性能の双方の見地から妥当な単精度計算区間を決定するというアプローチをとった。

まずプロトタイプ実装として、4 中心積分計算の際の行列-行列積を BLAS ライブラリの DGEMM を用いた倍精度密行列計算を SGEMM を用いた単精度密行列計算へ置換えた。さらに、4 中心積分の際に必要な 3 中心積分データの 1 対 1 通信 MPI_ISend, MPI_IRecv も倍精度から単精度へとデータ型の変更を行った。計算精度、演算時間、および通信時間の測定には、Intel Xeon Haswell 環境 (ノード数: 4 ノード, CPU: Xeon E5-2697 v4 2CPU/ノード, メモリ: DDR4 256GB/ノード, ネットワーク: InfiniBand FDR を用い、代表的なテスト分子を対象に混合精度化の効果の評価を行った (表 4.38)。いずれのテストケースにおいても、元の倍精度版を用いた際と比較して混合精度化による誤差の絶対値が 1.0E-6 Hartree 以内となり、実アプリケーションの際に影響が及ばない範囲となった。また、混合精度版を用いた際の実行時間は倍精度版の結果と比較して 1.5-2.0 倍高速化する結果となった。

表 4.38: RI-MP2 倍精度版コードおよび混合精度版コードを用いた際の計算精度、実行時間、性能倍率 (Intel Xeon Haswell 環境、RI-MP2/cc-pVTZ 計算)

問題規模 (分子)	$C_{24}H_{12}$	タキソール	$C_{54}H_{18}$	$(C_{54}H_{18})_2$	$C_{60}@C_{60}H_{28}$
混合精度化による誤差 [Hartree]	6.0E-7	8.4E-7	-3.5E-8	-1.3E-7	6.1E-6
混合精度版実行時間 [秒]	5.9	46.6	96.7	1986.5	2717.9
倍精度版実行時間 [秒]	8.6	78.5	172.5	3821.9	5356.7
性能倍率	1.5	1.7	1.8	1.9	2.0

次に、テスト計算における各演算・通信区間における単精度化の効果を調べた。表 4.39 の例の様に、いずれのテストケースにおいても、元の倍精度版で演算・通信に 8 秒以上要している区間については約 2 倍の高速化が達成されており、演算と通信の単精度化による高速化の効果が確認された。特に演算律速区間である 4 中心積分計算 (4c Ints calc.) については倍精度版では 3708 秒であったのに対し、混合精度版では 1825 秒であり大幅な時間短縮効果が確認された。また、通信

律速区間である 4 中心積分通信 (4c Ints comm.) についても、倍精度版では 1229 秒であるのに対して、混合精度版では 621 秒であり、こちらも大幅な時間短縮効果が確認された。

表 4.39: RI-MP2 倍精度コードおよび混合精度コードを用いた際の演算・通信区間の実行時間と性能倍率 (バッキーキャッチャー $C_{60}@C_{60}H_{28}$ RI-MP2/cc-pVTZ 計算)

ルーチン名	混合精度実装	混合精度 実行時間 [秒]	倍精度版 実行時間 [秒]	性能倍率
RIMDInt3c_calc		48.68	49.89	
MOInt3c_tran_1 (DGEMM)		28.85	28.79	
MOInt3c_tran_2 (DGEMM)		27.22	27.33	
MOInt3c_comm_1a,b (MPI_ISend)		15.56	15.12	
AOInt2c_calc		0.18	0.17	
Inv2c_DPOTRF		2.56	2.53	
Inv2c_DTRTRI		2.29	2.28	
MOInt3c_comm_2 (MPI_ISend)	実施	4.59	8.27	1.80
MOInt3c_tran_3 (DGEMM → SGEMM)	実施	32.65	70.43	2.16
MOInt3c_comm_3 (MPI_ISend)	実施	0.62	0.04	0.06
MOInt4c_calc (DGEMM → SGEMM)	実施	1824.61	3708.24	2.03
MOInt4c_comm (MPI_AllReduce)	実施	621.26	1228.72	1.98
MP2Energy_calc		41.84	56.03	1.34
MP2Energy_comm (MPI_AllRecude)		0.08	6.30	78.75
Total time		2717.93	5356.70	1.93

本プロトタイプ実装を用いた性能評価の結果、十分な精度を保ちつつ高速化が達成される結果となった。そこで、さらに実装を進めて、4 中心積分計算の区間（MOInt4c_calc ルーチン）に加えて、3 中心積分の変換（MOInt3c_tran_1、MOInt3c_tran_2、MOInt3c_tran_3 ルーチン）の 3 区間に対しても同様に、行列-行列積を倍精度密行列計算（DGEMM 利用）から単精度密行列計算（SGEMM 利用）へ置換えた実装を行った。またその際には計算の下流の工程である 4 中心積分計算から遡った 3 中心積分の変換を下流から順に単精度化した 5 種類の計算精度モデルを検討した。倍精度計算（計算精度モデル DP）と混合精度計算（計算精度モデル SP3、SP23、SP123b、SP123a）に対して、計算区間でどの GEMM ルーチンを用いたかを表 4.40 に整理する。

表 4.40: 計算精度モデルと採用する行列積のタイプの対応

計算精度モデル	MOInt3c_tran_1	MOInt3c_tran_2	MOInt3c_tran_3	MOInt4c_calc
SP123a	SGEMM	SGEMM	SGEMM	SGEMM
SP123b	DGEMM	SGEMM	SGEMM	SGEMM
SP23	DGEMM	DGEMM	SGEMM	SGEMM
SP3	DGEMM	DGEMM	DGEMM	SGEMM
DP	DGEMM	DGEMM	DGEMM	DGEMM

それぞれの混合精度モデルの数値誤差の影響と高速化の効果を調べるために、ナノグラフェン C24H12 と C54H18 の RI-MP2/cc-pVTZ 計算を Intel Xeon Haswell 環境 (ノード数: 4 ノード, CPU: Xeon E5-2697 v4 2CPU/ノード) で実施した結果を表 4.41 にまとめる。計算結果の数値誤差の影響については、SP23 と SP3 では実アプリに十分な 1,0E-6 Hartree の精度を満たす結果が得られたが、SP123b, SP123a と SGEMM の演算区間が増加するに連れて、数値誤差が増大し実アプリに使用するのに不十分な精度となることが明らかとなった。次に、計算の高速化については、SP3 は DP と比較して 1.4 倍以上の高速化の効果が確認されたが、SP123a, SP123b, SP23 は

SP3 と比較すると若干の高速化に留まることが明らかとなった。以上の計算結果の数値誤差の影響と高速化の効果を考慮すると、実アプリに用いることが可能な計算精度モデルとして、SP23, SP3 モデルが良好であることが示唆された。一方で、実アプリで対象となる様々な分子系での計算での数値誤差の影響については更なる精度評価が必要となる、そこで計算精度を検証するのによく用いられる代表的な分子系のベンチマークセットに対して、計算結果の精度の網羅的かつ体系的な検証を実施した。結果については次項で報告する。

表 4.41: 混合精度モデル (SP123a, SP123b, SP23, SP3) における倍精度モデル DP からの数値誤差と計算加速度

分子	$C_{24}H_{12}$	$C_{54}H_{18}$
誤差 [hartree]		
SP123a	-4.9.E-03	-8.2.E-01
SP123b	-6.7.E-04	-9.1.E-02
SP23	6.0.E-07	-3.5.E-08
SP3	8.1.E-08	3.4.E-07
実行時間 [秒]		
SP123a	5.9	97.0
SP123b	5.6	96.2
SP23	5.9	96.7
SP3	6.2	101.0
DP	8.6	172.5
計算加速度		
SP123a	1.5	1.8
SP123b	1.5	1.8
SP23	1.5	1.8
SP3	1.4	1.7

4.6.8 RI-MP2 計算の単精度・倍精度の混合精度化アルゴリズムの数値検証

前項 4.6.7 のプロトタイプ実装により、単精度・倍精度の混合精度化アルゴリズムが複数選択可能となった。その効果について、より網羅的な評価を行うため、表 4.40 に記載された計算精度モデルを用いた計算を様々な分子系へと適用し、倍精度計算（計算精度モデル DP）と単精度計算（計算精度モデル SP3、SP23、SP123b、SP123a）の計算結果を比較することにより、単精度計算による数値誤差の検証を行った。

RI-MP2 計算の特徴としては、計算コストを抑えて大きな分子の計算ができることと、平均場近似の Hartree-Fock 法では正しく記述できない、弱い分子間相互作用を正しく記述できることが挙げられる。ターゲット問題でも、グラファイトのモデル分子である平面構造の $C_{150}H_{30}$ 分子を 4 面重ねた (A-B stacking 構造) 分子系の RI-MP2 計算を行い、弱い面間の分子間相互作用エネルギーを求める。そこで、様々な分子系の RI-MP2 計算を行い、分子間相互作用エネルギーを算出し、前述の計算精度モデルを用いた結果を比較することにより、RI-MP2 計算より算出される分子間相互作用エネルギーの単精度計算の数値検証を行った。

様々な種類の分子間相互作用について検証を行うべく、多くの弱い相互作用を取り扱ったデータセットを複数用いて数値検証を行った。それぞれのデータセットに含まれている分子系はそれぞれ二つの分子が近接し、共有結合ではない弱い相互作用で集まる超分子である。分子 1 と分子 2

に形成される超分子 1-2 の分子間相互作用エネルギー (E_{bind}) は、超分子 1-2、分子 1 と分子 2 についてそれぞれエネルギー (E) を RI-MP2 計算より求めた上で、以下の式より算出する。

$$E_{bind} = E(\text{超分子 1-2}) - E(\text{分子 1}) - E(\text{分子 2})$$

表 4.42: データセットを用いた RI-MP2 分子間相互作用エネルギー計算における倍精度モデル (DP) と各混合精度モデルとの平均誤差絶対値 (kcal/mol)

データセット	SP3	SP23	SP123b	SP123a
S66x8	2.16E-05	2.09E-04	4.29E-04	6.51E-04
Water Cluster	1.80E-05	4.97E-04	5.49E-04	6.13E-04
X40	1.58E-05	1.17E-04	3.32E-04	5.62E-04
L7	8.98E-04	1.60E-03	1.51E+00	7.05E+02

RI-MP2 計算で分子間相互作用エネルギーを算出し、倍精度計算（計算精度モデル DP）と単精度計算（計算精度モデル SP3、SP23、SP123b、SP123a）の計算結果を比較することにより、単精度計算による数値誤差の検証を行った。基底関数としては、ターゲット問題と同じく cc-pVTZ を用いた。まず、小さな分子系 66 種類について分子間距離を 8 通り変化させるデータセット S66x8 データセット [47] を用いて数値検証を行った。分子間距離を変化させることにより、分子間相互作用エネルギーのわずかな変化についても調べることができるため、より精密な検証が可能となる。結果を表 4.42 に示す。その結果、分子間相互作用エネルギーにおける倍精度計算と単精度計算の平均絶対誤差は全ての計算精度モデルにおいて 0.0001 kcal/mol 以下となった。実用的な計算において、0.1 kcal/mol の計算精度を保証できるのであれば十分であることを考慮すると、この誤差は無視できるほど小さい。

次に、様々な水分子の集合分子系 38 種類を取り扱った Water Cluster データセット [45] を用いて検証を行った。水分子が 2-10 個まで含まれる集合分子を取り扱っており、S66x8 よりも大きな分子も含まれている。このデータセットの結果についても全ての計算精度モデルにおいて平均絶対誤差が 0.0001 kcal/mol 以下となった。

さらに、高周期のハロゲン原子を含む分子系 40 種類を取り扱った X40 データセット [46] についても同様の検証を行った。このデータセットの計算結果もこれまでのデータセット同様、平均絶対誤差が 0.0001 kcal/mol 以下となり、実用的な計算において、小さな分子系の弱い分子間相互作用計算に単精度計算を用いても全く問題ないことが確認された。

最後に、比較的大きい分子系の分子間相互作用を取り扱った L7 データセット [44] において検証を行った。RI-MP2 計算を用いて分子間相互作用エネルギーを算出したところ、倍精度計算と単精度計算の誤差は計算精度モデル SP3 と SP23 については平均絶対誤差が 0.002 kcal/mol 以下となった。しかし、計算精度モデル SP123b と SP123a を用いると 1 kcal/mol 以上の大きな誤差となった。一連の検証結果から、分子サイズを大きくするに従って誤差が増大し、比較的大きな分子系の分子間相互作用計算においては、単精度計算の置換えに注意を払う必要があることが明らかとなった。

計算コストの観点から考慮すると、DGEMM から SGEMM に置き換えた演算区間の中で 4 中心積分計算の区間（MOInt4c_calc ルーチン）が演算時間の大半を占め、次いで、3 中心積分の最後の変換（MOInt3c_tran_3 ルーチン）の演算時間の割合が大きい。ターゲット問題の計算においては、計算コストの高いこれら二つの演算区間を単精度計算にすることができれば、計算コストの大幅な削減が期待できる。それに対して、上記の演算区間を除いた二つの 3 中心積分変換区間

（MOInt3c_tran_1、MOInt3c_tran_2 ルーチン）を単精度化することは大きな分子系の計算では計算精度低下のリスクが大きく、それらの演算時間も全体の計算時間に占める割合は小さいため、単精度化する利点がない。

これらのことから詳細設計 (3) の期間においては演算時間の長い 4 中心積分計算区間（MOInt4c_calc ルーチン）と 3 中心積分計算の最後の変換区間（MOInt3c_tran_3 ルーチン）の 2 区間の演算を倍精度から単精度へ置き換えた計算精度モデル SP23 が、比較的大きな分子系の分子間相互作用計算においても計算誤差が小さく、計算精度を確保しつつ計算コストの大幅な削減につながる期待がもてるとの判断をえた。一方で、分子サイズの大規模化に伴う計算誤差が増大する一般的な傾向は無視できないため、詳細設計 (4) ではさらに分子規模が大きいターゲット問題について、SP23 モデル・SP3 モデルを中心に計算精度モデルを評価し、RI-MP2 計算におけるこのような倍精度・単精度混合計算の妥当性を確認した。

詳細設計 (4) の期間においてターゲット問題を用いて SP23、SP3、DP の 3 つの計算精度モデルのエネルギー値を計算したところ、ターゲット問題における DP からの誤差は表.4.43 に示すように、SP23 では 3.0kcal/mol、SP3 では 0.041kcal/mol であった。この結果からターゲット問題の規模での RI-MP2 計算において SP3 計算精度モデルが分子間相互作用の議論に必要な計算精度を維持できることが確認されたが、SP23 計算精度モデルの妥当性についてはさらに検討の余地があるとの評価となった。コデザイン版として完成し、最終的に対京性能倍率の評価に用いたのは SP3 計算精度モデルを採用した NTChem/RI-MP2 モジュールである。

表 4.43: ターゲット問題の RI-MP2 相関エネルギー (a.u.) と倍精度 (DP) との誤差 (kcal/mol)

計算精度モデル	相関エネルギー (a.u.)	DP との誤差 (kcal/mol)
倍精度 (DP)	-105.2921859	0.0
混合精度 (SP23)	-105.2970498	3.0
混合精度 (SP3)	-105.2921208	0.041

4.6.9 RI-MP2 計算における 3 中心積分計算の高速化

NTChem/RI-MP2 を用いたターゲット問題の計算時間全体に占める 3 中心積分計算 (電子反発積分計算) 区間の計算時間の割合は低いが、他の分子モデルの計算においては相対的に 3 中心積分の計算時間の割合が高くなる場合があり、また重点課題 5 において他のアプリケーションでも広く用いられている共通の計算基盤として位置付けられるため、3 中心積分計算の高速化が必要であるとの認識に至った。NTChem/RI-MP2 の当初の実装においては McMurchie-Davidson(McD) アルゴリズム [35] に基づく独自の 3 中心積分計算ルーチンを採用していたが、改めて 3 中心積分計算方法に関する調査研究を行い、分子科学研究所の石村和也研究員が開発している SMASH 量子化学計算プログラムの積分計算ライブラリが高速計算を実現するためのアルゴリズムを有し、NTChem/RI-MP2 計算プログラムに組み込み可能なプログラム構造を有しているとの評価結果を得た。この評価をうけ、SMASH の積分計算ライブラリを NTChem/RI-MP2 にルーチンとして組み込み、従来の独自ルーチンと SMASH ルーチンとを実行時に選択して計算することが可能となる実装を行った。

3 中心積分計算では特に高い角運動量関数を基底とする積分の計算コストが高くなる。SMASH の積分カーネルコードは角運動量の低い基底の低い積分と高い積分を異なるアルゴリズムで計算

することにより、計算コストの削減に成功している。角運動量の低い積分では Pople-Hehre(PH) アルゴリズム [43] と McD アルゴリズムを組み合わせた石村研究員開発のアルゴリズム (IN) [26] を、角運動量の高い積分では Rys 求積法を用いたアルゴリズム (DRK) [20] を採用している。

PH 法は基底関数中心間の二つのベクトルを座標変換することによって、積分を計算するための演算量を減らし、積分計算を高速化している。また、McD アルゴリズムはエルミートガウス関数と漸化関係式を導入することにより、計算コストを大幅に削減している。IN 法ではこの二つのアルゴリズムを組み合わせることで高速な計算を実現している。座標変換したのちに (SUBROUTINE int2phmd)、McD 法を用いた積分の計算を行う。不完全ガンマ関数 (誤差関数) を区分求積法で算出し、その不完全ガンマ関数からエルミートガウス関数を求めた上で、電子反発積分を計算する。SMASH 積分カーネルでは高速計算を実現するため、それぞれの積分の角運動量 ((ss|ss)、(ps|ps)、(pp|pp)、(dd|dd) など) に応じて専用ルーチンを用いてこれらの計算を実行している (SUBROUTINE int2ssss、int2psps、int2pppp、int2dddd など)。

DRK 法は、数値積分を用いることにより、高い計算コストの要因となる不完全ガンマ関数の計算を回避する手法である。角運動量が高い基底の積分計算を行うときに威力を発揮するため、SMASH においても角運動量の高い積分に用いている。Rys 求積法 (SUBROUTINE rysquad) から出発して、漸化式を用いて中間積分を計算し、それら中間積分から電子反発積分を算出する (SUBROUTINE int2rys2)。

導入した SMASH ライブラリルーチンを用いてターゲット問題の 3 中心積分区間の計算時間を FX100 システム上で試験的に短時間評価した結果、当初のプログラムでは 1.91 秒かかっていたのが、SMASH ライブラリルーチン版では 0.49 秒となり大幅な時間短縮効果が得られた。

この結果をもとに、以降の NTChem/RI-MP2 コデザインにおいては 3 中心積分計算を SMASH ライブラリルーチンを用いて進めることとした。公開されているバージョンの SMASH ソースプログラムを ASIS の状態で NTChem/RI-MP2 に組み込んで、FX100 システム上で 3 中心積分区間を詳細プロファイル分析を実施した結果、以下の改善がさらに望まれることが判明した。

- 演算待ち時間削減のための命令スケジューリング改善
- 整数・浮動小数点 L1D アクセス待ち時間削減のための命令スケジューリング改善
- 整数・浮動小数点 L2 アクセス待ち時間削減のためのデータアクセスの改善

これらの改善課題に対してコンパイラオプションによるチューニングおよびソースプログラム修正によるチューニングを各々試験的に実施し、以下の効果を得た。

- コンパイラオプションによるチューニング

  最内ループの回転数が 10 以下のため、-Kshortloop を指定して小さいループ向けの最適化 (ソフトウェアパイプライニングの抑止) を適用して高速化の効果あり。

- ソースプログラム修正によるチューニング

  ループの削除による実行命令数の削減効果あり。

3 中心積分計算の FX100 計算時間とハードウェア性能統計データを当初の McD 実装ルーチン版、SMASH ライブラリ組み込み版、SMASH にチューニングを実施した版で比較した結果を表 4.44 に示す。

SMASH ライブラリルーチンはそれ自体が相当量のソースプログラム行数を含み (1 万行以上)、ライブラリの中でさらに計算負荷特性が異なる複数のサブルーチンが呼び出されるため、今後の

表 4.44: 3 中心積分計算の高速化

ソースプログラムのレベル	当初版	SMASH 版	SMASH チューン版
実行時間 (秒)	1.91	0.47	0.41
実効性能比 (%)	0.28%	0.98%	1.13%
L1 ビジー率 (%)	27%	24%	27%
L2 ビジー率 (%)	6%	4%	5%
メモリビジー率 (%)	0%	0%	0%
L2 スループット (GB/s)	36.51	25.56	27.87

富岳性能評価に向けて、SMASH ライブラリの内部をさらに細かく区間化し、中でも経過時間が長くかかる数区間を改めてカーネル化して、性能シミュレータによる性能評価・最適化を行う方針とした。

4.6.10　入力ファイル読み込みの高速化と選択切り替え

ターゲット問題の分子軌道係数ファイルサイズは、デフォルトのテキストデータでは 9.4GB であり、京 1 ノードで読み込み時間を測定すると 588.8 秒であった。分子軌道係数ファイルのフォーマットをバイナリに変更することにより、ファイルサイズは 2.9GB まで削減され、さらに書式変換の処理が不要となるため、京 1 ノードでの読み込みは 82.6 秒へ短縮できた。このため富岳コデザイン版では分子軌道係数の読み込みはテキストデータでもバイナリデータでも可能とし、プログラム実行時に選択できるようにプログラム仕様と入力仕様を変更した。

4.6.11　ターゲットジョブのプロセス構成再精査

前出の 4.6.4 節で説明されたように詳細設計期間に検討されたメモリサイズ削減案を全て実装した結果、当初案の 82928 プロセスを用いた京上でのターゲット問題実行時のメモリサイズはプロセスあたり 7.5GB となり、1 ノードあたり 4 プロセスを割り当てる場合、OS の領域と合計するとノードあたりのメモリ容量に対して余裕がない評価となった。

一方、分散処理のロードインバランスを考慮して、異なるプロセス構成あるいはノード上のプロセスレイアウトを検討することで計算効率を向上させる可能性についても議論が進んだ。前出 4.6.4 節で示されたプロセス数 71280 と二次元分割パラメタ NProcsMat 値 8 を用いたジョブが実行されたのは、演算・通信負荷の負荷分散の観点で、仮想軌道数の整数倍のプロセス数を採用することによって計算負荷が高い 4 中心分子積分における行列積サイズ及び処理回数を均等にできる期待から、仮想軌道数 17820 の 4 倍である 71280 を採用したためである。

上記の背景のもと、メモリサイズの制約の下でターゲットジョブを富岳の上で最も効率よく実行できると推定されるジョブのプロセス構成、ノード内レイアウトを決定するための網羅的な探索を実施して構成の再精査を行った。すなわち利用するノード数・プロセス数・スレッド数・仮想軌道分散数・演算のブロック分割数の組み合わせに対して、利用できる最大メモリサイズと 4 中心分子積分区間の行列積演算における行列のサイズ・アスペクト比を拘束条件とした網羅的な検索を行うための NTChem/RI-MP2 のサブセットプログラムを作成して実行した。

富岳において推奨されるノードあたり 4 プロセス x 12 スレッドを起動する標準的なレイアウトの他、ノードあたり 2 プロセス x 24 スレッド、およびノードあたり 1 プロセス x 48 スレッドのいわゆる fat なプロセスレイアウトについても網羅的な検索を実施した結果、多数の構成案が得られた。それらの中からターゲットジョブの構成案として改めて 6 案に絞り込みを行った。表 4.45

に絞り込んだ構成案と、その密行列属性を示す。各構成案とも全て SP3 計算精度モデルを採用している。最終的には一番下の行に示す構成がターゲットジョブとして選択され、対京性能倍率の評価に用いられた。

表 4.45: ターゲットジョブのプロセス構成再精査 (全て SP3 計算精度モデル)

構成案	Nprocs	スレッド /プロセス	NProcs Mat	NProcs MO	メモリ GB /process	メモリ GB /node	密行列 m	n	k
当初案 82928	82928	12	16	5183	7.50	30.0	7440	3263	7440
分割案 71280	71280	12	8	8910	6.08	24.3	1860	1860	6525
分割案 71288	71288	12	8	8911	6.86	27.4	3720	3720	6525
Fat 35640x2	35640	24	8	4455	10.1	20.3	7440	7440	6525
Fat 20376x1	20376	48	8	2547	14.2	14.2	13020	13020	6526
Fat 17820x1	17820	48	4	4455	13.7	13.7	7440	7440	13049

これらのジョブ構成案の演算量は等しいが、計算性能と通信パターンはそれぞれに異なる。行列演算の実効／ピーク性能比はしばしば行・列のサイズとアスペクト比に依存し、また通信の待ち時間はメッセージ長と通信頻度により変化する。表 4.46 に通信待ち時間が大きい区間の通信統計情報の例を示す。

表 4.46: ターゲットジョブの通信統計情報

ジョブ構成案	routine	MPI ルーチン (API)	メッセージ長 (Bytes)	呼び出し回数	データ量 (GB)
分割案 71280	MOInt3c_comm_2 + MOInt3c_comm_3	MPI_ISend/IRecv	48546000	35642	1730.3
	MOInt4c_comm	MPI_Allreduce	13838400	17821	246.6

4.6.12 計算と通信のオーバーラップの明示的な促進

アシスタントコアを用いたノンブロッキング MPI 通信時間と計算コアを用いたの演算時間との積極的なオーバーラップを行なって、通信時間の見かけ上の待ち時間を短縮するために、富岳で利用可能な MPI ノンブロッキング通信促進 API を NTChem/RI-MP2 へ追加した。具体的には富岳における OpenMPI 4.0 に対する拡張ルーチンである FJMPI_Progress_start() および FJMPI_Progress_stop() をソースプログラム中に追加した。通信と演算のオーバーラップによる時間短縮効果は、小規模データを用いた測定では目立たないが通信時間・演算時間ともに長いターゲット問題規模のジョブでは効果が非常に大きく、ターゲットジョブの測定結果で示されるように、計算時間全体における通信の待ち時間は効果的に削減された。

4.6.13 PMlib 性能モニターライブラリの組み込み

各プロセスの計算性能を精密に測定するために PMlib 性能モニターライブラリ [36, 42] を NTChem/RI-MP2 に組み入れた。PMlib を利用することにより、利用者が環境変数で指定する性能カテゴリごとに富岳の詳細なハードウエア性能カウンター（HWPC) イベント情報を取得整理し、標準出力にそのレポートを表示することが可能となる。富岳でサポートされる PMlib の性能カテゴリとレポートされる HWPC 統計項目の概要を表 4.47 に示す。

表 4.47: 富岳で利用可能な PMlib レポート機能

性能カテゴリ	レポートされる HWPC 統計項目の概要
FLOPS	floating point operations in single precision and double precision FLOPS and peak % (sustained over peak performance)
BANDWIDTH	CMG local memory read and write counts, bytes and bandwidth
VECTOR	floating point operations in single and double precision by SVE and scalar/armv8 instructions percentage of SIMD (vectorized) floating point operations
CACHE	memory load and store instructions L1 data cache hits and misses L2 cache misses data access hit(%) in L1 cache and in L2 cache
LOADSTORE	memory load and store instructions grouped into: scalar instructions SVE and Advanced SIMD load and store instructions Advanced SIMD multiple vector contiguous structure load/store instructions SVE non-contiguous gather-load and scatter-store instructions percentage of SVE load/store instructions over all load/store instructions
CYCLE	total cycles and instructions percentage of fused multiply+add (FMA) instructions over all f.p. instructions performed instructions per machine clock cycle
USER	User provided argument values (Arithmetic Workload) are accumulated

プロセスごとに測定される HWPC 統計情報および経過時間情報は負荷インバランスの評価や計算と通信のオーバーラップの評価を行う上で有効である。後続の節で示されるように、最終的に富岳上で測定されたターゲットジョブにおいては非常に高い性能が確認された。PMlib はオープンソースソフトウエアであり、富岳以外の複数のシステムにおいても利用可能である。

4.6.14　試作機での小規模データ実行状況の確認

富岳試作機を利用できる環境が提供されたため、小規模データを用いて京や Intel Xeon クラスターでの計算結果の精度が十分に合致することをエネルギー値を指標として確認した。また期間限定で富岳 3 ラック構成（1152 ノード）での利用が可能となったため、原子数 1/4 規模のデータを用いた試験的な性能評価を行なった。この試験的な性能評価においては、システム利用上の制限から 4.6.12 で述べた明示的オーバーラップを抑制する必要があったため、最も効率の良い結果ではないが、原子数 1/4 規模のデータで 1 ジョブあたりの NTChem の性能が対京性能倍率が 5.8 倍、システム規模で評価する対京性能倍率が 44.5 倍であった。

4.6.15　コデザインによるシステムへの実装要請項目

本コデザインを通してシステムへフィードバックした実装要請・改善項目を以下に示す。

- 様々な入力行列に対する高効率計算を実現する DGEMM ライブラリ

  後述する富岳向け性能推定の節で詳しく説明されるが、NTChem/RI-MP2 計算において最も演算量が多く計算時間が長くかかる演算区間である 4 中心積分計算区間の計算内容は倍

精度密行列の行列-行列積演算 (DGEMM) である。基本設計時に設定したジョブの構成案でターゲット問題を計算する場合、行列は長方行列でありそのサイズは以下となる。

$$\Big(C\Big) = \Big(A\Big)\Big(B\Big), \quad \Big(A\Big) \equiv A(7440, 3263), \quad \Big(B\Big) \equiv B(3263, 7440) \text{ または } B(3263, 5580)$$

HPL(Linpack) などからの知見により、DGEMM の計算は各システムに固有な最適ブロックサイズで右辺行列をブロック化 (タイリング) を行い、L1 キャッシュ、L2 キャッシュなどメモリ階層の各レベルでデータ局所性・再利用性を高めることにより、より高い実効性能で計算を実行できることが知られていて、富岳においても同様な技法により実装される見込みである。

一般に DGEMM は HPL などの大規模行列を入力として想定した最適化実装が多い。NTChem のターゲット問題においてはそのような大規模行列と比較すると一桁程度サイズが小さく、このサイズに最適なタイリングパターンの検討が行われている。

一方 富岳ではターゲット問題よりも小さな分子を対象とした計算も頻繁に行われる予定であり、上記ターゲット問題の DGEMM サイズよりさらに小さな DGEMM の高性能化も次いで重要である。

NTChem を活用していく上で様々な規模の行列とその並列度 (プロセスとスレッドのパターン) に対して富岳が高い演算性能を発揮できるためには大きなサイズの DGEMM と小さなサイズの DGEMM とで高性能化・高効率化の実装手法が異なる可能性があるため、様々なサイズの DGEMM での高性能化を今後の検討課題として挙げ、DGEMM ライブラリの内部で最適なタイリングの選択と命令スケジューリングが行われるような DGEMM ライブラリの実装をアークテクチャへのフィードバックとして要請した。

- 高性能 SGEMM ライブラリの実装

  4.6.5、4.6.7、および 4.6.8 で検討した単精度計算を部分適用して計算を高速化を実現するためには単精度密行列の行列-行列積演算 (SGEMM) が上記の倍精度演算 (DGEMM) と同様に十分に高性能であることが必要である。上記と同じサイズの行列に対して DGEMM と同程度の対ピーク実効性能比を実現する SGEMM ライブラリの実装を要請した。

  詳細設計（３）の期間においては、性能シミュレータと論理検証エミュレータを併用して DGEMM および SGEMM の推定性能の精査を行った。

- 単精度と倍精度が混在する浮動小数点演算ループの SIMD 処理による高速化

  単精度の変数と倍精度の変数が混在する計算式を単一ループの中で処理する場合、SIMD 幅が等しくないため、通常のコンパイラ実装では単純な SIMD 命令の適用ができずにスカラー処理されることになる。コンパイラに対して 8 要素 SIMD 命令および 16 要素 SIMD 命令を適切に組み合わせを行い、ループ計算の高速化を実現する高度な最適化機能の実装を要請した。例題プログラムを 2 例提示して要請を行った。これについて、単精度と倍精度が混在する計算式を単純に 16 要素 SIMD 化するとレジスタ要素処理と型変換処理が増加するために汎用的な型変換を含んだ 16SIMD 化では性能の向上が見込めないが、

  倍精度＝倍精度.*op*.単精度演算列

  の形式のリダクション計算は SIMD 化による性能向上が期待できるため、該当機能を実装することとなった。

表 4.48: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (NTChem)

実装要請課題	対応
高性能 DGEMM ライブラリの評価と実装	ターゲット問題サイズおよび様々なサイズでの DGEMM の高性能・高効率実装を行う。性能シミュレータおよび論理検証エミュレータ を用いた詳細推定を実施。
高性能 SGEMM ライブラリの評価と実装	倍精度演算と単精度演算の混合あるいは切り替えによって行列積の計算時間短縮がどの程度期待できるかを定量化するため、SGEMM の性能評価を実施する。性能シミュレータおよび 論理検証エミュレータ を用いた詳細推定を実施。
単精度と倍精度の演算が混在するループの効率の良い SIMD 化実現	倍精度の変数に対する単精度の変数のリダクション機能を開発する。型変換コストが発生するため、SIMD 化の効果が大きいと評価可能なケースについて適用

4.6.16 富岳向け性能推定

NTChem/RI-MP2 の富岳向け性能推定の手法は 4.1.5 節に説明される性能推定手法を基本としたうえで、カーネル切り出し対象区間を特定するための原子数・原子軌道数を基にした演算量・時間の専用の推定式を採用したこと、DGEMM 計算部分に関して専用の推定式を採用していることなど、いくつか留意すべき点がある。それらの説明も含めて、以下に改めて性能推定の全手順を記述する。

4.6.16.1 ベースライン版コードの概要

本来であればコデザイン開始前の NTChem/RI-MP2 モジュールをベースラインとして位置付けるのが妥当であるが、前出の 4.6.2 項で説明した超並列実装を行ったことにより初めて 京の上でターゲット問題規模の超並列計算を実行するために必要な特性を有することになったため、この超並列実装版 4.6.2 を行った版の NTChem/RI-MP2 モジュールをベースライン版コードとして京および富岳における性能倍率比較のために用いることにする。ベースライン版コードにおける主な演算区間とその処理概略を表 4.49 に示す。

表 4.49: NTChem/RI-MP2 ベースライン版コードの演算区間と主な処理の概略

演算区間名	主な処理内容	演算コスト
RIMDInt3c_calc	3 中心積分計算 (独自カーネル)	$O(N^3)$
MOInt3c_tran_1	3 中心積分第 1 変換 (BLAS DGEMM)	$O(N^4)$
MOInt3c_tran_2	3 中心積分第 2 変換 (BLAS DGEMM)	$O(N^4)$
AOInt2c_calc	2 中心積分計算 (独自カーネル)	$O(N^2)$
Inv2c_DPOTRF	2 中心積分行列のコレスキー分解 (LAPACK DPOTRF)	$O(N^3)$
Inv2c_DTRTRI	コレスキー行列の逆行列計算 (LAPACK DTRTRI)	$O(N^3)$
MOInt3c_tran_3	3 中心積分第 3 変換 (BLAS DGEMM)	$O(N^4)$
MOInt4c_calc	4 中心積分計算 (BLAS DGEMM)	$O(N^5)$
MP2Energy_calc	MP2 相関エネルギー計算 (独自カーネル)	$O(N^4)$

ベースライン版コードは主に 9 つの演算区間で構成されている。RIMP2Int3c_calc では独自計算カーネルを用いた 3 中心積分計算を行う。AOInt2c_calc では独自計算カーネルを用いた 2 中心積分計算を行っている。これらの独自計算カーネルの実装には McD アルゴリズム [35] が採

用されている。3 中心積分計算では高い角運動量関数を基底とする積分の計算コストが高くなるが、McD アルゴリズムでは、電子反発積分はエルミートガウス関数の線形結合で記述され、計算に必要なエルミートガウス関数と線形結合の係数を漸化式で求めることにより、計算コストを大幅に削減している。NTChem/RI-MP2 の実装では、電子反発積分の算出の基となる不完全ガンマ関数 (誤差関数) を区分求積法で算出した (SUBROUTINE Int2_FFunc) のちに、その不完全ガンマ関数からエルミートガウス関数を求める (SUBROUTINE MDInt2_R0tuv)。次に漸化式より、線形結合の係数を計算した上で (SUBROUTINE MDInt2_ECoef1, MDInt2_ECoef2)、得られた係数とエルミートガウス関数の線形結合を計算し、電子反発積分を算出する (SUBROUTINE RIMP2_RIInt2_MDInt2_ERI3c_Sph)。最後に必要に応じてガウス関数をデカルト座標から球座標への変換を行う (SUBROUTINE MDInt2_CarToSph)。

Inv2c_DPOTRF では、AOInt2c_calc で計算した 2 中心積分行列のコレスキー分解を行う。ここでの演算カーネルには LAPACK の DPOTRF を用いている。Inv2c_DTRTRI では、Inv2c_DPOTRF で計算したコレスキー行列の逆行列計算を行う。この演算カーネルには LAPACK の DTRTRI を用いている。

MOInt3_tran_1、MOInt3_tran_2、および MOInt3_tran_3 では、RIMP2Int3c_calc で計算した 3 中心積分と Inv2c_DTRTRI で計算した行列データを用いて、3 中心積分変換計算を行っている。MOInt4c_calc では MOInt3_tran_3 で計算した 3 中心積分データを用いて、4 中心積分計算を行う。これら 4 つの計算カーネルは全て倍精度密行列の行列-行列積で処理することができるため、計算カーネルとして BLAS Level 3 の DGEMM を用いた実装としている。MP2Energy_calc では MPInt4c_calc で計算した 4 中心積分データを用いて MP2 相関エネルギー計算を行う。この処理は独自カーネルとして実装を行っている。

ベースライン版のコードにおける主な通信区間とその処理概略を表 4.50 に示す。

通信処理は MPI_COMM_WORLD が前出 4.6.2 項で説明した 2 階層の並列空間に対応して分割され、2 つのサブコミュニケータ MPI_COMM_MO および MPI_COMM_MAT 内で行なわれる。全プロセス数 N_{procs} はこれらサブコミュニケータに以下のように分割される

$$N_{procs} = N_{procsMO} * N_{procsMat}$$

N_{procs} は MPI_COMM_WORLD の全プロセス数、$N_{procsMO}$ は MPI_COMM_MO に割り当てられたプロセス数を、$N_{procsMat}$ は MPI_COMM_MAT に割り当てられたプロセス数を示す。

4.6.16.2　ターゲット問題の設定

NTChem/RI-MP2 の性能が富岳において京と比較してどの程度向上するかを評価するために、基準としての計算問題を以下のように設定した。これをターゲット問題と称する。

- ターゲット問題

  化学反応のメカニズムを解明し、エネルギー変換材料の候補物質スクリーニングを行うための高精度第一原理電子状態計算の典型的計算例として、720 原子、19680 電子軌道のカーボンナノグラフェン分子複合体 $(C_{150}H_{30})_4$ のエネルギー計算を実行する。この計算を 20 ケース実行するのに必要な時間を性能評価の基準とする。

表 4.50: NTChem/RI-MP2 ベースライン版コードの通信区間と主な処理の概略

通信区間名	通信処理の内容	通信コミュニケータ	通信データ量	通信回数
MO_comm	分子軌道係数行列の MPI_Bcast	MPI_COMM_WORLD	$O(N^2)$	1
RIMDInt3c_comm	3 中心積分の MPI_Allreduce	MPI_COMM_MAT	$O(N^3/N_{procsMO})$	$N_{procsMO}$
MOInt3c2_comm	3 中心積分の MPI_Allreduce	MPI_COMM_MAT	$O(N^3/N_{procsMO})$	$N_{procsMO}$
NBF_RI_rank_comm	補助基底関数シェルの指数の MPI_Allgather	MPI_COMM_MO	$O(N)$	1
MOInt3c_comm_1a	3 中心積分の MPI_ISend,MPI_IRecv	MPI_COMM_MO	$O(N^3/N_{procsMO})$	$N_{procsMO}$
MOInt3c_comm_1b	3 中心積分の MPI_ISend,MPI_IRecv	MPI_COMM_MO	$O(N^3/N_{procsMO})$	$N_{procsMO}$
MOInt3c_comm_2	3 中心積分の MPI_ISend,MPI_IRecv	MPI_COMM_MO	$O(N^3)$	$N_{procsMO}$
MOInt3c_comm_3	3 中心積分の MPI_ISend,MPI_IRecv	MPI_COMM_MO	$O(N^3)$	$N_{procsMO}/2$
MOInt4c_comm	4 中心積分の MPI_Allreduce	MPI_COMM_MAT	$O(N^4/N_{procsMO})$	$N_{procsMO}/2$
MP2Energy_comm	MP2 エネルギーの MPI_Allreduce	MPI_COMM_WORLD	1	1

4.6.16.3　カーネルコードによる京ベースライン性能の推定と実測

ベースライン版コードを用いて京の上でターゲット問題の計算を実行することによっていわゆるベースライン性能が定まるが、ベースライン版コードでターゲット問題を計算するためには 1 プロセスあたり 31.9GB のメモリ量が必要であり、京の計算ノードに搭載されたメモリ量 16GiB を超過し、そのままでは実行ができない。また基本設計時には、ターゲット問題の分子軌道データは得られていなかった。そのため、 4.6.3 節に説明したメモリ分割の実装が実現されて必要なメモリサイズが削減されることを前提として、その実装状態における実行時の性能挙動を示す性能評価用のカーネルコードをベースライン版コード中から切り出して測定・推定評価するという手法を採用した。

カーネルコードは、NTChem/RI-MP2 において計算処理 (演算あるいは通信) の負荷が高いと判断されるプログラム部分を独立したプログラムとして切り出して、ターゲット問題に対応する部分的計算を実現可能とする単体プログラムの集まりである。カーネルコードは計算処理の内容により演算カーネルと通信カーネルに大別される。カーネルコードの実行時間、すなわち演算カーネルの経過時間と通信カーネルの経過時間を合計したものが、ベースライン推定実行時間である。このベースライン推定実行時間が性能向上の基準値に用いられる。カーネルの一覧を表 4.51 に示す。

カーネルコードとして切り出す区間の選択は、主な演算区間と通信区間に対して実行時間の推定式を構築して行った。推定式はカーネルを切り出す目的のためだけに用いられるものであり、富岳向けの性能評価は節 4.6.16.4 で説明する手法により詳細に実施されることに留意しておかなくてはいけない。

演算区間についてはアルゴリズムに対する考察から概算推定式を立て、フィッティング手法を採用して演算時間の概算値を求めた。すなわち、原子軌道数 n に対する計算時間 T を以下と仮定する。

$$T_{\text{全区間}} = \sum T_k = \sum a_k n^{p_k}$$

表 4.51: 切り出しカーネルコード

区間	カーネルコード名	処理内容
演算	MOInt4c_calc_n1	4 中心分子積分計算
	MOInt4c_calc_n2	4 中心分子積分計算
	MOInt3c_calc	3 中心分子積分計算
通信	MOInt3c2_comm	3 中心分子積分の集団通信
	MOInt3c_comm_1a	3 中心分子積分の大域通信
	MOInt3c_comm_1b	3 中心分子積分の大域通信
	MOInt3c_comm_2	3 中心分子積分の隣接通信
	MOInt3c_comm_3	3 中心分子積分の隣接通信
	MOInt4c_comm	4 中心分子積分の集団通信
	MP2Energy_comm_1	MP2 相関エネルギーの集団通信
	MP2Energy_comm_2	MP2 相関エネルギーの集団通信

ここで、T_k : 区間の推定時間、a_k : 区間の係数パラメータ、p_k : 区間の次数

区間の次数 p_k は各演算区間の演算量のオーダーより決定し、係数パラメータ a_k はベースライン版コードを用いて問題規模を変えて実行時間 T_k を複数回測定した結果を元にフィッティング決定した。係数パラメータ決定用の測定には、ターゲット問題と類似した組成で実行可能なサイズの分子であるナノグラフェン $C_{54}H_{18}$ およびその 2 倍のサイズの分子 $(C_{54}H_{18})_2$ を入力データとして用いた。フィッティングにより係数パラメータが決定された推定式に対して、ターゲット問題に相当する原子軌道数 n を適用した推定時間を算出し、その推定時間における該当区間の占める比率が 1%以上となった区間を演算区間のカーネルコード切り出し対象として選択した。この結果、上記の 4 中心分子積分計算区間および 3 中心分子積分計算区間がカーネル化された。

通信区間に対するカーネルコード切り出しでは、主要な通信処理を含む区間を全て対象として採用し、測定値を直接採用する方針をとった。具体的な切り出し作業は、ベースライン版ソースプログラムにおける MPI 通信処理部分を抽出して、MPI 通信パターン毎 (API 毎) に分類して再構成した単独プログラム群として作成した。

上記手法により切り出したカーネルコードを用いて京におけるベースライン性能の推定を行った。基本設計においてターゲット問題の計算用に想定した利用プロセス数は京の全計算ノード数に近い規模の 82928 である。評価された京における NTChem/RI-MP2 のベースライン推定実行時間は表 4.52 の上段のようになった。

表 4.52: カーネルコードによる京ベースライン推定と実測された結果

ベースラインの測定方法	全体時間 [秒]	演算時間 [秒]	通信時間 [秒]	I/O 時間
京カーネル推定	7725.1	7608.9	116.2	-
京実測結果	10938	9073	1265	600

コデザインが進み、詳細設計（4）の時点でメモリの分散化がほぼ完了し、必要なメモリサイズが削減されるに従い、実際に京上でターゲットジョブを実行することが可能となった。測定されたターゲットジョブの計算時間は表 4.52 下段のようになった。

推定演算時間と比較して実際の演算時間は 19%長い結果となった。この差は、抽出したカーネルコードに対するフィッティングパラメタの精度が不十分であったこと、また推定ジョブと実測したジョブで異なる二次元分割パターンを用いる必要があり実測ジョブではプロセスあたりが処理する行列のサイズが行・列とも 1/4 のサイズになったことなどが影響したと考えられる。

通信時間は推定値に対して実際の通信時間が 10 倍以上かかり、1265 秒となった。この理由は実行時の隣接通信が効率よく行われるようなジョブのトポロジを確保できず、通信時間の遅延が大きく発生したノードペアが生じてしまったため結果的に通信時間のバリア待ち時間の長時間化が発生したものと推測されるが明確ではない。またベースライン推定時には考慮していなかった軌道データの読み込み処理が 600 秒程度必要であった。

4.6.16.4　性能電力予測ツールを用いた演算区間の性能推定

演算区間の富岳性能評価は、切り出し整備したカーネルコードに対して、FX100 システム上で詳細プロファイラ (FAPP) 測定を複数回行い、サイクルアカウンティングによる分析を実行し、その測定結果を富岳の性能を推定するために開発された専用の性能電力予測ツール を用いて、富岳の性能を推定する。

FX100 システムではサイクルアカウンティングに必要な情報を収集するために同一ジョブに対して詳細プロファイラを 11 回繰り返して情報を合算する必要がある。詳細プロファイラジョブの連続実行、統計情報の CSV 変換、CSV 情報の可視化の手順を連続で実施して富岳の性能推定結果を得ることになる。

性能電力予測ツールを用いた富岳の性能推定は上記手順によってほぼ自動的に行うことができるため、コデザインの様々な段階で実施された。また、コデザインの結果は既存システムにおける実行時間の短縮に結びつくことも多いことから、まず先行的に FX100 システムで性能解析と細かなチューニングを行い、効果的だと思われる版を性能電力予測ツールを用いて富岳向けに評価するという作業フローが定着した。

4.6.16.5　通信区間の性能評価

通信区間の富岳性能評価は通信区間カーネルコードを用い、京での測定をベースとした。

MOInt3c_comm_2,3、MOInt3c_comm_1a、MOInt3c_comm_1b、MOInt3c2_comm、MOInt4c_comm の各カーネルについてはターゲット問題を富岳で実行する場合と同じ条件 (MPI プロセス数 20732) で、京の上で実行し、得られた測定値をもとに以下の算出手順 1 および算出手順 2 の方法により富岳での推定値を算出した。また、MP2Energy_comm_1、MP2Energy_comm_2 通信カーネルについては算出手順 3 に示す専用の推定式によって算出した。

これらはともに 1 要素 (8 バイト) の MPI_Allreduce 通信を行うカーネルである。富岳において 1 要素の MPI_Allreduce 通信は、計算ノード内の CMG 計算コアまでデータ転送を行うことなく、Tofu の NIC レベルで実現される。

算出手順 1 京のノードあたりの通信スループット概算値

- MPI_ISend/IRecv 通信: $L_{msg}P_{node}N_{cc}/T_K$
- MPI_Alltoall 通信: $L_{msg}P_{total}/T_K$
- MPI_Allreduce 通信: $2 \times L_{msg}/T_K$

ここで通信スループット概算値は送受信の片方向分であり、各記号は以下を意味する。L_{msg} : メッセージ長 (平均バイト数)、P_{total} : 全 MPI プロセス数、P_{node}: ノード内プロセス数、T_K : 京の測定時間、N_{cc} : 同時通信数

算出手順２ 通信バンド幅の性能向上比によるスケーリング

算出したノードあたり通信スループットが 1GB/s を超える場合、その通信はバンド幅で律速されるものとみなし、京の実測値を富岳の通信バンド幅性能向上比 1.36 で割った結果を富岳の推定時間とする。通信スループットが 1GB/s を超えない場合は、京の実測値をそのまま富岳の推定時間とする。

算出手順３ 特定の通信タイプに対する推定

メッセージ長が 3 要素以下の非常に短い集団通信 MPI_Allreduce については Tofu Barrier Interface と呼ばれる専用のハードウェア資源を使用しツリー構造でリダクション処理およびブロードキャスト処理を行うことで実現される予定であるため、ハードウェア仕様に基づく独自の評価式を用いて推定する。MP2Energy_comm_1、MP2Energy_comm_2 通信カーネルがこれに該当し、それらはともに 1 要素 (8 バイト) の MPI_Allreduce 通信を行う。

通信区間カーネルコードについて上記の方法に沿って求めた通信区間の通信のサイズと頻度、京における測定値、および富岳の推定値を表 4.53 に示す。

表 4.53: 通信区間の通信パターンと富岳の推定値

区間名	MPI 通信タイプ	メッセージ長 (バイト)	通信回数	京測定値 (秒)	富岳推定値 (秒)
MOInt3c2_comm	MPI_Allreduce	265161600	2	0.4	0.5
MOInt3c_comm_1a	MPI_Alltoall	36	1	0.0	0.0
MOInt3c_comm_1b	MPI_Alltoall	133920	1	0.6	2.5
MOInt3c_comm_2	MPI_ISend/IRecv	48546000	2591	26.5	77.8
MOInt3c_comm_3	MPI_ISend/IRecv	48546000	2591	26.5	77.8
MOInt4c_comm	MPI_Allreduce	27676800	2591	58.8	68.0
MP2Energy_comm_1	MPI_Allreduce	8	1	0.0	0.0
MP2Energy_comm_2	MPI_Allreduce	8	1	0.0	0.0

全プロセスに対して各通信区間毎に通信待ち時間を測定した結果、１対１通信において通信待ち時間が特に長いプロセスがあった。この原因はベースライン性能測定において用いた MPI プロセスが Tofu トポロジ上で最適なレイアウトとなっておらず、プロセスペアによっては１対１通信のホップ数が最大で５となっているためであった。詳細設計（３）の期間においてはランクマップファイルの見直しと最適化をはかり、富岳において１対１通信のホップ数が最大でも２とすることが可能なマッピングを考案し、適用した。富岳での通信区間性能推定においてはそのコデザイン効果が見込まれている。また、これまで時間推定対象ランクをランク 0 で固定していた条件を、全ランク中最も通信時間が長いランクを特定して採用した。

詳細設計 (4) 時点の NTChem/RI-MP2 コデザイン最新版を用いて、ターゲット問題を 82928 プロセス、および 71280 プロセスを用いて計算する場合にジョブで実際に発生するメッセージ長および繰り返し回数を京の上で測定した。MOInt3c_comm_2 区間+MOInt3c_comm_3 区間 (MPI_ISend と MPI_IRecv の和) MOInt4c_comm 区間について測定した結果を表 4.54 に示す。基本設計時の推定値に比べて、82928 プロセスでは通信回数は同じであったが、メッセージ長が 4 倍長い結果となった。この 82928 プロセスでの測定はベースラインと同様に全区間を倍精度データを用いて計算した。71280 プロセスの測定値では、82928 プロセスに比べてメッセージ長が減少し通信回数が増加し、全データ量は ISend+IRecv では増加し、Allreduce では減少する。メッセージ長と通信回数とはジョブの構成案によってかわり、通信区間の経過時間を短縮することは興味深い課題と言え

るが、最終的に完成した NTChem/RI-MP2 コデザイン版においては 4.6.12 項で説明した計算と通信のオーバーラップが効果的に実装されたため、MOInt3c_comm_2 区間と MPI_ISend/IRecv 区間のノンブロッキング通信待ち時間はほぼ無視できる水準まで削減された。

表 4.54: NTChem/RI-MP2 コデザイン最新版のターゲット問題通信区間の通信パターン

プロセス数	区間名	MPI 通信タイプ	メッセージ長 (バイト)	通信回数	全データ量 (GB)
82928	MOInt3c_comm_2	MPI_ISend/IRecv	194451840	2592	504.0
(DP,16)	MOInt3c_comm_3	MPI_ISend/IRecv	194451840	2592	504.0
	MOInt4c_comm	MPI_Allreduce	442828800	2592	1112.3
71280	MOInt3c_comm_2	MPI_ISend/IRecv	48546000	17821	865.2
(SP3,8)	MOInt3c_comm_3	MPI_ISend/IRecv	48546000	17821	865.2
	MOInt4c_comm	MPI_Allreduce	13838400	17821	246.6

4.6.16.6　富岳の全区間合計推定時間

上述した演算区間および通信区間の性能推定作業はコデザインの経過に伴ってパラメータを変えながら多数回実施された。表 4.55 に詳細設計 (4) 期間における富岳の全区間合計推定時間を示す。4.6.16.3 の表 4.52 で示された 2 通りの京ベースライン実行時間、すなわち当初推定の基準値としていた京ベースライン推定実行時間（京カーネル推定）を基にした富岳推定時間と、実際に測定された京ベースライン実行時間（京実測時間）を基にした富岳推定時間とを併記している。京におけるベースラインの推定および実測は全て DP 計算精度モデルによるものであり、4.6.7 において詳細に検討された混合精度化アルゴリズムの効果は反映されていない。

表 4.55: NTChem の富岳推定計算時間 (通常モード、電力制御なし、プロセス数 82928)

全体時間 [秒]	演算時間 [秒]	通信時間 [秒]	I/O 時間 [秒]	BL 基準
1481	1254	227	0	京カーネル推定
1842	1495	227	120	京実測時間

4.6.16.7　DGEMM に対する推定手法と性能電力予測ツールの補正機能

NTChem/RI-MP2 計算において演算量が多い 4 中心積分計算区間のなかでも特に高い比重を占める行列積演算に関しては精度の高い評価が必要となる。富岳における行列積計算

$$\Big(C\Big) = \Big(A\Big)\Big(B\Big)$$

の最適化された実装では、入力行列 $(A), (B)$ を内部で小行列にブロック化し、小行列の積和演算部分において列ベクトルが L1 キャッシュに収まるデータレイアウトとなることを仮定して DGEMM の性能推定を行う。

まず FX100 上での測定値から富岳の性能推定を行う基本的な手法を説明する。

- 計算時間のモデル化

  DGEMM の計算時間が以下の 4 種類の処理量により決定可能であることを前提とする。

  - $T1$:レジスタ上のデータの積和演算量
  - $T2$:計算結果 C 要素の L1 へのストア量
  - $T3$:B の小行列の処理列を L1 にロードする量
  - $T4$:A の小行列を L2 にロードする量

  各演算量、ロード・ストア量は入力行列のサイズ $M, K, N[(A)_{M,K}, (B)_{K,N}]$ とブロック化された後の小行列のサイズ $m, k, n[(A)_{m,k}, (B)_{k,n}]$ を用いて表わされる。これらの各演算量あるいはロード・ストア量に係数を乗じて各処理時間を定め、その合計時間が DGEMM の計算時間であるとモデル化する。

  $$T_{DGEMM} = C1 * T1 + C2 * T2 + C3 * T3 + C4 * T4$$

- FX100 向けモデル係数の決定

  前項における $C1, .., C4$ の値を求めるために、行列のサイズと内部ブロックサイズ (タイルサイズ) を変えた複数パターンの DGEMM 引数条件を設定して、FX100 の上で実際に経過時間測定を 4 回行う。その結果、4 元の一次方程式から係数 $C1, .., C4$ が求まる。

- 富岳向けモデル係数の決定

  上記係数 $C1, .., C4$ を富岳の設計値を基に補間補正した係数 $C1', .., C4'$ を決定する。

  - $C1'$: $C1$ に対して FX100 と富岳の積和演算性能比を補正
  - $C2'$: $C2$ に対して FX100 と富岳の L1 キャッシュ転送性能比を補正
  - $C3'$: $C3$ に対して FX100 と富岳の L2 キャッシュ転送性能比を補正
  - $C4'$: $C4$ に対して FX100 と富岳のメモリ転送性能比を補正

- 富岳の経過時間・性能の推定

  上記で決定したモデル係数 $C1', .., C4'$ を用いて富岳における経過時間推定式を得る。

  $$postKT_{DGEMM} = C1' * T1 + C2' * T2 + C3' * T3 + C4' * T4$$

  この推定式は行列のサイズ M, K, N およびおよび内部でブロック化された後の小行列のサイズ m, k, n を変数とする関数であるので、これらのサイズとして具体的な数値を代入するとそのサイズにおける富岳の推定経過時間が求まる。

以上が FX100 上での DGEMM 測定値から富岳の性能推定を行う基本的な手法であるが、FX100 の上で DGEMM 引数条件を設定して経過時間測定を 4 回行う処理で 12 スレッドを用いた計算において十分に性能が高い結果を得るためには一般利用者では利用できないモードでセクターキャッシュを設定・利用する必要があることが判明した。つまり FX100 で通常得られる結果を直接利用した性能推定では妥当な評価が得られないことが判明し、上記手法を直接利用することが難しくなった。そこで実際の推定においては DGEMM の推定精度を向上するための DGEMM 専用モード補正 (SIMD/非 SIMD 命令の比率強制補正) 機能を追加した性能電力予測ツールを準備して富岳

の経過時間および計算性能を推定することになった。上述した計算時間のモデル化の各処理量に対してマイクロアーキテクチャが実現すべき性能挙動（レイテンシ・スループット）を論理的に織り込んだ推定式と対応すべき性格の補正機能といえる。

数学的な観点での計算性能は、計算量が定義式から $M*K*N$ であることから、以下で求まる。

$$M*K*N/postKT_{DGEMM} \text{ (Flops)}$$

詳細設計（3）においては、性能シミュレータ および 論理検証エミュレータを用いてターゲット問題の行列サイズに対する DGEMM の処理を命令ベースで精密に評価し、あわせて富岳のメモリレイテンシ最新値を補正に反映して上記手法の妥当性を確認した。

詳細設計（4）においては、4.6.4、4.6.8 で詳細に検討した新しいメモリー分割レイアウトおよび単精度・倍精度の混合精度化アルゴリズムの実装を全て行なったコデザイン最新版を用いてターゲット問題の計算特性分析に着手した。

ソフトウエア調整（1）においては、ジョブのプロセス構成再精査を実施し、4.6.11 に示した新しい候補群が複数発生したが、ソフトウエア調整（2）において富岳での試行計算が可能となり、最終的には表 4.45 の最も下の案を採用することが決定された。当然ながらジョブの構成、すなわち 2 次元のプロセス分割のパターンによって、1 プロセスが処理する行列のサイズと呼び出し回数が変わる。基本設計時の 82928 プロセス案と、4.6.4 で説明した 71280 プロセス案とについて主な行列積のサイズと実行回数を測定した結果を表 4.56 に示す。

表 4.56: 4 中心分子積分計算における行列積のサイズ及び回数の測定値（プロセス番号 0）

プロセス数	NProcsMat	行列 A 行数	行列 A 列数	行列 B 行数	実行回数
82928	16	7440	3267	7440	2271
		7440	3267	5580	321
71280	8	1860	6525	1860	17821

4.6.17　富岳で達成された性能倍率

上述したコデザインの成果を全て反映して富岳で達成された京に対する性能倍率及び消費電力を表 4.57 に示す。

表 4.57: NTChem の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
NTChem	70 倍	26 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

4.6.18　執筆者

本節の主な執筆者、および執筆協力者の氏名と所属を記す。（主な参加時期における所属）

表 4.58: NTChem の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
NTChem	70 倍		67 倍		49 倍		45 倍	
	24 MW	26 MW	21 MW	23 MW	18 MW	19 MW	16 MW	17 MW

4.6.18.0.1　執筆者（執筆責任者）

- 三上 和徳（理化学研究所）
- 河東田 道夫（理化学研究所）

4.6.18.0.2　執筆協力者（５０音順）

- 安藤 嘉倫（名古屋大学）
- 石村 和也（分子科学研究所）
- 稲垣 耕司（大阪大学）
- 上島 基之（神戸大学）
- 大西 裕也（神戸大学）
- 岡崎 進（分子科学研究所）
- 神谷 宗明（岐阜大学）
- 川島 雪生（理化学研究所）
- 澤田 啓介（理化学研究所）
- 土持 崇嗣（神戸大学）
- 中嶋 隆人（理化学研究所）
- 中塚 温（岐阜大学）
- 南 一生（理化学研究所）

4.7 重点課題 6 ADVENTURE のコデザイン

4.7.1 アプリケーションの特徴とコデザインの方針

4.7.1.1 アプリケーションの特徴

領域分割法（Domain Decomposition Method, DDM）構造解析プログラム ADVENTURE では、解析対象は互いに重なり合わない複数の部分領域に分割されており、部分領域間境界上節点の変位に関する連立一次方程式（インターフェース問題）を、共役勾配法（Conjugate Gradient, CG）法によって反復的に解く部分が主要処理部である。つまり、その主要処理部では、部分領域間境界での仮の強制変位から部分領域間境界での反力が計算され、その反力の合計から部分領域間での仮の強制変位が更新されるということが、部分領域間境界での反力の合計が十分にゼロに近付くまで繰り返される。さらに、この CG 反復の反復回数を減少させるために、部分領域毎の対角スケーリングと、コースグリッド修正の 2 段階からなる前処理も、この CG 反復の各回に行われる。この前処理は、BDD（Balancing Domain Decomposition）-diag 前処理と呼ばれ、この BDD-diag 前処理付きの CG 反復は、BDD 反復とも呼ばれる。

部分領域間境界での仮の強制変位から部分領域間境界での反力を計算する際の主要処理は、部分領域毎の有限要素法（Finite Element Method, FEM）計算（DomainFEM）である。DomainFEM では、部分領域毎に、一辺の長さがインターフェース自由度の対称密行列（ローカル Schur 補元行列）と、長さがインターフェース自由度のベクトル（インターフェース自由度ベクトル）との積（MatVec_product）が行われる。各部分領域のインターフェース自由度とは、各部分領域における、隣接部分領域との境界面上の節点の自由度の総数のことである。節点当たりの自由度は変位のみの 3 である。ここで、ローカル Schur 補元行列とインターフェース自由度ベクトルの両方とも、倍精度型で保持されており、この二つを掛け合わせる MatVec_product に関する演算も倍精度で行われる。DomainFEM におけるプロセス並列化は、各プロセスに同数の部分領域を割り当てることにより行われるものであり、MatVec_product のソースプログラム自体はプロセス並列化されていない。つまり、各プロセスにおいては、そのプロセスに割り当てられた部分領域の個数回 MatVec_product が実行される。

DomainFEM の結果から、部分領域間境界上の反力の釣り合いを評価するために、隣接する部分領域との間で境界面上の節点データ（倍精度型）を交換するための隣接部分領域間境界面通信（Exchange：mpi_isend& mpi_irecv）と、2 つの作業変数（倍精度型）に関する総和計算用縮約通信（Allreduce）が行われる。さらに、反力の合計が十分にゼロに近づいたか否かの判定を行うためと、反力の合計から部分領域間境界面上の仮の強制変位を更新するためのそれぞれにおいて、1 つの作業変数（倍精度型）に関する総和計算用縮約通信（Allreduce）が行われる。

BDD-diag 前処理の後半部であるコースグリッド修正では、残差計算とコースソルバー求解が行われる。コースグリッド修正における残差計算の主要処理は、DomainFEM と同様に、部分領域毎に、一辺の長さがインターフェース自由度の倍精度型対称密行列と、長さがインターフェース自由度の倍精度型ベクトルとの倍精度型積（MatVec_product）を行うことと、隣接する部分領域との間で境界面上の節点データ（倍精度型）を交換するための隣接部分領域間境界面通信（Exchange）を行うことである。

コースグリッド修正におけるコースソルバー求解では、一辺の長さがコース問題の自由度（= 解析対象全体の部分領域の個数 ×6（=変位 3+回転 3））の疎行列（コース行列）と、その逆行列を、BDD 反復に入る前に予め生成しておき、BDD 反復中において、その逆行列を、CG 法の反復ベクトルをコースグリッドにマップ（疎視化）させたベクトル（コース自由度ベクトル）に掛け

る方法（逆行列アプローチ）が採用されている。そのため、BDD 反復中のコースグリッド修正におけるコースソルバー求解の主要処理は、逆行列とコース自由度ベクトルの、密行列ベクトル積（CoarseMatVec）である。ここで、コース自由度ベクトルは倍精度型であるが、逆行列は倍々精度形式で保持されており、この二つを掛け合わせる演算である CoarseMatVec も倍々精度形式で行われる。CoarseMatVec におけるプロセス並列化は、各プロセスに逆行列の行を同数割り当てることにより行われる。つまり、各プロセスにおいて、長さがコース問題の自由度の倍々精度形式ベクトルと、同じ長さの倍精度型ベクトルの、倍々精度形式内積が、そのプロセスに割り当てられた逆行列の行数回実行される。この際、全てのプロセスがコース自由度ベクトル（倍精度型）の全体を保持している必要があるため、コース自由度ベクトル集約通信（Allgather）が、CoaseMatVec の直前に行われる。また、CoarseMatVec の結果得られる倍々精度型ベクトルは、CoarseMatVec の後、直ちに倍精度型に変換されるため、ADVENTURE で倍々精度形式の変数及び演算が取り扱われるのは、CoarseMatVec に関してのみである。

コースグリッド修正の後には、コースグリッド修正の結果（倍精度型）を、部分領域間境界上の節点の変異（倍精度型）に反映させるために、隣接部分領域間境界面通信（Exchange）が行われる。

4.7.1.2　コデザインの方針

ADVENTURE の実行時間の見積もりに際しては、ADVENTURE の主要処理は、時間ステップ反復内かつ BDD 反復内の処理であるとし、それに掛かる時間を、ADVENTURE の実行時間とする。BDD 反復内を、以下の通り、8 つの演算区間と 3 つの通信区間に分け、それぞれの区間の実行時間を見積もり、それらを合計することで、ADVENTURE の実行時間を見積もる。

- DomainFEM：倍精度型演算 - 対称密行列ベクトル積（2 回/BDD 反復）
- CoarseMatVec：倍々精度型演算 - 倍々精度形式の密行列ベクトル積（1 回/BDD 反復）
- CollectRhs.P3：倍精度型演算 - コース自由度ベクトルの並べ替え（1 回/BDD 反復）
- QuadMatVec.P1：倍精度型演算 - コース自由度ベクトルの倍々精度形式化（1 回/BDD 反復）
- Exec.BalancingL：倍精度型演算 - 反復ベクトルの疎視化変換と逆変換（1 回/BDD 反復）
- Exch.Buffering:倍精度型演算 - Exchange 通信用バッファリングと重ね合わせ演算（3 回/BDD 反復）
- DomainFem.P1：倍精度型演算 - 反復ベクトルの並べ替え（2 回/BDD 反復）
- Calc._Others：倍精度型演算 - 反復ベクトルの内積演算（4 回/BDD 反復）
- Exchange：倍精度型通信 - 隣接部分領域間境界面通信（mpi_isend&mpi_recv）（3 回/BDD 反復）
- Allreduce：倍精度型通信 - 総和計算用縮約通信（4 回/BDD 反復）
- Allgather：倍精度型通信 - コース自由度ベクトル集約通信（1 回/BDD 反復）

ADVENTURE の BDD 反復内において、負荷の重い処理は、密行列ベクトル積（DomainFEM、CoarseMatVec）と、メッセージ長の短い集約通信（Allgather）であり、その観点からコデザインに寄与する。

4.7.1.3　富岳性能推定用想定問題

ひとつの部分領域は立方体であり、$256 \times 256 = 65536$ 個の部分問題が平面的に並んでいる薄板状の領域を解析対象の系とし、この非線形応答問題を、富岳における ADVENTURE の実行時間を見積もる際の想定問題とする。想定問題 1 ケース当たり 1 万回の時間ステップ反復が行われ、1 時間ステップ反復当たり 500 回の BDD 反復が行われるものとする。つまり、想定問題 1 ケース当たり 500 万回の BDD 反復が行われるものとする。

この想定問題を 100 ケース処理するのに要する時間を ADVENTURE の実行時間として、富岳における ADVENTURE の実行時間を見積もる。

富岳においては、想定問題 1 ケースは、4096 ノード上の 16384 プロセスにより、処理されるものする。そのため、1 プロセス当たり 4 個の部分領域を担当することとなる。

ひとつの部分領域の一辺は、三次元ソリッド四面体二次要素（要素辺毎に中間節点があり合計 10 節点）により 10 要素分割されており、ひとつの部分領域の一辺当たりの節点数は、$(10 \times 2 + 1) = 21$ と計算される。ここで、複数の部分領域が平面的に並んでいる系においては、その系の平面に沿う方向の一辺当たりの節点数を、その辺の方向に並んだ部分領域の数 $\times 21$ とは計算できないことに注意する必要がある。部分領域間境界上の節点のダブルカウントを避ける必要がある。2 つの部分領域が横並びに並んでいる場合には、その横方向の一辺当たりの節点数は、$(2 \times 10 \times 2 + 1) = 41$ と計算される。このことを踏まえると、解析対象の系の総節点数は、$(256 \times 10 \times 2 + 1)^2 \times (1 \times 10 \times 2 + 1) = 550717461$ と計算される。節点あたりの自由度数は変位のみの 3 であり、解析対象の系の総自由度数は、総節点数の 3 倍の、約 16.5 億となる。

各部分領域のインターフェース自由度とは、各部分領域における隣接部分領域との境界面上の節点の自由度の総数のことであり、今解析対象としている系においては、$(21 \times 21 - 19 \times 19) \times 21 \times 3 = 5040$ と計算される。ここで、全ての部分領域は平面的に並んでおり、上下に隣接する部分領域は無く、各部分領域における隣接部分領域との境界面の数は 4 であることが前提となっていることを留意する必要がある。節点あたりの自由度は変位のみの 3 である。

コース問題の自由度は、部分領域当たり 1 節点で各部分領域を代表させていることと、この場合の 1 節点当たりの自由度は変位 3+ 回転 3 の合計 6 であることより、部分領域の数の 6 倍であり、$65536 \times 6 = 393216$ と計算される。ここに記述した内容を図 4.36 に示す。

4.7.2　DomainFEM 区間の主要ルーチン MatVec_Product の高速化

演算区間 DomainFEM を切り出した 1 プロセスのカーネルプログラムでは、一辺の長さが 5040 の倍精度対称密行列と長さが 5040 の倍精度ベクトルの倍精度の積（MatVec_product）が、複数回行われる。

このカーネルプログラムでは、対称行列の下三角部のみを保持し、上三角部として使う際は、転置して使用する最適化が採用されている。データ構造は、下三角行列を 6 行 6 列のブロックに分割した、ブロック単位の行優先形式となっている。各 6 行 6 列のブロック内のデータ構造も、行優先形式である。ループ構成は、外側からブロック単位の行のループ、次にブロック単位の列のループとなっている。そのため、下三角行列の先頭から順に、ブロック単位の行のループの回転が後になるに従って、1 行あたりの計算負荷量が大きくなる。この 2 重ループの内側では、6 行 6 列のブロックについての処理が行われることとなる。

このカーネルプログラムは、6 行 6 列のブロック単位の処理を繰り返すものであり、オリジナルのソースプログラムでは、6 行 6 列のブロックについての処理がフルアンロールされていたため、

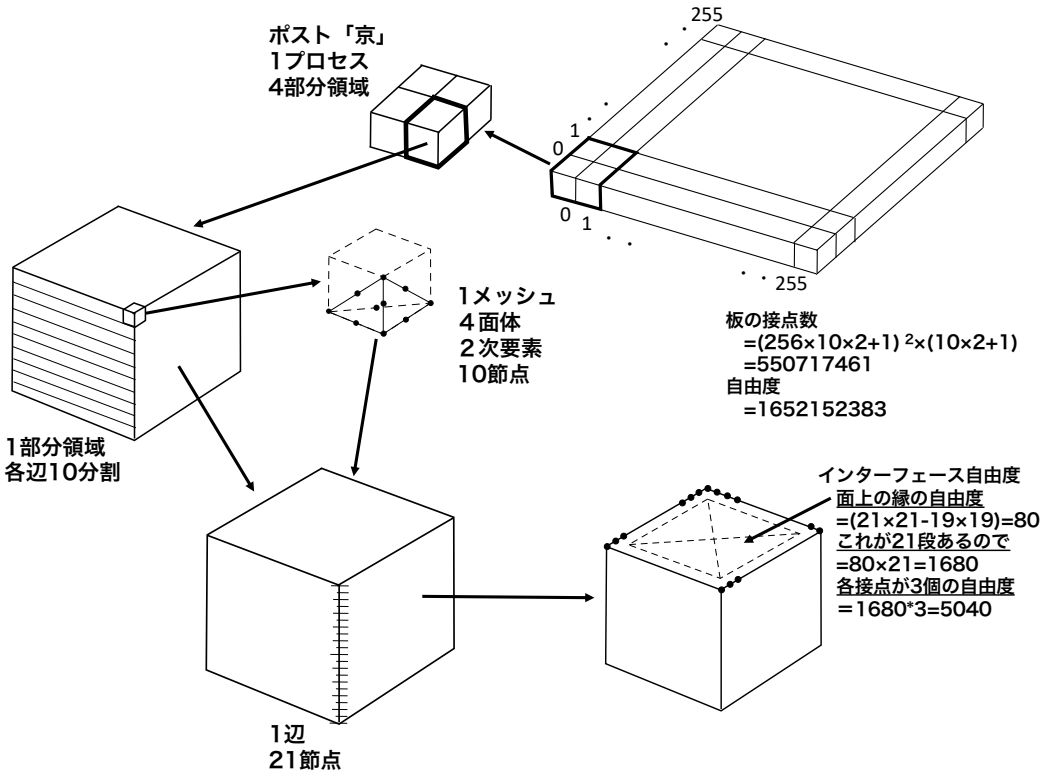


図 4.36: 富岳性能推定用想定問題

ブロック単位の列のループが、コンパイラによる SIMD 化軸となっていた。つまり、SIMD 化軸が、6 行 6 列のブロックをまたぐ方向となっていた。このため、フルアンロールと相まって、データアクセスの連続性が途切れがちである。そのため、ハードウェアプリフェッチが失敗しやすく、そのことが、オリジナルのソースプログラムでは、性能劣化要因となっていた。そこで、ビルトインプリフェッチ関数によるソフトウェアプリフェッチ命令の発行により、性能を改善した。

オリジナルのソースプログラムでは、スレッド並列化が、ブロック単位の行のループにおいて、チャンクサイズ 1 の動的スケジューリングにて行われており、チャンクサイズが 1 と小さいことによるメモリアクセスの連続性の低さと、動的スケジューリングに係るオーバーヘッドの大きさが懸念されていた。そこで、ブロック単位の行のループのスケジュール方式が、チャンクサイズ 12 の降順折り返しサイクリック方式の静的スケジューリングとなるように、ソースプログラムの改変を行い、性能を改善した。

オリジナルのソースプログラムでは、6 行 6 列のブロックについての処理がフルアンロールされていたため、ブロック単位の列のループが、コンパイラによる SIMD 化軸となっていた。つまり、SIMD 化軸が、6 行 6 列のブロックをまたぐ方向となっていた。このとき、ストライドが $6 \times 6 = 36$ のアクセスのため、間接 SIMD アクセスとなる。この場合、SIMD 長の範囲で複数のキャッシュラインのデータが必要となり、L1D キャッシュのビジー率が高くなっていた。また、間接 SIMD アクセスのためのアドレス計算命令が多く出ており、命令数増加の要因となっていた。そこで、6 行 6 列のブロックについての処理の列ループ（最内ループ）をリロールし、それを SIMD 化軸とすることにより、SIMD アクセスを連続化し、L1D キャッシュのビジー率を低下させた。このことによ

り、フルアンロールされていた演算の大部分が SIMD 命令の中に閉じ込められるため、必要レジスタ数を大幅に減らす効果も期待できる。ただし、ここでの列のループの長さは 6 と、FX100 の SIMD 長の 4 でも、富岳で想定されている SIMD 長の 8 でも割り切れない。そのため、最適化指示行を指定することにより、マスク付き SIMD 命令を利用することで、端数部分を含めて SIMD 処理させた。

6 行 6 列のブロックについての処理の列ループ（最内ループ）をリロールし、それを SIMD 化軸とすると、長さ 6 のベクトルの内積計算を SIMD 処理することとなり、SIMD 要素間縮約の SIMD 処理（SIMD 内縮約）が発生することとなる。この SIMD 内縮約は、通常処理と比べてより多くの命令を必要とする。そこで、内積計算中の掛け算の結果を作業配列に足し込むように、ソースプログラムを変更することによって、縮約を SIMD 要素間ではなく SIMD 命令間で行うようにし、命令数の増加を抑制した。

最内ループの回転数が SIMD 長で割り切れないにも関わらず、最適化指示行を指定することにより、マスク付き SIMD 命令を利用することで、端数部分を含めて SIMD 処理するようにしたところ、このマスクされている部分は、実メモリアクセスの際に、作業配列宣言の 2 次元目が 1 だけ後の部分とオーバーラップしており、ロードとマスクされたオーバーラップ域を含むストアとの間で、ロードがストアを追い越さず、逐次化動作となる（Store-Fetch-Interlock、SFI）。プログラムの論理的な動きとしては追い越しても問題は無いが、ハードウェアの動きとして、安全を見た逐次化動作となっており、一般的に用いられている方法でもある。その結果、パイプライン動作が崩れ、性能が低下する。そこで、作業配列の 1 次元目に SIMD 幅を考慮したマージンを追加して、SIMD ロード・ストア間のオーバーラップ検出を回避することで性能劣化を回避する。

上述のソースプログラムの改良により、DomainFEM を切り出した 1 プロセスのカーネルプログラムを、FX100 の 12 コアを使用して実行した際の、MatVec_product1 回当たりの実行時間は、1.902×10^{-3} 秒から 1.091×10^{-3} 秒に短縮された。

現状のコンパイラのオブジェクトでは最内ループ内で毎回ロードストアが行われるが、この領域は SIMD 幅未満であり、外側ループの回転に対して不変である。そのため、レジスタ上に維持したままとすることで処理が可能であり、ロードストア命令を削減し、L1 キャッシュの負荷を軽減することができる。富岳性能シミュレーションでは、コンパイラによるこのような最適化を想定して、組込み関数（ACLE）を用いて記述したコードを暫定的に用いた。

4.7.3 コースグリッド修正区間の主要ルーチン CoarseMatVec の高速化

演算区間 CoarseMatVec を切り出した 1 プロセスのカーネルプログラムでは、24 行 393216 列の倍々精度形式行列と、長さ 393216 の倍精度型ベクトルの、倍々精度形式の積が、繰り返し多数回行われる。ここで、1 つの倍々精度形式数は、2 つの倍精度型数で表現されており、この 2 つの倍精度型数のうちの 1 つは倍精度型数そのものであり、もう 1 つは精度拡張用の値を保持している。

オリジナルのソースプログラムでは、倍々精度形式の行列と倍々精度形式のベクトルの倍々精度形式積を計算する関数を使用していた。行列のデータ構造は、行優先形式で、ループ構成は、外側から行列の行のループ、次に行列の列のループとなっていた。この 2 重ループの内側では、2 つの倍々精度形式数の倍々精度形式乗算と、その結果の総和計算用作業変数への倍々精度形式足し込みが、逐次的に行われていたため、最内ループである行列の列のループが SIMD 化されていなかった。行列の列のループをスレッド数に等分割し、それぞれを各スレッドに割り当てるスレッド並列化が行われていたために、各スレッドにおける処理が終わった後に、スレッド毎の総和計算用作業変数の、倍々精度形式総和を計算する必要があった。さらに、行列の列の長さが $393216 = 2^{17} \times 3$

と冪指数が大きな 2 の冪乗の倍数である状況下で、行列の行のループが 3 段にアンロールされていたため、L1D キャッシュにおいてキャッシュライン競合が発生しやすい状況となっていた。さらに、本来倍精度型であるベクトルを倍々精度形式化した上で行列との積を計算していたために、データストリーム数が 1 つ増え、キャッシュライン競合がより発生しやすい状況となっていた。

そこで、プログラム構造を見直すことにより、本演算カーネルプログラムの性能を改善した。まず、最外の行列の行のループのアンロールを止めることにより、L1D キャッシュにおけるキャッシュライン競合が発生しやすい状況を緩和した。行列の行のループをスレッド並列化ループとすることにより、スレッド毎の総和計算用作業変数の倍々精度形式総和を、後で計算する必要をなくした。行列の行のループの内側において、倍々精度形式ベクトルと倍精度型ベクトルの倍々精度形式内積を行うサブルーチンを呼び出すこととすることにより、本来必要のなかった倍精度ベクトルの倍々精度形式拡張を止め、データストリーム数を 1 つ削減し、キャッシュライン競合が起こりやすい状況を緩和した。倍々精度形式ベクトルと倍精度ベクトルの倍々精度形式内積に係る処理を、SIMD 演算器を活用して行うために、そのサブルーチンの内部を、乗算部と総和部に二分した。

乗算部では、倍々精度形式ベクトルと倍精度型ベクトルのそれぞれの要素番号を同一とする要素同士の倍々精度形式乗算のみが行われ、その結果は倍々精度形式の総和計算用作業配列に格納される。この乗算部における処理は、SIMD 処理が可能である。

総和部では、この倍々精度形式の総和計算用作業配列の全ての要素の倍々精度形式総和が、前半後半二分割ベクトル和繰り返し方式により計算される。ここで、前半後半二分割ベクトル和繰り返し方式とは、SIMD 演算機を活かしたベクトルの要素の総和を計算する方法のうちの一つである。一つの要素数 2N のベクトルの総和を計算する際に、前半の要素番号 1 から N までと、後半の要素番号 N+1 から 2N までの、それぞれを独立した一つのベクトルとして見なし、この二つのベクトルの和を計算する。この二つのベクトルの和の計算は、SIMD 処理可能である。そして結果として、要素数 N のベクトルが一つ得られる。続けて、この一つの要素数が N のベクトルの、前半の要素番号 1 から N/2 までと、後半の要素番号 N/2+1 から N までの、それぞれを独立した一つのベクトルとして見なし、この二つのベクトルの和を計算する。この二つのベクトルの和の計算も、SIMD 処理可能である。そして結果として、要素数が N/2 のベクトルが一つ得られる。以上の処理を、ベクトルの要素数が 1 になるまで繰り返す。これが、前半後半二分割ベクトル和繰り返し方式と呼ばれる、SIMD 演算機を活かしたベクトルの要素の総和を計算する方法である。ただし、Fortran 2008 規格で新たに定義された「do concurrent」文を使用せず、「do」文等の通常のループ文を使用してプログラムを書くとすると、総和部を SIMD 処理可能とするためには、言語仕様に従ってソースプログラムの記述をする限りは、倍々精度形式の総和計算用作業配列を 2 つ用意することが必要となり、ロードに関してデータ量とデータストリーム数の両方を増大させることとなるという弱点もあった。富岳では Fortran 2008 規格に準拠するため、「do concurrent」文を採用することにより、言語仕様に従ってソースプログラムを記述をする場合でも、倍々精度形式の総和計算用配列を 2 つ用意する必要をなくした。そして、倍々精度形式の総和計算用配列を 1 つのみ使用するソースプログラムに書き換えることで、ロードされるデータ量とデータストリーム数の両方を削減し、実行時間を削減した。さらに、総和計算用の倍々精度形式の作業配列をキャッシュ内に収めるために、乗算部と総和部の両方を含む内積計算部全体に、ストリップマイニング最適化を適用した。さらに、CoarseMatVec を切り出した 1 プロセスのカーネルプログラムを、FX100 の 12 コアを使用して実行する場合に、その実行時間が最小となるストリップ長の探索調査も行った。

上述のソースプログラムの改良により、CoarseMatVec を切り出した 1 プロセスのカーネルプログラムを、FX100 の 12 コアを使用して実行した際の、CoarseMatVec1 回当たりの実行時間は、

22.31×10^{-3} 秒から 2.41×10^{-3} 秒に短縮された。詳細設計 (3) における、ポスト京性能推定には、この改良版のソースプログラムを使用している。

詳細設計 (3) では、CoarseMatVec のソースプログラムのさらなる改良も行った。

具体的には、上記では、倍々精度形式ベクトルと倍精度ベクトルの倍々精度形式内積に係る処理を、乗算部と総和部に二分していたが、それを止め、次のような処理を行うこととした。先ず、内積を計算しようとする倍々精度形式ベクトルと倍精度型ベクトルのそれぞれの最初の B 個の要素について、要素番号を同一とする要素同士の倍々精度形式積を計算し、その結果を、長さ B の総和計算用倍々精度形式作業配列に代入する。次に、今内積を計算しようとしている二つのベクトルのそれぞれの次の B 個の要素について、要素番号を同一とする要素同士の倍々精度形式積を計算し、その結果を先程の長さ B の総和計算用倍々精度形式作業配列に対して倍々精度形式足し込みを行う。その後も、同様に、内積を取ろうとしている二つのベクトルについて、要素番号を同一とするベクトル要素同士の積の結果を総和計算用作業配列に足し込むということを、要素数 B 毎に、内積を計算しようとしている二つのベクトルの終端に到るまで、繰り返す。最後に、長さが B の総和計算用倍々精度形式作業配列の全要素の倍々精度形式総和を、前述の前半後半 2 分割ベクトル和繰り返し方式により求めることで、倍々精度倍精度形式ベクトルと倍精度ベクトルの倍々精度形式内積を計算することとした。詳細設計 (3) では、B の大きさは、暫定的に 896 としている。

詳細設計 (3) による、CoarseMatVec のソースプログラムの改良の結果、CoarseMatVec を切り出した 1 プロセスのカーネルプログラムを、FX100 の 12 コアを使用して実行した際の、CoarseMatVec1 回あたりの実行時間は、1.95×10^{-3} 秒まで短縮された。

4.7.4 コデザインによるシステムへの実装要請項目

短メッセージ・ノード内複数プロセス時の MPI_Allgather の高速化

ポスト京性能推定では、コース自由度ベクトル集約通信（Allgather）に要する時間が、主要演算区間の一つである CoarseMatVec の実行時間に匹敵することとなっており、1BDD 反復当たりの実行時間の 15%近くを占めることとなっている。そのため、コース自由度ベクトル集約通信（Allgather）にかかる時間の短縮も求められる。コース自由度ベクトル集約通信（Allgather）では、1 プロセス当たりの送出バイト数が 192 バイトと小さい 16364 プロセスの集約通信（mpi_allgather）が、1BDD 反復当たり 1 回行われる。さらに、富岳では、1 ノード内に 4 つの CMG が搭載されており、ADVENTURE の実行時間の見積に際しては、1 プロセス当たり 1CMG の実行を想定されているため、1 ノード内に 4 つのプロセスが存在することとなる。そのため、短メッセージ・ノード内複数プロセス時の mpi_allgather 通信に要する時間の短縮が、システム側への実装要請として挙げられる。

表 4.59: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (ADVENTURE)

実装要請課題	対応
ノード内に複数プロセスが存在する場合の短メッセージ MPI_Allgather の高速化	改良版 Gtbc アルゴリズムにより性能改善
逐次版 BLAS とスレッド並列版 BLAS の併用	スレッド並列実行時における逐次版 BLAS 呼び出し機能を開発する

表 4.60: 京の 32768 ノード（32768 プロセス）での実行時間

処理区間名	1BDD 反復当たりの実行時間
DomainFEM	8.36972 ミリ秒
CoarseMatVec	17.84025 ミリ秒
CollectRhs.P3	0.26833 ミリ秒
QuadMatVec.P1	1.04247 ミリ秒
Exec.BlancingL	0.76123 ミリ秒
Exch.Buffering	0.52835 ミリ秒
DomainFem.P1	0.31601 ミリ秒
Calc._Others	0.24571 ミリ秒
Exchange	0.27125 ミリ秒
Allreduce	0.20164 ミリ秒
Allgather	3.84919 ミリ秒
合計	33.69414 ミリ秒

4.7.5　富岳向け性能推定

性能電力予測ツールおよび性能シミュレータ等の環境を用いた富岳向け性能推定は第 4.1.5 節に説明される手法により実施した。

4.7.5.1　京ベースライン測定

BDD 反復内の 8 つの演算区間のそれぞれと 3 つの通信区間のそれぞれの、想定問題 1 ケースを京の 32768 ノード上の 32768 プロセスにより処理した場合の、京での実行時間を測定し（表 4.60）、それら 11 個の測定値の合計値を、ADVENTURE の実行時間の京での測定値（京ベースライン測定値）とし、富岳の対京性能向上比を計算する際に使用した。

この京ベースライン測定値（1BDD 反復当たり 33.69414 ミリ秒）は、ADVENTURE の開発元が、フラッグシップ 2020 プロジェクトの開始以前に、想定問題 1 ケースを京の 32768 ノード上の 32768 プロセスにより処理した際の、1BDD 反復当たり 33 ミリ秒（プレゼンテーション資料「領域分割法ソルバーの性能ベンチマーク」（河合ほか））と概ね一致している。

京ベースライン測定では、想定問題 1 ケースを、京の 32768 プロセス（32768 ノード）により処理することを想定しているため、1 プロセス当たり 2 個の部分領域を担当することとなっている。一方、富岳性能推定では、想定問題 1 ケースを、富岳の 16384 プロセス（4096 ノード）により処理することを想定しているため、1 プロセス当たり 4 個の部分領域を担当することとなる。そのため、京ベースライン測定における 1 プロセスあたりの演算量は、富岳性能推定における 1 プロセスあたりの演算量の半分となっている。

4.7.5.2　富岳性能推定

BDD 反復内の 8 つの演算区間のそれぞれと 3 つの通信区間のそれぞれの、想定問題 1 ケースを富岳の 4096 ノード上の 16384 プロセスにより処理した場合の、富岳での実行時間を推定し（表

表 4.61: 富岳の 4096 ノード（16384 プロセス）での実行時間（通常モード、エコモード無効）

処理区間名	1BDD 反復当たりの実行時間	推定手法
DomainFEM	5.04982 ミリ秒	性能電力予測ツール
CoarseMatVec	1.70331 ミリ秒	性能電力予測ツール
CollectRhs.P3	0.08187 ミリ秒	性能電力予測ツール
QuadMatVec.P1	0.26126 ミリ秒	性能電力予測ツール
Exec.BalancingL	1.67776 ミリ秒	FX100 時間 CPU 周波数換算
Exch.Buffering	1.02033 ミリ秒	FX100 時間 CPU 周波数換算
DomainFem.P1	0.64285 ミリ秒	FX100 時間 CPU 周波数換算
Calc._Others	0.32115 ミリ秒	FX100 時間 CPU 周波数換算
Exchange	0.21520 ミリ秒	机上見積式
Allreduce	0.04440 ミリ秒	机上見積式
Allgather	1.64762 ミリ秒	京時間通信バンド幅換算
合計	12.66557 ミリ秒	

4.61)、それら 11 個の測定値の合計値を、富岳での ADVENTURE の実行時間の推定値とし、富岳の対京性能向上比を計算する際に使用した。

演算区間の富岳性能推定に際しては、DomainFEM と CoarseMatVec については、4.7.2 節と 4.7.3 節に記述した高速化チューニングを行ったソースプログラムを用いたが、その他の演算区間については、ADVENTURE の開発元から提供されたオリジナルのソースプログラムを用いた。

演算区間のうち、スレッド並列化されている区間である、DomainFEM、CoarseMatVec、CollectRhs.P3、QuadMatVec.P1 のそれぞれについては、各演算区間の FX100（12 スレッド/CMG の制限付）上での詳細プロファイラ情報を基に、性能電力予測ツールを使用することにより、富岳での実行時間と消費電力を推定した。演算区間でも、スレッド並列化されていない区間である、Exec.BlancingL、Exch.Buffering、DomainFem.P1、Calc._Others のそれぞれについては、各演算区間の FX100（12 スレッド/CMG の制限付）上での実行時間を、CPU 周波数の比を用いて換算することにより、富岳での実行時間を推定した。性能電力予測ツールによる電力予測を行わない演算区間の富岳での消費電力は、性能電力予測ツールにより推定した演算区間の消費電力の時間重み付き平均とした。また、演算区間 DomainFEM と演算区間 CoarseMatVec については、ポスト京 CPU 性能シミュレータによる性能推定も行った。

通信区間 Exchange は、演算区間 DomainFEM の結果から部分領域間境界上の反力の釣り合いを評価するために、隣接部分領域との間で境界面上の節点データを交換するためのものである。複数の部分領域が平面的に並んでいる系において、各部分領域が節点データ（倍精度型）を交換する相手となる部分領域の数は、面的に接する（面を共有する）前後左右方向の 4 個と、線的に接する（辺を共有する）斜め方向の 4 個の、合計 8 個である。想定問題 1 ケースを、富岳の 16384 プロセス（4096 ノード）で実行することにしたことにより、演算区間 DomainFEM では、1 プロセス当たり 4 個の部分領域を担当することとなる。そして、この 4 個の部分領域は前後左右に面的に隣接した 2×2 の配置で平面的並んでおり、ひとつの部分領域塊を形成している。この 4 個の部分領域間にプロセス間通信は発生しない。また、富岳では 1 計算ノード当たりに 4 プロセスが割り当てられ、各プロセスが担当する部分領域塊は、前後方向に互いに面的に接して 1 列に並んでいるものとする。これらの条件の下で、通信区間 Exchange における、プロセス間の通信量

を見積もった。4 個の部分領域が 2×2 の配置で平面的に並んでいる部分領域塊の、平面に沿う方向の一辺当たりの節点数は、$(2 \times 10 \times 2 + 1) = 41$ と計算される。一方、その平面に垂直な上下方向の一辺当たりの節点数は、ひとつの部分領域の一辺当たりの節点数そのものの 21 である。節点当たりの自由度数は 3 であり、節点データは倍精度型（8 バイト）で保存されている。これらより、通信区間 Exchange は 1 回の実行につき、各プロセスは、前後左右方向に面的に接する 4 つのプロセスとの間で 20664 バイト（= 41 節点 × 21 節点 × 3 自由度 × 8 バイト）のデータ交換（mpi_isend&mpi_irecv）を行い、さらに、斜め 4 方向に線的に接する 4 つのプロセスとの間で 504 バイト（= 21 節点 × 3 自由度 × 8 バイト）のデータ交換を行うこととなる。ここで、各計算ノードは、隣接する方向が 180 度異なる 2 つの計算ノードとのみ、同時にデータ交換を行うことができ、さらにその際、1 方向につき 1 組のプロセス間通信のみが一度に実行可能であるものともする。以上より、富岳で通信区間 Exchange を 1 回実行するのに掛る時間の見積もり式は、以下のようになり、71.734×10^{-6} 秒と見積もられる。この際、富岳でのノード間通信スループットの実効値としては、富岳のノード間通信スループットの理論値 6.8×10^9 バイト/秒の 80% 値を使用した。通信レイテンシとしては、4.0×10^{-6} 秒を使用した。

$$\frac{20664 \times 1 + 20664 \times 4 + 504 \times 8}{6.8 \times 0.8 \times 10^9} + (1 + 4 + 8) \times 4.0 \times 10^{-6} \approx 71.734 \times 10^{-6} \tag{4.9}$$

この通信区間 Exchange は、1BDD 反復当たり 3 回実行される。つまり。通信区間 Exchange に掛る 1BDD 反復当たりの富岳での実行時間の見積もり値は、0.21520×10^{-3} 秒となる。

通信区間 Allreduce では、想定問題 1 ケースを、富岳の 16384 プロセス（4096 ノード）で実行することにしたことにより、1 プロセス当たりの送出バイト数が 16 バイト（= 2 要素 × 8 バイト）の 16384 プロセスの縮約通信（mpi_allreduce）が、1BDD 反復当たり 1 回と、1 プロセス当たりの送出バイト数が 8 バイト（= 1 要素 × 8 バイト）の 16384 プロセスの縮約通信（mpi_allreduce）が、1BDD 反復当たり 2 回行われることとなるが、富岳での実行時間の見積もりに際しては、簡単化のために、1 プロセス当たりの送出バイト数が 8 バイト（= 1 要素 × 8 バイト）の 16384 プロセス（4096 ノード、4 プロセス/1 ノード）の縮約通信（mpi_allreduce）が、1BDD 反復当たり 4 回行われることとなるものとした。16384 プロセス（4096 ノード、4 プロセス/1 ノード）の 1 要素（8 バイト）の縮約通信（mpi_allreduce）に、富岳にて要する時間は、富士通により 11.1×10^{-6} 秒と見積もられているため、それを単純に 4 倍することにより、通信区間 Allreduce の実行に掛る 1BDD 反復当たりの富岳での実行時間は、0.04440×10^{-3} 秒と見積もられた。

通信区間 Allgather は、各プロセスがそれぞれ互いに重なりなく、コース自由度ベクトルを保持している状態から、全プロセスがそれぞれ、コース自由度ベクトルの全体を保持するようにするために行われるものである。コース問題では、各部分領域を 1 部分領域当たり 1 節点で代表させており、この場合の 1 節点当たりの自由度は変位 3+ 回転 3 の合計 6 である。想定問題 1 ケースを、富岳の 16384 プロセス（4096 ノード、4 プロセス/1 ノード）で実行することにしたことにより、1 プロセス当たりの送出バイト数が 192 バイト（= 4 部分領域 × 6 自由度 × 8 バイト）の 16384 プロセス（4096 ノード、4 プロセス/1 ノード）の集約通信（mpi_allgather）が、1BDD 反復当たり 1 回行われることとなった。通信区間 Allgather の富岳での実行時間は、同種（1 プロセス当たりの送出バイト数が 192 バイトの 16384 プロセス（4096 ノード、4 プロセス/1 ノード））の集約通信（mpi_allgather）の京での実行時間を、通信スループットの理論値（京では 5.0×10^9 バイト/秒、富岳では 6.8×10^9 バイト/秒）の対京向上比（1.36）で割ることにより求めることとした。1 プロセス当たりの送出バイト数が 192 バイトの 16384 プロセス（4096 ノード、4 プロセス/1 ノード）の集約通信（mpi_allgather）の京での実行時間の測定値は、2.240763×10^{-3} 秒だった。その 2.240763×10^{-3} 秒を 1.36 分の 1 することにより、通信区間 Allgather に掛る 1BDD 反復当たりの

富岳での実行時間を、1.64762×10^{-3} 秒と見積もった。

この時点の性能推定結果が、ノーマル周波数・エコモード無効の条件において「京」の性能に対し、41 倍であった。

4.7.6 富岳での性能測定

4.7.6.1 MulMatVec のチューニング

この処理は対称行列とベクトルとの掛け算であり、CG 法で必要となる行列ベクトル積計算である。2020 年 3 月提供の新コードは下三角行列を中心の行番号で一回のみ折り返したときの行ブロック対をスレッドに割り当て、スレッド当たり処理量の均等化を行っている。また対称行列の格納法は、対角要素を除く下三角行列部をスカイライン格納し、対角要素を別配列に格納している。これら 2 点がこれまでに使用したコードと異なる。

新コードのスレッド割り当て方法ではループ長の分布が不均等であり、メモリアクセスコストのインバランスが発生する。そこでブロック単位で折り返しサイクリック割り当てを行うことによって、メモリアクセスコストの均等化を図る。その際に下三角行列の最後尾行からブロック単位の割り当てを行う。これによって次第に各ブロックの列数が減り最後は列数が 1 近辺となるため、折り返しサイクリック割り当ての周期の余りによるインバランスの影響度が低下し無視できるほど小さくなる。また、新コードでは最内ループの行アンロール数が 3 である。行アンロールによってアンロール数に比例した回数だけ、レベル 1 キャッシュ上ないしはレジスタ上にベクトルデータがあるうちにアクセスすることが可能となり、メモリアクセスないしはレベル 2 キャッシュアクセスのコストを削減することができる。この効果がアンロール数が 3 では不十分な可能性があるためアンロール数を 6 にする。

上記アイデアで再構築したコードの抜粋を図 4.37 に示す。Adventure ではデータが構造体に格納されており、演算部で直接構造体を参照すると、副作用により多くの命令が生成されるため、一旦、構造体メンバを指すポインタに置き換えて参照することで副作用を回避している。

次にチューニングコードを 100 回実行したときの、CPU 解析レポートの結果を図 4.38 に示す。アクセス待ち時間や命令コミットサイクルがスレッド間でほぼ均等になっており狙い通りとなっている。演算待ち時間がやや目立つが、コンパイルリストによればソフトウェアパイプライン化されておりスピルは発生していない。またメモリスループットは上限に近い。

```
void AutoLinAS_MatVec_Sym_Omp4_product (AutoLinAS_MatVec_Sym_Omp4 * restrict self,
 double * restrict vector, double * restrict result_OUT)
{   ……
#pragma omp parallel private(idof,jdof,threadid,nthreads), shared(vector,self,Chunk,nDofs)
{
    threadid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    int ncycle = (int)( (int)( (nDofs - 1)/Chunk )/nthreads ) + 1;    /*Chunk 単位行ブロックの繰り返し数 */
    int iasc = threadid + 1;
    int idsc = nthreads - threadid;
    double * restrict wkv = self->thread_result[threadid];     /* 構造体メンバを指すポインタへの置き換え */
    double * restrict matsky = self->offDiagCmps;               /*          〃                    */
    double * restrict diag = self->diagCmps;                    /*          〃                    */
    for(ic = 0; ic < ncycle; ic++){
        int ithr = iasc;                                  /* ascending thread order. */
        if(ic%2 == 1){ ithr = idsc; }                     /* descending thread order. */
        int ioff = nDofs - (nthreads*ic + ithr)*Chunk;    /* row offset of chunk block. スレッドへの割当*/
        if(ioff >= 0){                                          /* 最終行からの折り返しブロックサイクリック */
          for (i = 0 ; i < Chunk; i=i+6) {                /* ブロック内  6 行単位ループ */
            int idof0 = ioff + i + 0;
            int idof1 = ioff + i + 1;
            int idof2 = ioff + i + 2;
                ……..
            int offset0 = (idof0 * (idof0 - 1)) / 2; /* offset of a block in off-diagonal skyline matrix store. */
            int offset1 = (idof1 * (idof1 - 1)) / 2;
            int offset2 = (idof2 * (idof2 - 1)) / 2;
                ……
            double vec_i0 = vector[idof0];
            double vec_i1 = vector[idof1];
            double vec_i2 = vector[idof2];
              ……                    /* vec_i5 まで */
            for(jdof=0; jdof<idof0; jdof++){          /* 下三角列番号、上三角行番号ループ */
              double vec_j = vector[jdof];
              double mat_i0j = matsky[offset0+jdof];
              double mat_i1j = matsky[offset1+jdof];
              double mat_i2j = matsky[offset2+jdof];
                ……                  /* mat_i5j まで */
              product0 += mat_i0j * vec_j;            /* 下三角部 */
              product1 += mat_i1j * vec_j;            /* 下三角部 */
              product2 += mat_i2j * vec_j;
                ……                  /* product5 まで */
              wkv[jdof] += mat_i0j * vec_i0 + mat_i1j * vec_i1 + mat_i2j * vec_i2 + mat_i3j * vec_i3
                                   + mat_i4j * vec_i4 + mat_i5j * vec_i5;          /* 上三角部 */
            }
            product0 += diag[idof0] * vector[idof0];    /* 対角部 */
            wkv[idof0] += product0;
            (ブロック内対角近傍三角行列及び対角部分の計算)
            (Chunk ブロック余りの行の計算：列数が 1 の近傍)
```

図 4.37: MulMatVec のチューニングの概要

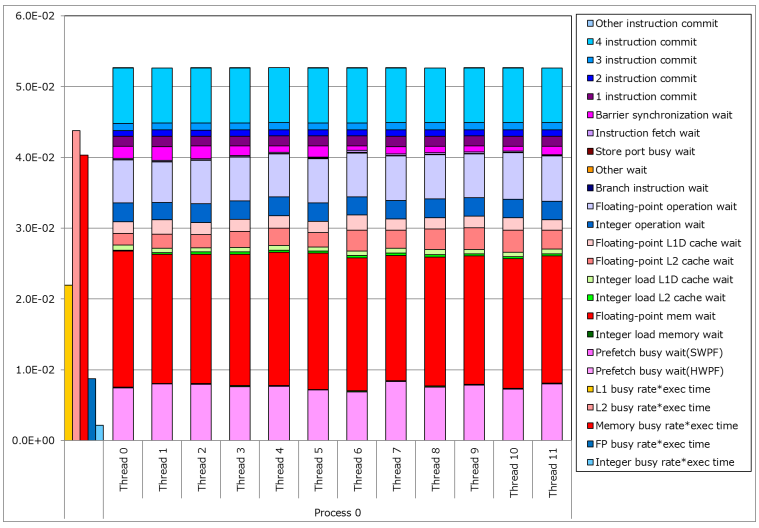


図 4.38: MulMatVec の CPU 解析レポートの結果

4.7.6.2 MatVec_Rect_Quad のチューニング

2020 年 3 月提供の新コードの 4 倍精度演算行列ベクトル積を行う Region1,2 では、内積計算の内の加算計算部分は最内ループを手動アンロールしたコードになっている。その結果、ストライド 4 の SIMD アクセスとなっており、アクセス性能がベストではない。最内ループの手動アンロールは廃止すべきである。また同コードでは、内積計算をブロック化して、乗算部から加算計算部分に乗算結果を引き渡しメモリアクセスの軽減を図っている。ブロック化はブロック内乗算ループとブロック内全要素加算の Recursive Halving ループの繰り返しとなっている。しかし Recursive Halving ループは最内ループ長が次第に短くなりパイプライン処理効率が悪くなるため、可能な限り回避するべきである。そこで、ブロック化を、ブロック内の乗算部と、ブロック内ではなくブロック間の加算を行う部分の繰り返しとし、最後にブロックサイズ分だけ残った結果を Recursive Halving によってブロック内加算するように変更する。このためには、最初のブロックは乗算のみを行い、次のブロックでのブロック間加算に備える。以上を纏めると、ブロッキング手法を改良した処理手順は以下となる。

(1) 1 番目のブロック内の乗算結果を作業配列に格納する

(2) i 番目のブロック内の乗算を行い、i-1 番目のブロックの結果と要素同士の加算を行う。これをブロック数-1 回繰り返す。

(3) 最後の結果のブロックについて Recursive Halving によるブロック内縮約加算を行う

上記アイデアで再構築したコードの抜粋を図 4.39、図 4.40 に示す。

4.7.6.3 その他のチューニング

その他のチューニングとして、後述する図 4.41 の 4、5 に対応する処理を BDD 反復前に移動した。また、11、12 の処理を omp 化し、23 に対応する処理は、削除した。

```
  void AutoLinAS_MatVec_Rect_Quad_Omp2_product (AutoLinAS_MatVec_Rect_Quad_Omp2 * __restrict__
self,
    double * __restrict__ vector_hi, double * __restrict__ vector_lo,
    double * __restrict__ result_hi_OUT, double * __restrict__ result_lo_OUT)
  {
    double * __restrict__ cmps_hi = self->cmps_hi;
    double * __restrict__ cmps_lo = self->cmps_lo;
    int nSegments = (nColumns + SEGMENT_LEN - 1) / SEGMENT_LEN;   /* ブロック数計算 */
    ........
  #pragma omp parallel for private(iSegment, jDof)
     for (iDof = 0; iDof < nRows; iDof++) {                          /* 行列行番号ループ */
        double * __restrict__ work_cmps_hi = self->thread_work_cmps_hi[threadId];
        double * __restrict__ work_cmps_lo = self->thread_work_cmps_lo[threadId];
        int length = SEGMENT_LEN;
        if (nColumns < SEGMENT_LEN) {   length = nColumns;   }
        long matOffset = (long)iDof * nColumns;
        for (jDof = 0; jDof < length; jDof++) {                      /* ブロック内行列列番号ループ*/
           AutoMT_inline_declare_dd(M_ij);
           AutoMT_inline_declare_dd(a);
           AutoMT_inline_declare_dd(b);
           AutoMT_inline_load_dd_d_d(M_ij, cmps_hi[matOffset + jDof],
cmps_lo[matOffset + jDof]);
           AutoMT_inline_load_dd_d_d(a, vector_hi[jDof], vector_lo[jDof]);
           AutoMT_inline_mul_dd_dd_dd(M_ij, a, b);
           AutoMT_inline_store_dd_d_d(b, work_cmps_hi[jDof], work_cmps_lo[jDof]);
        }   /* jDof */
        for (iSegment = 1; iSegment < nSegments; iSegment++) {       /* ブロックループ */
           long matOffset = (long)iDof * nColumns + iSegment * SEGMENT_LEN;
           int vecOffset = iSegment * SEGMENT_LEN;
           int length = SEGMENT_LEN;
           if (nColumns < (iSegment + 1) * SEGMENT_LEN) { length = nColumns - iSegment * SEGMENT_LEN; }
           for (jDof = 0; jDof < length; jDof++) {                   /* ブロック内行列列番号ループ */
              AutoMT_inline_declare_dd(M_ij);
              AutoMT_inline_declare_dd(a);
              AutoMT_inline_declare_dd(b);
              AutoMT_inline_load_dd_d_d(M_ij,   cmps_hi[matOffset   +
jDof], cmps_lo[matOffset + jDof]);
              AutoMT_inline_load_dd_d_d(a, vector_hi[vecOffset + jDof], vector_lo[vecOffset + jDof]);
              AutoMT_inline_mul_dd_dd_dd(M_ij, a, b);
              AutoMT_inline_declare_dd(c);
              AutoMT_inline_declare_dd(d);
              AutoMT_inline_load_dd_d_d(c, work_cmps_hi[jDof], work_cmps_lo[jDof]);
              AutoMT_inline_add_dd_dd_dd(b, c, d);
              AutoMT_inline_store_dd_d_d(d, work_cmps_hi[jDof], work_cmps_lo[jDof]);
           }   /* jDof */
        }   /* iSegment */
                          ( 続く )
```

ブロッキング前処理:1 ブロック目ブロック内乗算演算

2 番目以降のブロック内乗算演算とブロック間加算

図 4.39: MatVec_Rect_Quad のチューニングの概要 (1)

```
     .......
        length = SEGMENT_LEN;
        if (nColumns < SEGMENT_LEN) {   length = nColumns;      }
        for (;;) {     /* recursive halving of a resulting segment */
           half = (length - 1) / 2 + 1;
           shalf = length - half;
           for (jDof = 0; jDof < shalf; jDof++) {
              AutoMT_inline_declare_dd(a);
              AutoMT_inline_declare_dd(b);
              AutoMT_inline_declare_dd(c);
              AutoMT_inline_load_dd_d_d(a, work_cmps_hi[jDof], work_cmps_lo[jDof]);
              AutoMT_inline_load_dd_d_d(b, work_cmps_hi[jDof + half], work_cmps_lo[jDof + half]);
              AutoMT_inline_add_dd_dd_dd(a, b, c);
              AutoMT_inline_store_dd_d_d(c, work_cmps_hi[jDof], work_cmps_lo[jDof]);
           }
           length = half;
           if (length < 2) { break;   }
        }   /* recursive halving */
        result_hi_OUT[iDof] = work_cmps_hi[0];
        result_lo_OUT[iDof] = work_cmps_lo[0];
     }   /* iDof */
}
```

最終ブロック内 Recursive Halving 縮約加算

図 4.40: MatVec_Rect_Quad のチューニングの概要 (2)

4.7.6.4　富岳での性能測定結果

富岳での性能測定結果と富岳性能推定結果を対比した図を図 4.41 に示す。表 4.60 に示した区間と図 4.41 に示した区間番号の対応を表 4.62 に示す。

MulMatVec のチューニング、MatVec_Rect_Quad のチューニング、その他のチューニングの効果により高速化が図られていることがわかる。

1BDD反復当たり時間 (ミリ秒)	2020.7版 TUNE 富岳16384P node=32x32x4 tcsds-1.2.26	従来版 ツール推定値 +通信机上 16384P
Rank : Region0がMedian	13238	
2_TMI_Middle_Loop	8.1385	12.6656
3_TMI_MLoop_ZeroClear	0.0018	R8
4_TMI_Clear_BC	0.0000	R8
5_TMI_Constrain_DispBC	0.0000	R8
6_TMI_R7_PartIF_Disp_BC	0.0572	0.6429
7_TMI_Reaction_Force_Copy1	0.0262	R8
8_TMI_R0_MulMAtVec	4.1955	5.0498
9_TMI_Reaction_Force_Copy2	0.0283	R8
10_TMI_Get_Reaction_Force	0.0467	R8
11_TMI_R6_Exchange_Buffering	0.0469	1.0203
12_TMI_R6_Exchange_Superpose	0.0412	-
13_TMI_Comm_Exchange	0.1782	0.2152
14_TMI_Inner_Product1	0.0089	R8
15_TMI_Comm_Getsum1	0.0151	ComGetsum
16_TMI_Scaling	0.0030	R8
17_TMI_Inner_Product2	0.0054	R8
18_TMI_Comm_Getsum2	0.0155	ComGetsum
19_TMI_Update_Vector	0.0020	R8
20_TMI_Inner_Product3	0.0024	R8
21_TMI_Comm_Getsum3	0.0104	ComGetsum
22_TMI_R8_ExecuteBDD_Lcl	0.0088	R8
23_TMI_Set_Mesh	0.0000	R8
24_TMI_R5_Exec_Balancing_Lcl	0.0642	1.6778
25_TMI_Comm_Allgather	1.3161	1.6476
26_TMI_R3_Collect_RHS	0.0852	0.3431
27_TMI_R12_MatVec_Rect_Quad	1.3723	1.7033
28_TMBI_Pre_Exchange	0.4960	
29_TMBI_Pre_Getsum	0.0632	
30_TMBI_Pre_Allgather	0.0278	
31_TMBI_Mloop_post	0.0001	
R8 合計	0.1247	0.3212
ComGetsum 合計	0.0410	0.0444

図 4.41: 富岳性能測定結果と性能推定結果

表 4.62: 性能推定時の区間と性能測定時の区間対応

性能推定時の区間名	性能測定時の区間
DomainFEM	8
CoarseMatVec	27
CollectRhs.P3	26
QuadMatVec.P1	26
Exec.BlancingL	24
Exch.Buffering	11
DomainFem.P1	6
Calc._Others	R8 合計
Exchange	13
Allreduce	ComGetsum 合計
Allgather	25

最終的な「京」に対する性能向上、および消費電力について、図 4.42 に示す。測定の概要は以下の通りである。

- 測定日時　2020/8/9
- コンパイラ・MPI のバージョン：lang/tcsds-1.2.26
- ライブラリ autolinas fccpx -Kfast,ocl,openmp,noeval -SSL2BLAMP -lm
- ライブラリ autonosim mpifccpx -Kfast,ocl,openmp -lm
- ソルバー solver mpifccpx -Kfast,ocl,openmp -SSL2BLAMP –lm –lautolinas -lautonosim

最終的にノーマル・エコなしで 63 倍の性能向上が得られている。性能倍率の大幅な向上の背景は以下の通りである。

- 1 反復あたり 12.7 ミリ秒から 8.1 ミリ秒に短縮した.
- 富岳での通信時間の見積もりを「京」での実測値を元にした机上推定で行っていたため 1 反復当たり通信合計を 1.9 ミリ秒と見積もったが，実測では 1.5 ミリ秒であった.
- 先に示したように演算カーネルのチューニングが進んだため.

システム	京	富岳			
モード	ベースライン	ノーマル エコ無効	ブースト エコ無効	ノーマル エコ有効	ブースト エコ有効
ジョブID		811306	812440	811616	821442
ノード数	32768	4096	4096	4096	4096
プロセス数	32768	16384	16384	16384	16384
部分領域数/プロセス	2	4	4	4	4
総部分領域数	65536	65536	65536	65536	65536
実行時間 [秒]	168470.7	40692.4	39294.9	42722.7	40989.9
対京倍率		63.5	65.7	60.5	63.0
MAX MEMORY SIZE (USE) [GB]		13	13	13	13
消費電力 [MW] 最大と(平均)		21 (19)	23 (19)	20 (18)	23 (19)

※キャパシティ倍率

図 4.42: 富岳の対「京」性能倍率

4.7.7　執筆者

本節の主な執筆者、および執筆協力者の氏名と所属を記す。（主な参加時期における所属）

4.7.7.0.1　執筆者（執筆責任者）

- 南 一生（理化学研究所）
- 宮本 健吾（理化学研究所）

4.7.7.0.2　執筆協力者（５０音順）

- 河合浩志 (東洋大学)
- 山田知典 (東京大学)
- 吉村忍 (東京大学)

4.8 重点課題 7 RSDFT のコデザイン

4.8.1 アプリケーションの特徴とコデザインの方針

新機能デバイスと高性能材料の開発の重点課題 7 の分野において、京では実際のデバイスからチャネル部分のみを切り出したモデルや、ナノ界面を小さな領域に制限したモデルとして、1〜4 万原子、1〜数ケースの規模の、第一原理電子状態の Self-Consistent Field（SCF と略す）計算が実現している。富岳により、数万〜十万原子規模の SCF 計算が日常化すると期待され、デバイス界面の乱れ（界面構造、不完全性、異種原子混入）と界面形成の物理的化学的本質、およびその電子的特性を解明するためのコンビナトリアル探索、界面原子反応と電子・熱輸送特性のデバイス・プロセスシミュレーションが、実デバイス構造で実現できると期待される。

RSDFT は、密度汎関数法による第一原理電子状態計算および構造最適化計算を行うアプリケーションである。演算については、DGEMM 性能が最大限出るようなハードウェアの構成・仕様を必要とする。メモリ量については、システムのメモリ容量を最大限利用することで効率な計算を行うため、システムのトータルのメモリ容量が現システム以下になると、ターゲット問題が実行出来なくなる。また、通信については集団通信が多く、1CMG・1MPI プロセスの構成で Tofu 専用アルゴリズムの適用や、代表プロセス方式等の通信性能が最大となるハードウェアの構成・仕様を必要とする。RSDFT は、これらの観点からコデザインに寄与するアプリケーションである。

コデザインを進めるにあたり、ハードウェア・システム設計のパラメータを基に性能予測を進めながら、アプリケーションが最適化状態となるパラメータや改良を検討し、最適化のためのハードウェア・システム設計に対するコデザイン検討課題の抽出を進めた。次に、システムのパラメータの変更や一部のコデザイン検討結果を反映して性能予測を行い、更にハードウェアの増強による性能予測を行い、結果のフィードバックを進めた。

また、性能予測ではメモリ不足を回避するためにメモリ分散の実装を前提としている。メモリ分散の開発のために、新アルゴリズムの検討とメモリ分散を推進した。

性能・電力の予測結果は、メモリ分散した時の値である。

4.8.2 主要演算 (DGEMM)・通信性能 (集団通信) パラメータの最適化

RSDFT は演算性能が DGEMM、通信性能が集団通信性能に依存するため、それぞれの最適なパラメータによる性能の最大化と性能改善の検討を進めた。

- 演算性能の最適化の検討
  DGEMM の呼び出し部分について、最適な DGEMM パラメータのケースを可能な限り使用するようにした。特に性能への影響が大きいと考えられる行列サイズが大きい時の実行パターンについて、入力パラメータのチューニングを検討した。

  DGEMM は、解くべき問題行列サイズが大きいとデータの再利用性が高まり効率が高くなる。但し、DGEMM の整合寸法 LDA、LDB、LDC ともに 2 の冪乗を避けることで、キャッシュのスラッシングを回避でき、より高性能が期待できる。その他に、以下の条件でより高速な計算が可能になる。

  1. N: thread 分割方向
     N 方向のレジスタブロックサイズ=NRB として、thread 数 ×NRB の倍数で端数が発生せず、より高速に実効可能である。京並びに FX100 では、NRB=4 である。富岳で

は、実機が出てからチューニングにより、NRB の値が決まる（京：32、FX100：64、富岳：12×NRB）。

2. M: SIMD 分割方向
   M 方向のレジスタブロックサイズ=MRB は、京並びに FX100 では、MRB=SIMD 長×8 である。M 方向については、SIMD 長の倍数にすると高速である。但し、富岳では、実機上でのチューニングにより、MRB の値が決まる。（京：16、FX100：32、富岳：64(未定)）。
3. K: ループ展開方向
   K 方向は、ソフトウェアパイプラインを有効にするためループを 2 展開しており、2 の倍数で高速に実行可能である。

但し端数について、富岳では、更に再分割する設計をしているため、このような制限は少なくなると予想される。

- 通信性能の最適化の検討（概要）
  集団通信については、通信性能が最大となるチューニングを以下の 5 点について検討した。それぞれ適用した内容と、初期のシステム構成における性能改善による通信時間の変化を表 4.63 に示す。性能改善は今後も進める。

(1) （旧）代表プロセス方式の適用　→ 空間方向・eigen/X 方向について適用
(2) バンドプロセス方向の通信の効率化　→ 見積式の訂正とプロセス分割パターンを変更
(3) ノード内プロセス数の検討　→ (4) と合わせて適用)
(4) eigen モジュールの通信の効率化　→ eigen モジュールのみノード内 1 プロセス化
(5) （新）代表プロセス方式の改善　→ 新見積式の適用及び改善

なお、以下の通信性能最適化の検討は、基本設計開始時点の 8CMG 構成に対して行われたものであり、本項中での性能評価値は、旧構成での数値を記している。但し本稿中の図並びに最終の性能評価値については現行の 4CMG 構成を用いて記載しているので注意が必要である。

表 4.63: RSDFT の性能改善

通信方向	性能改善毎の実行時間[秒]（初期のシステム構成Type0）				
↓見直し内容=	オリジナル（8CMG）	1)旧代表プロセス方式	2)バンド方向改善	3,4)eigenノード内1プロセス	5)改新代表プロセス方式
↓ np_2d=	24x24x24x3	24x24x24x3	32x48x9x3	32x48x9x3	32x48x9x3
空間	801.7	190.8	188.2	197.5	93.3
バンド	660.7	660.7	85.3	85.3	85.3
barrier(B)	57.2	19.5	18.7	21.4	19.2
eigen/X*	299.3	18.0	18.0	59.7	59.7
eigen/Y	329.1	329.1	329.1	52.0	52.0
(隣接)	32.1	32.1	32.1	32.1	32.1
1SCF通信計	2,180.1	1,250.2	671.3	447.9	341.5
1SCF演算計	899.0	899.0	899.0	923.7	889.2
1SCF合計	3,079.1	2,149.2	1,570.3	1,371.6	1,230.7
(倍率.vs.京)	18.3	26.3	34.7	39.5	45.2
ノード数:	5,184	5,184	5,184	5,184	5,184
eigen:	96x96	96x96	96x96	72x72	72x72

↑演算はeigen時間が増加↑

→ 代表プロセス方式の適用と改善
→ バンド方向のチューニング適用
→（GS/対角ブロックの処理待ち(バンド方向)）
→ 旧代表プロセス方式 → ノード内1プロセスの適用
→ ノード内1プロセスの適用
→ 変化なし

→ 5)はDGEMM性能改善が適用

通信性能を最適化するために検討内容とその結果について詳細を示す。

(1) （旧）代表プロセス方式の適用

富岳は、ノード内複数 CMG 構成のため、1CMG あたり 1MPI プロセスにて並列実行を行う場合、集団通信の通信バンド幅はノード間ネットワークバンド幅の 1 ／ (ノード内プロセス数) となり、ネットワーク性能を生かすことが出来ない。RSDFT においても、1CMG あたり 1MPI プロセスにて並列実行すると、京における通信時間と比べると 2 倍以上の時間がかかり、通信時間が問題となる。

そこで、ノード内の特定のプロセスが代表してノード内通信の集約及びノード間通信を行うことで、ノード間ネットワークバンド幅をフルに使い、通信性能の改善を進めた。この特定のプロセスを代表プロセスと呼び、代表プロセスによる通信方式を代表プロセス方式と呼ぶ。(旧) 代表プロセス方式の考え方を図 4.43 に示す。

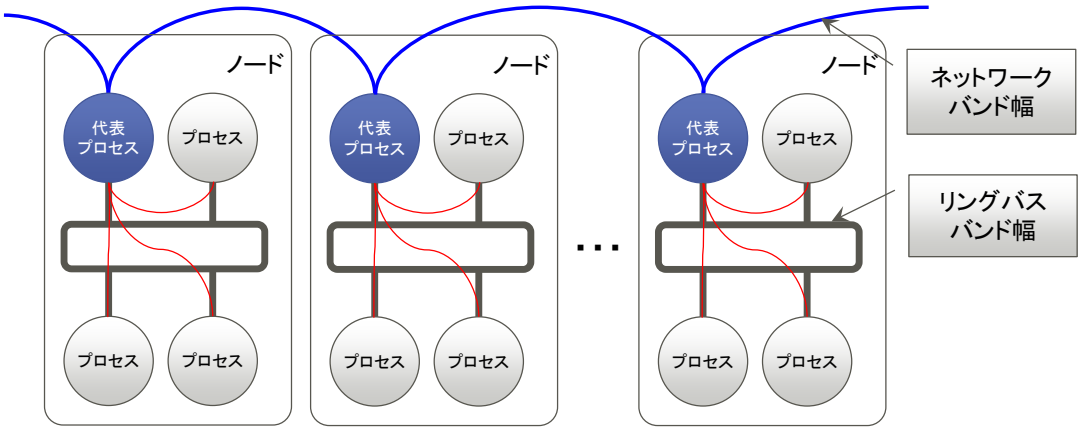


図 4.43: (旧) 代表プロセス方式の考え方

(旧) 代表プロセス方式による通信時間の見積式は以下の通りである。

[通信時間] = [ノード内通信時間] + [ノード間通信時間]

[ノード間通信時間] = [京用の机上計算用アルゴリズムを利用]

[ノード内通信時間:Bcast] = 2*{N/(Bm*Em)+Lm} + (P-1)*{N/(Br*Er)+Lr}

[ノード内通信時間:Reduce] = 2*{N/(Bm*Em)+Lm} + (P-1)*{N/(Br*Er)+Lr} + (P-1)*N/datatype/C

P ；ノード内 MPI プロセス数	C；1 コアの演算性能 [FLOPS]
Bm; 主記憶のバンド幅 [B/s]	Br; リングバスのバンド幅 [B/s]
Em; 主記憶の効率 [%]	Er; リングバスの効率 [%]
Lm; 主記憶のコピーのレイテンシ [s]	Lr; リングバスのレイテンシ [s]

上記の式を RSDFT における空間方向と eigen/X 方向に適用した結果を、表 4.64 に示すように、性能が改善した。

(2) バンドプロセス方向の通信の効率化

代表プロセス方式により、空間方向と eigen/X 方向の通信時間が改善したので、バンド方向

表 4.64: RSDFT の通信性能改善 1

通信時間 [秒]	オリジナル (8CMG)	1) （旧）代表プロセス方式
空間	801.7	190.8
eigen/X	299.3	18.0
通信合計	2180.1	1250.2

の通信のコスト比率が 50%と主要となり、改善が必要である。
バンド方向は空間 Z 方向の分割数を飛び越えてとびとびに通信を行う。各空間のプロセスが同時にバンド方向に通信を行うため、空間 Z 方向のプロセス数分のデータが重なり、バンド幅が（1/Z 方向分割数）となる。
バンド方向 3 分割の例を図 4.44 に示す。

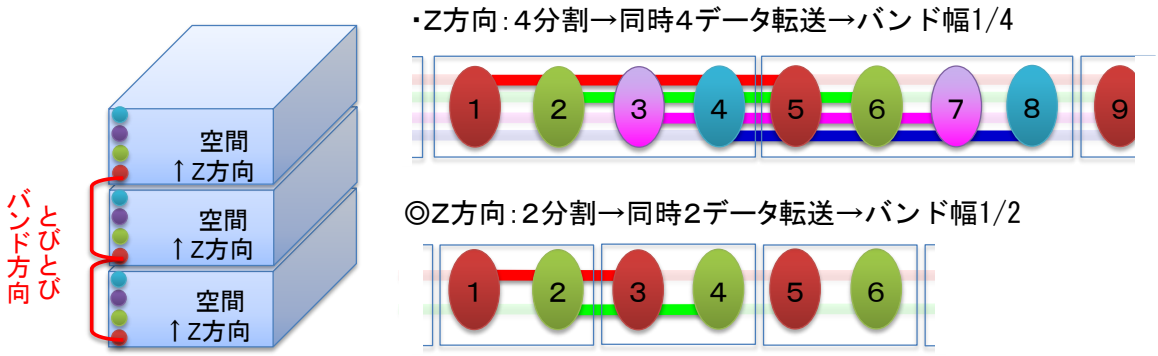


図 4.44: RSDFT のバンド方向のデータ転送

従って、Z 方向のプロセス分割数を小さくすることでバンド方向の通信時間が削減可能となる。そこで、プロセス分割パターンを $24 \times 24 \times 24 \times 3 \rightarrow 48 \times 32 \times 9 \times 3$ へ変更した。バンド方向の通信見積式の訂正と合わせて、表 4.65 に示すように、性能が改善した。

表 4.65: RSDFT の通信性能改善 2

通信時間 [秒]	1) （旧）代表プロセス方式	2) バンド方向改善
バンド	660.7	85.3
通信合計	1250.2	671.3

(3) ノード内プロセス数の検討
代表プロセス方式以外の通信のバンド幅の改善のために、ノード内プロセス数の削減を検討した。ノード内プロセス数の変化により、eigen/Y 方向の通信が改善されるが、複数 CMG 実行により演算性能が低下したため、演算時間が増加し、複数 CMG による実行の条件が必要となることが検討課題となる。

(4) eigen モジュールの通信の効率化
eigen モジュール（以降 eigen と略す）の通信時間は通信全体に占める比率が最大であり、複

数の代表プロセスによる通信により性能改善するが、複数の代表プロセスによる通信を実現する条件があり、検討課題となる。
先の 3) で検討したように、eigen の通信時間としては、ノード内プロセス数が 1 の場合が最小となるので、eigen のみノード内 1 プロセスで実行し、他のモジュールはノード内複数プロセスで実行することで、演算時間の増加を抑え、且つ、eigen の通信時間を最小とすることが可能となる。
尚、eigen のみノード内 1 プロセスの実行を実現するために、eigen を呼び出す処理にて、ランク問い合わせ関数によるランクマッピングを行う必要がある。eigen では $96\times96 = 9216$ 並列で処理するが、ノード内 1 プロセスで実行する場合、実行ノード数が 5184 のため、$72\times72 = 5184$ 並列へ変更する必要がある。更に、ノード形状は $6 \times 32 \times 27$ だがここから 72×72 の 2 次元ランクマッピングが出来ないため、プロセス分割を $48 \times 32 \times 9 \times 3 \rightarrow 32 \times 48 \times 9 \times 3$ へ変更することで、ランクマッピングが可能となる。
また、プロセス数を $96 \times 96 = 9216$ から $72 \times 72 = 5184$ へ変更することで、演算時間が 2-3%程度増加する。
eigen のノード内 1 プロセスによる性能の改善を表 4.66 に示す。

表 4.66: RSDFT の通信性能改善 3、4

通信時間 [秒]	2) バンド方向 改善	3、4)eigen ノード内 1 プロセス
eigen/X	18.0	59.7
eigen/Y	329.1	52.0
通信合計	671.3	447.9
演算時間	899.0	923.7
合計時間	1570.3	1371.6

(5) （新）代表プロセス方式の適用と改善
(1) にて検討した代表プロセス方式を実現に向けて見直した。(新) 代表プロセス方式では以下の手順と図 4.45 に示した 3 次元双方向アルゴリズムにより通信を実現する。

1) 6 分割したデータをノード内の CMG0、CMG7 の 2 つの CMG に分配する

2) 6 分割したデータを CMG0、CMG7 が別の TNI を使用し 3 次元の双方向へ転送する
（転送総ホップ数はX＋Y＋Z、セグメントサイズ毎にパイプラインで転送）

3) 転送後は CMG0、CMG7 が残りの CMG にノード内コピーしデータを共有する

(新) 代表プロセス方式による通信時間の見積式は次の通りである。

[通信時間：Bcast] = A + max(B,C) + soft
A = (M/2) / c1
B = (x+y+z)*(L+seg/Tbw) + (M/6)/seg*max(L,seg/Tbw) + L
C = (M/2) / c1

[通信時間：Reduce] = A + B + C + soft
A = $\log_2$Ncmg * (M/c1 + M/Cbw) + (M/2)/c1
B = (x+y+z)*(L+seg/Tbw+seg/Cbw) + (M/6)/seg*max(L,seg/Tbw,seg/Cbw) + L
C = (M/2) / c1

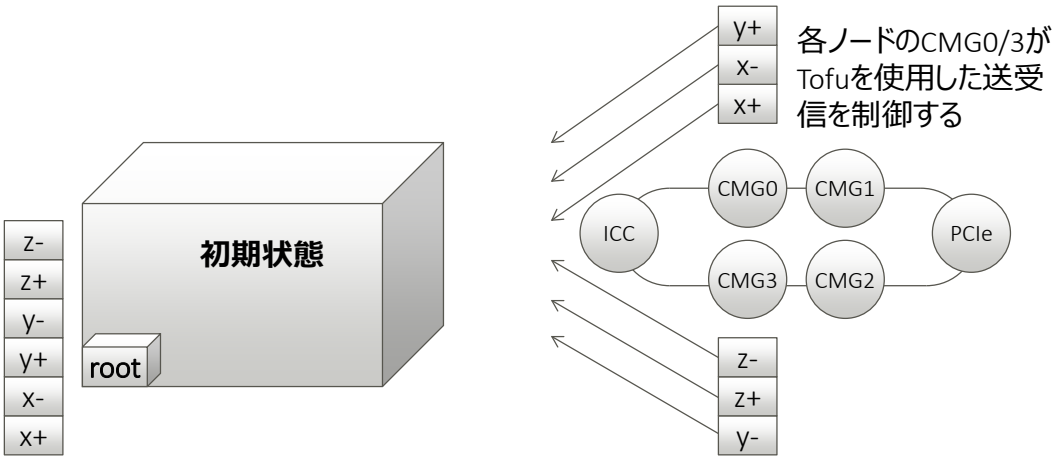


図 4.45: (新) 代表プロセス方式の考え方

M　；メッセージ長
x,y,z；3 次元ノード形状
seg　；セグメント長
Cbw；セグメント長の演算バンド幅
Ncmg; ノード内プロセス数（CMG 数）
soft　；ソフトウェアオーバーヘッド
c1　；1 コアコピー性能
L　；MPI レイテンシ
Tbw; 1TNI バンド幅

Reduce 処理において、ノード内のコピー及び演算は式 A で表され、2 分岐により処理される。処理の考え方を図 4.46 に示す。

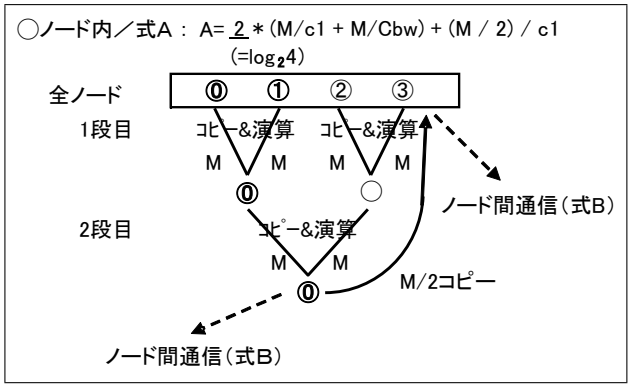


図 4.46: (新) 代表プロセス方式の Reduce の処理

2 分岐によりコピーと演算を行うと、データの転送時間と比べると数倍時間がかかり、メッセージ長が大きい場合にノード内時間が大きく、スループットが高くならない問題がある。そこで、Reduce のノード内通信処理を見直して、2 分岐からパイプライン処理に変更することで、コピーと演算を並行して行い、1 段で処理が可能となり、ノード間通信時間を抑えることが可能である。この見直しにより、Reduce の見積式の式 A は以下のように変更となる。

$$A = (M/Cbw) + (M/2)/c1$$

ノード内通信のパイプライン処理の考え方を図 4.47 に示す。ただし、パイプライン処理により見積式の 2 項目の CMG7 へのコピーが不要と見られる。また、パイプライン処理のセグメント長がメッセージサイズに対して小さいことが前提であり、見直しが検討課題である。

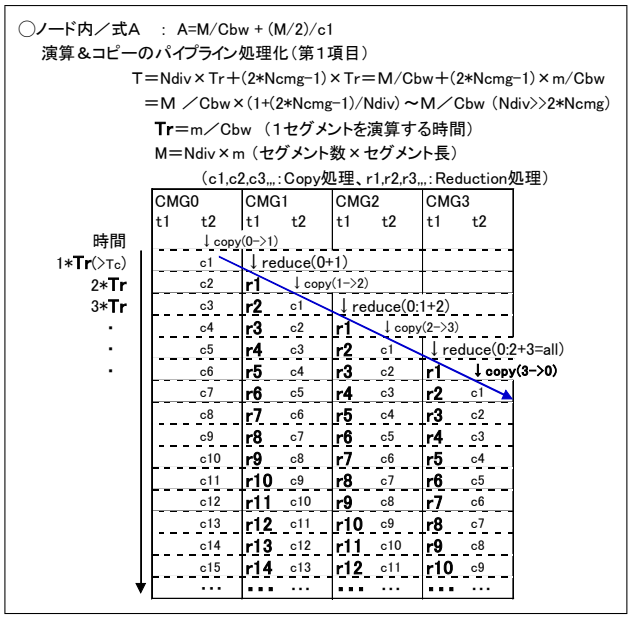


図 4.47: (改新) 代表プロセス方式の Reduce の処理

この見直した (改新) 代表プロセス方式により、表 4.67 に示すように性能が改善した。演算時間については、DGEMM の見積式の見直しが適用されている。

表 4.67: RSDFT の通信性能改善 5

通信時間 [秒]	3、4)eigen ノード内 1 プロセス	5)(改新) 代表プロセス方式
空間	197.5	99.5
通信合計	447.9	347.9
演算時間	923.7	889.2
合計時間	1371.6	1237.2

以上の全ての通信性能の見直しにより、通信時間が先に示した表 4.63 の 1) → 5) へ変化し、6 倍程度性能が改善した。

4.8.3 メモリ分散アルゴリズムの検討

RSDFT のターゲット問題の計算において、現状のコードでは 1 プロセスあたり 10.8GB のメモリ使用量となり、富岳ではメモリ不足となるため、メモリ分散を実装してメモリ不足を解消する必要がある。オリジナルのコードでは未適用であるバンドプロセス方向のメモリ分散を実装す

ることで、1 プロセスあたり 5.4GB （MPI と OS の使用量を除く）のメモリ使用量と見積もられた。性能・電力の予測結果は、メモリ分散した場合の値である。

メモリ分散の開発のために、新アルゴリズムの検討とメモリ分散の試行を進めた。

4.8.3.1 メモリ分散に向けての新アルゴリズムの検討

直交化処理では、全てのバンド方向の波動関数を持っていることを前提として、バンドプロセス並列を実現している。また、直交化処理は実行時間の観点から主要な処理の 1 つであり、バンドプロセス方向のメモリ分散を実現する上で、直交化処理のメモリ分散は重要な課題である。そこで、直交化処理について、メモリ分散を可能とするアルゴリズムを評価し、採用するアルゴリズムの検討を行なった。

4.8.3.1.1 2015 年度実施

・内容：直交化処理のアルゴリズムとして、以下の 3 つのアルゴリズムについて、テストプログラムにより性能を評価した。

1) 高橋 GS(TGS：従来法)　：現状のコードの Gram-Schmidt 直交化処理
2) コレスキー分解法 (LUGS)　：ScaLAPACK のコレスキー分解ルーチン
3)QR 分解法 (QRGS)　：ScaLAPACK の QR 分解ルーチン

・状況：1MPI プロセス当りの規模をターゲット問題と同様の規模とし、バンドプロセス方向のメモリ分散無しの場合のテストプログラムを実行して、各アルゴリズムの性能を確認した。実行時間と演算ピーク比を以下の表 4.68 に示す。

1)TGS では、行列の対称性により三角部分の処理となっているため、実行時間が小さい傾向にある。576 並列と 1152 並列ではブレがあるが、実行時間は約 14 秒でピーク比は約 43%である。

2)LUGS では、フル行列を処理して演算数が約 2 倍となっているため、ピーク性能は約 52%と高いが実行時間が約 17 秒と TGS よりかかっている。

3)QRGS では、最適化されていない ScaLAPACK/QR 分解ルーチンを使っているため、性能が極端に低く、性能の良い QR 分解ルーチンを調査する。

表 4.68: RSDFT 新アルゴリズムの性能（2015 年度）

並列数	実行時間 [秒](演算ピーク比)		
	1)TGS	2)LUGS	3)QRGS
288	14.3 (42.9%)	17.6 (52.6%)	384.6 (1.3%)
576	24.7 (24.8%)	17.6 (52.2%)	386.8 (1.3%)
1152	19.0 (32.1%)	17.0 (54.2%)	391.4 (1.3%)
1728	13.8 (44.4%)	17.7 (51.9%)	392.2 (1.3%)

4.8.3.1.2　2016 年度実施

新アルゴリズムの検討として、LU(コレスキー) 分解法について、性能改善の試行とメモリ分散化の試行を進めた。実施した項目を以下に示す。全体の進捗状況を表 4.69 に示す。

(1) LU(コレスキー) 分解法の性能改善の試行
(2) LU(コレスキー) 分解法のメモリ分散化の試行

以下に、それぞれの項目について実施内容を説明する。

表 4.69: RSDFT 新アルゴリズムの検討状況 (2017 年度)

アルゴリズム案	性能評価	メモリ分散試作	メモリ分散評価
TGS（高橋 GS）	○	○	○
LU（コレスキー）分解法	○	○	○
QR 分解法	△ (簡易調査)	−	−
SD 一般化固有値問題	未	未	未

(1) LU(コレスキー) 分解法の性能改善の試行
メモリ分散無しの LU(コレスキー) 分解法のテストプログラムについて、無用な処理の削除、並列数の絞り込みの修正、三角行列の行列積のライブラリ化 (dtrmm) を行い、性能改善を試行した。

オリジナルプログラムでは、コレスキー分解ルーチン pdpotrf と逆行列計算ルーチン pdtrtri の間に上三角化の処理が含まれていたが、pdpotrf ルーチンは上三角行列を出力するため不要となるので、上三角化の処理を削除した。

次に、一部のプロセスを使って LU(コレスキー) 分解法の計算を行う際に、データを収集する処理を全プロセスで行っていたため計算結果が異なっていたので、担当する範囲でのみデータを収集するように修正を行った。

最後の波動関数の更新の処理は、オリジナルプログラムでは DGEMM を使っていたが、波動関数に掛ける行列は三角行列なので、三角行列用の行列積ルーチン DTRMM を使うことで実行時間を短縮した。効率は DGEMM が約 93%、DTRMM が約 62%と低下するが、DGEMM がフル行列に対して DTRMM は三角行列のみのため演算数が少なく、実行時間が短い。

以上の試行により、LU(コレスキー) 分解法の性能を改善した。ScaLAPACK では計算に使うプロセスの範囲を絞り込んで実行することが可能である。改善したプログラムについて、プロセス範囲の変化による性能を表 4.70 に示す。プロセス範囲は 12×12 から 24×24 が最大の性能であった。この最大性能は TGS と比較して同等の性能である。

(2) LU(コレスキー) 分解法のメモリ分散化の試行
LU(コレスキー) 分解法のテストプログラムのメモリ分散版を調査した。大規模問題の実行を想定すると、以下の問題点があることがわかった。

(a) 波動関数の初期化時間の増加
初期化処理において、全波動関数の初期化を 1 ランクで行っているため、大規模問題になると時間がかかる。初期化を並列化して、各プロセスが担当する範囲のみの波動関数を初期化することが必要である。

表 4.70: LUGS の性能（プロセス範囲の変化）

○FAPP結果（Application/MAX時間[秒]/(ピーク比：全ﾌﾟﾛｾｽの演算数をMAX時間で除算したピーク比)）

node=	TGS	LUGS/NPROWxNPCOL[sec](peak)					
	[sec](peak)	4x4	6x6	8x8	12x12	24x24	(all)
288	16.121 (38.0%)	22.256 (27.6%)	18.051 (34.0%)	15.708 (39.1%)	15.095 (40.7%)	14.979 (41.0%)	- -
576	12.923 (47.4%)	31.784 (19.3%)	17.676 (34.7%)	15.080 (40.7%)	14.844 (41.3%)	14.702 (41.7%)	- -
1152	14.675 (41.7%)	23.427 (26.2%)	17.796 (34.5%)	15.716 (39.0%)	14.960 (41.0%)	14.760 (41.6%)	15.563 (39.4%)
1728	14.520 (42.2%)	28.149 (21.8%)	18.147 (33.8%)	15.687 (39.1%)	14.636 (41.9%)	15.165 (40.4%)	16.134 (38.0%)

288:16x18↑　↑1152:32x36
↑1728:36x48

(b) 作業領域不足

LU(コレスキー) 分解法の初めに、グラム行列 S を全体行列として生成するため、MB × MB の領域を確保する。大規模問題では MB が大きいので領域不足となる。従って、全体行列を作らず、部分行列を初めから作成するように改善が必要である。

(c) 行列積の実行時間の増加

最後の波動関数の更新処理の際に、ScaLAPACK の dGEBR2D ルーチンを使って ScaLAPACK のプロセス方向に行列を転送するが、大規模問題になると dGEBR2D の実行時間がその後の行列積の 10 倍以上の時間がかかる。従って、行列の転送処理を Bcast 通信等により改善が必要である。

4.8.3.1.3　2017 年度実施

詳細設計 (3) では、新アルゴリズムの検討として、LU(コレスキー) 分解法について、メモリ分散の実装並びに性能評価を進めた。

(1) LU(コレスキー) 分解法のメモリ分散化の実装
(2) LU(コレスキー) 分解法のメモリ分散版のオーバーヘッドの評価並びに性能チューニング

以下に、詳細の実施内容を記述する。

(1) LU(コレスキー) 分解法のメモリ分散化の実装

詳細設計 (2) までに試作したテストコードに対して、2017 年度前半の詳細設計 (3) 開発期間 I ではメモリ分散化の実装を行った。LU(コレスキー) 分解法による直交化は 4 つの手順で構成される。グラム行列要素 S を生成する前半部分の開発が完了した。外部呼び出しされる ScaLAPACK の制約であるバンド数が分割数で割り切れる条件で動作することが確認できている。メモリ分散化により、データの収集や計算結果の配布など通信が新たに発生しうるが、メモリ分散前に三角行列の行列積になっていた演算内容が分散化された行列積に置き換わるため演算効率が高くなる可能性がある。但し、波動関数はブロックサイクリック分割をされているため、$\Phi \cdot B$ 計算での B が上三角行列であり、工夫が必要である。

バンド方向のメモリ分散化はメモリ使用量を減らすことを目的として行っている。しかし、新たに発生する通信を削減するために通信と演算をオーバーラップさせると余分な作業配列が必要となり、今まで測定で用いてきたバンド方向の分割数が 3 ではメモリ使用量の節約にはならない。

またクラスターマシンで性能を測定したところ、ブロックサイズを小さくすると、ScaLAPACK へ行列を持ち直す部分のオーバヘッドがブロックサイズの逆数の二乗で処理時間が増

大した。波動関数の持ち替えに必要なデータ量はブロックサイズに依存しないため、データ持ち替えのレイテンシが見えているためと想定される。

(2) LU(コレスキー) 分解法のメモリ分散版のオーバーヘッドの評価並びに性能チューニング
詳細設計 (3) 開発期間 II では、上記のオーバーヘッドについて、blacs サービスルーチン pdgemr2d への置き換えを行い、性能評価を実施した。この評価により、区間 ovlp の実行時間は 17.3[s] → 0.453[s] に削減され、ブロックサイズが小さい時に見られたオーバーヘッドは解消された (図 4.48)。しかし、初期化処理 (init) が 10 倍の時間かかるようになった。初期化処理の内訳を調べると、ScaLAPACK 用のプロセスグリッド (4 × 6 プロセス分割) を準備する処理 (init2) が、8grid × 3band のプロセスグリッドを用意する処理 (init1)(図 4.49) と比べて時間がかかっていた。今回評価したテストケースでは、全体に占める時間の割合は 1 割程度であった・

Ngrid = 32^3 = 32,768
Nband = 1536
PROCS = 2^3 (grid) × 3 (band) = 24
NPROW × NPCOL = 4 × 6
MBSIZE = NBSIZE = (1) 32　(2) 64　(3) 128

Old data (2017/9/25)

	rank	init	ovlp	pdpotr	pdtrtr	gather	dtrmmb	finali	tot
(1)	0	0.011	17.364	0.210	0.131	0.063	0.371	0.000	18.150
(2)	0	0.013	4.562	0.203	0.162	0.063	0.353	0.000	5.355
(3)	0	0.019	1.350	0.194	0.161	0.063	0.348	0.000	2.135

New data (2017/12/22)

	rank	init	ovlp	pdpotr	pdtrtr	bdtrmm	finali	tot
(1)	0	0.140	0.453	0.120	0.027	0.392	0.000	1.131
(2)	0	0.141	0.464	0.107	0.025	0.381	0.000	1.118
(3)	0	0.143	0.385	0.149	0.109	0.389	0.000	1.175

	rank	ini1	ini2	ini3	ovlp	pdpotr	pdtrtr	bdtrmm	finali	tot
(3)	0	0.005	0.123	0.000	0.377	0.147	0.110	0.376	0.000	1.138

図 4.48: LU による直交化おける実行時間の内訳の変化

4.8.3.2　直交化処理におけるメモリ分散アルゴリズムの性能評価

RSDFT 本体の主要な処理について、バンドプロセス方向のメモリ分散を試作して、性能評価と性能改善を推進中である。

4.8.3.2.1　2015 年度実施

・目的：従来の直交化アルゴリズム (TGS) について、バンドプロセス方向のメモリ分散の実装を試行することで、メモリ分散の動作の確認、実装方法の特定を進める。
・内容：(1) と同様のアルゴリズムのテストプログラムに対して、Gram-Schmidt 処理のモジュールにメモリ分散を適用し、インターフェースルーチンにより分散の前後の整合性を取って実装する方式を試行した。

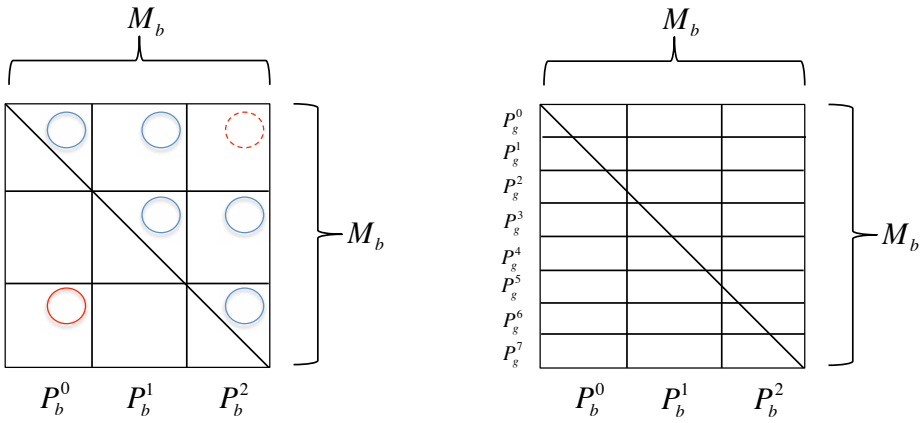


図 4.49: LU 分解における上三角行列の並列分散方法、左図：RSDFT 内部の並列分散方法、右図：ScaLAPACK での並列分散方法

また、Gram-Schmidt 直交化モジュール以外の部分空間対角化モジュールと共役勾配法モジュールについても、メモリ分散の方法を検討した。

・状況：Gram-Schmidt 処理のモジュールは、バンドプロセス方向の手続きの分割が適用されているので、波動関数の配列を担当バンド分のみ持つようにし、ブロック・サイクリックに受け持つ担当分が連続となるように修正した。修正したコードを、小規模データ・バンド並列のみの場合について、動作確認を行い、計算結果が合うことを確認した。

その他のモジュールについて、部分空間対角化モジュールでは、他のプロセスの担当バンド分の波動関数を参照する必要があり、通信処理が増える見通しである。共役勾配法モジュールでは、他のプロセスのデータを参照しないので、メモリ分散された配列のアドレスの変更により対応が可能な見通しである。

4.8.3.2.2　2016 年度実施

メモリ分散の試行として、TGS の性能改善と対角化処理 (SD) のメモリ分散版の試作・評価を行い、メモリ分散の動作の確認、実装方法の特定を進めた。実施した項目を以下に示す。全体の進捗状況を表 4.71 に示す。

(1) Gram-Schmidt(GS) 処理の TGS アルゴリズムのメモリ分散版の性能評価
(2) TGS アルゴリズムに対するチューニングの実施と性能評価
(3) Subspace-Diag(SD) 処理のメモリ分散化の実施と大規模問題により性能評価
(4) SD 処理内の固有値計算に EigenExa を利用するための連携部分のメモリ分散化

各項目の詳細な内容を以降にて記述する。

(1) TGS のメモリ分散版の性能評価
TGS のカーネルプログラムのオリジナル版とメモリ分散版の性能を測定した。測定条件を表 4.72 に、測定した結果を表 4.73 に示す。バンド並列数を 4 として実行し、各バンドプロセスにおいてランク番号が一番小さいプロセスの実行時間を比較した。メモリ分散のための通信が追加されているため、メモリ分散版の方が実行時間が大きい傾向ではあるが、概ね同様の性能である。

表 4.71: RSDFT メモリ分散の試作・評価の状況 (2016 年度)

手続き	メモリ分散試作	性能評価	チューニング
CG/レイリー商最小化	未	未	未
GS/規格直交化	済	済	済/メモリ分散未
SD/部分対角化	済	済	未
/Eigen 連携	済	未	未
CPMD 計算	未	未	未

表 4.72: TGS/メモリ分散版の性能測定ケース

ケース	1	2	3	4
node	1152	2304	4608	6912
PROCS	$12 \times 12 \times 2 \times 4$	$12 \times 12 \times 4 \times 4$	$12 \times 12 \times 8 \times 4$	$12 \times 12 \times 12 \times 4$
ML	$192 \times 192 \times 36$	$192 \times 192 \times 72$	$192 \times 192 \times 144$	$192 \times 192 \times 216$
ML0	4608			
MB	9216			
NBLK	1152			

(2) TGS のチューニング・性能評価

TGS のカーネルプログラムに対して、対角部を優先的に処理を行い、非同期通信により転送を行いながら非対角部の処理を行う、チューニングを適用した。これにより、対角部の処理待ちによるインバランスを解消することが可能である。

TGS のカーネルプログラムについて、オリジナル (ASIS)、旧 RSDFT 本体に実装されているチューニング「対角移動」と今回チューニングを行った「対角優先」の 3 つのプログラムについて性能を測定して評価した。測定条件は、空間並列数変化、ブロックサイズ変化とバンド並列数変化の 3 種類である。以降では性能評価の結果を説明する。尚、メモリ分散は適用せずに測定したものである。

(a) 空間並列数変化によるチューニング性能

空間方向の並列数の変化による測定条件を表 4.74 に示す。ターゲット問題に合わせてバンド並列数を 3 とし、1 プロセス当りの処理量を同等となるウィークスケールにより測定を行った。

測定結果は、各並列数で同様の実行時間であった。代表してケース 1-1 と 1-5 の測定した結果のグラフを図 4.50 に示す。ケース 1-5 については各処理における演算・通信の実行時間の結果を表 4.75 に示す。ASIS に比べ、対角移動はバンドプロセス方向のロードインバランスが改善されるので速くなり、対角優先では更にインバランスが改善され速くなっている。

通信時間について、対角部・非対角部それぞれの時間を図 4.51 に示す。対角移動では処理する対角部のブロック数がバンドプロセスによって異なるため、オリジナル・対角優先と同じであるバンドプロセス#1 の通信時間を比較した。並列数が増加すると (No.1-1→1-5)、通信時間が増加する傾向である。対角部ではチューニングによる違いはないが、非対角部の通信時間は ASIS < 対角移動 < 対角優先であり、チューニングによって増加する傾向にある。これは、対角移動では非対角部の処理中にバンドプロセス#2 の対角部をバンドプロセス#0 へ送信するため、通信の衝突が発生する。対角

表 4.73: TGS/オリジナル版とメモリ分散版の比較

オリジナル版（バンドプロセス数4）							メモリ分散試行版（バンドプロセス数4）						
(node)	実行時間[秒]					rank	(node)	実行時間[秒]					rank
	diag	barrier	bcast	nondiag	total			diag	barrier	bcast	nondiag	total	
1152	1.55	3.84	0.13	0.94	6.46	0	1152	1.78	4.19	0.13	0.94	7.04	0
	0.75	4.00	0.11	1.60	6.46	288		0.77	4.27	0.13	1.88	7.04	288
	0.75	3.57	0.13	2.02	6.46	576		0.78	3.88	0.13	2.26	7.04	576
	0.75	3.13	0.11	2.46	6.45	864		0.78	3.57	0.13	2.57	7.04	864
2304	1.72	4.35	0.18	0.99	7.25	0	2304	2.00	5.70	0.30	0.99	8.99	0
*1)	0.80	4.32	0.18	1.94	7.25	576	*1)	0.82	4.64	0.30	3.23	8.99	576
	0.79	3.91	0.15	2.40	7.25	1152		0.82	4.63	0.30	3.25	8.99	1152
	0.80	3.27	0.14	3.03	7.24	1728		0.82	3.40	0.30	4.47	8.99	1728
4608	2.45	4.56	0.63	1.00	8.64	0	4608	1.81	4.54	0.61	1.00	7.96	0
	0.88	5.28	0.54	1.94	8.64	1152		0.90	4.64	0.61	1.81	7.97	1152
	0.88	4.75	0.63	2.39	8.64	2304		0.92	4.06	0.61	2.37	7.97	2304
	0.87	4.57	0.54	2.61	8.59	3456		0.90	3.73	0.61	2.73	7.97	3456
6912	2.17	5.06	0.94	1.09	9.26	0	6912	1.99	5.01	0.94	1.04	8.98	0
	0.92	5.27	0.80	2.27	9.26	1728		0.95	4.93	0.93	2.17	8.98	1728
	0.93	4.30	0.93	3.10	9.26	3456		0.94	4.48	0.94	2.62	8.98	3456
	0.99	4.64	0.80	2.76	9.19	5184		0.94	4.36	0.93	2.75	8.98	5184

*1)2304並列にて計算結果の一致を確認

表 4.74: TGS/チューニングの性能測定ケース（空間並列数変化）

ケース	1-1	1-2	1-3	1-4	1-5
node	648	5184	10368	20736	41472
PROCS	$4 \times 6 \times 9 \times 3$	$16 \times 12 \times 9 \times 3$	$16 \times 24 \times 9 \times 3$	$16 \times 48 \times 9 \times 3$	$32 \times 48 \times 9 \times 3$
空間並列数	216	1728	3456	6912	13824
バンド並列数			3		
ML	$68 \times 102 \times 153$	$272 \times 204 \times 153$	$272 \times 408 \times 153$	$272 \times 816 \times 153$	$544 \times 816 \times 153$
ML0			4913		
MB			110592		
NBLK			1536		
ブロック数			72		
Node 形状	$4 \times 27 \times 6$	$16 \times 27 \times 12$	$16 \times 27 \times 24$	$32 \times 27 \times 24$	$48 \times 27 \times 32$

優先では、非対角部の処理中に対角部を非同期でバンド方向に転送するため、通信の衝突が発生する。（対角移動で 15%、対角優先で 30%程度の通信時間の劣化がある）

(b) ブロックサイズ変化によるチューニング性能

ブロックサイズ NBLK の変化による測定条件を表 4.76 に示す。ケース 1-1 に対して、ブロックサイズを 2 倍・1 倍・1/2 倍・1/4 倍・1/8 倍と変化させた。

測定した結果、NBLK=1536 のケース 2-1(1-1) が最速のケースであった。代表してケース 2-2(1-1) と 2-5 の測定した結果のグラフを図 4.50 に示す。ケース 2-2(1-1) については、各処理の演算・通信の実行時間を表 4.77 に示す。ブロックサイズが小さくなると、DGEMM の効率が低下するので、演算時間が大きくなり全体の時間が増加する。

(c) バンド並列数変化によるチューニング性能

バンド方向の並列数の変化による測定条件を表 4.78 に示す。ケース 1-1 に対して、バンド並列数を 1,2,3,4,6 と変化させたストロングスケールを測定した。

測定した結果、並列数の増加と共に実行時間が短縮した。ケース 3-3(1-1) と 3-6 の測定した結果のグラフを図 4.53 に示す。ケース 3-6 については各処理における演算・通信の実行時間の結果を表 4.79 に示す。

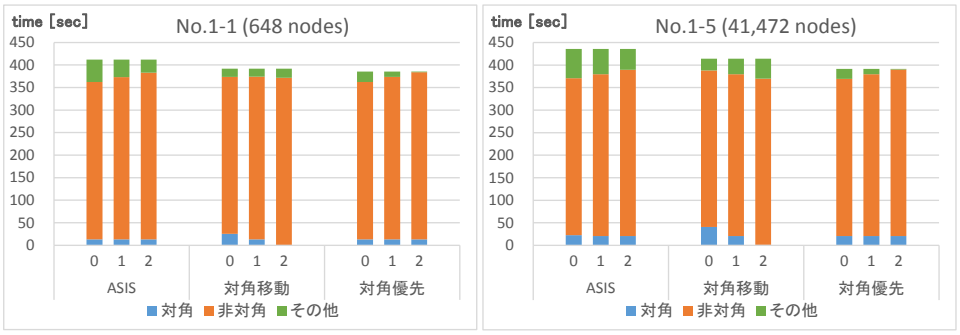


図 4.50: TGS/チューニング性能（空間並列数変化/No.1-1,1-5）

表 4.75: TGS/チューニング性能（空間並列数変化/No.1-5）

No.1−5	チューニング		ASIS			対角移動			対角優先		
	バンドプロセス		0	1	2	0	1	2	0	1	2
並列数	全体		435.52	435.52	435.52	414.25	414.25	414.25	391.33	391.33	391.29
・全体	対角		22.58	20.69	20.58	40.65	20.65	0.93	20.8	20.7	20.76
41,472		演算	8.46	8.45	8.46	16.67	8.47	0.35	8.49	8.48	8.48
・空間		通信	4.9	4.9	4.88	9.55	4.88	0.22	4.9	4.9	4.89
13,824		同期	9.22	7.34	7.24	14.42	7.3	0.36	7.4	7.32	7.39
・バンド	非対角		348.03	358.81	368.85	347.53	359.18	369.1	348.5	358.93	368.98
3		演算	338.73	348.86	358.55	339.24	348.98	358.54	339.01	348.89	358.75
		通信	6.11	6.28	6.48	6.12	7.07	7.4	7.66	8.04	8.17
[単位：		同期	3.19	3.67	3.82	2.18	3.13	3.16	1.83	2.01	2.06
秒]	その他		64.91	56.02	46.09	26.08	34.42	44.22	22.03	11.69	1.54

※1：対角・非対角の経過時間には、空間方向のMPI_Allreduce通信を含む。
※2：その他の経過時間は、主にバンド方向での対角成分の通信とインバランスである。

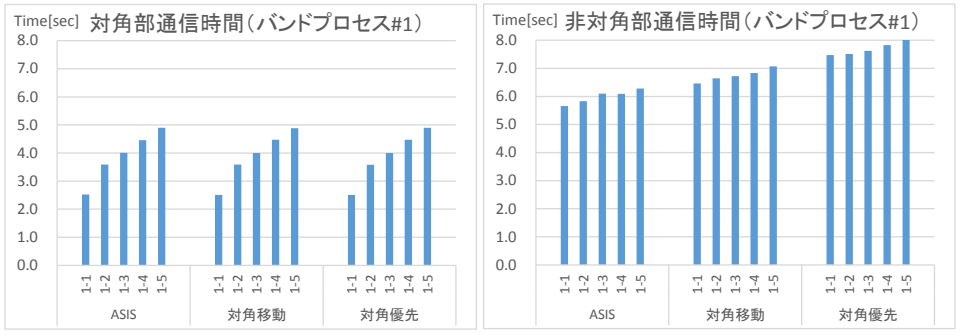


図 4.51: TGS/通信時間（空間並列数変化）

表 4.76: TGS/チューニングの性能測定ケース（ブロックサイズ変化）

ケース	2-1	2-2(1-1)	2-3	2-4	2-5
node	648				
PROCS	$4 \times 6 \times 9 \times 3$				
空間並列数	216				
バンド並列数	3				
ML	$68 \times 102 \times 153$				
ML0	4913				
MB	110592				
NBLK(ブロックサイズ)	3072	1536	768	384	192
ブロック数	36	72	144	288	576
Node 形状	$4 \times 27 \times 6$				

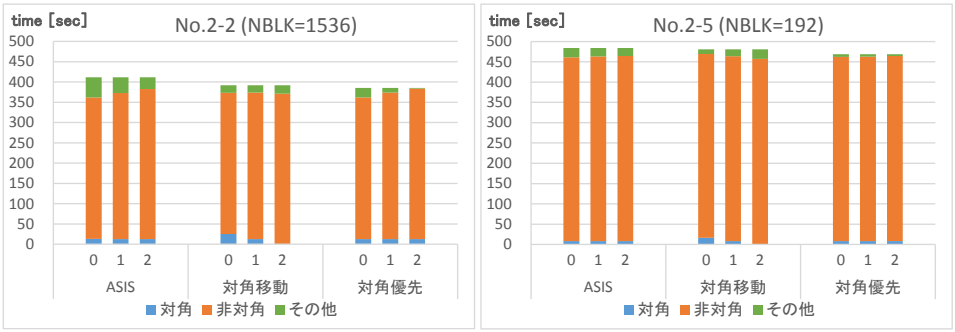


図 4.52: TGS/チューニング性能（ブロックサイズ変化/No.2-2,2-5）

表 4.77: TGS/チューニング性能（ブロックサイズ変化/No.2-2）

No.2-2	チューニング		ASIS			対角移動			対角優先		
	バンドプロセス		0	1	2	0	1	2	0	1	2
並列数	全体		411.66	411.65	411.65	391.89	391.89	391.89	385.50	385.49	385.46
648	対角		13.19	12.97	12.96	25.55	12.95	0.55	13.02	12.94	12.94
		演算	8.48	8.47	8.48	16.56	8.45	0.35	8.46	8.45	8.45
NBLK		通信	2.61	2.52	2.50	4.98	2.51	0.11	2.53	2.51	2.50
1,536		同期	2.09	1.98	1.98	4.02	1.99	0.09	2.04	1.98	1.99
blk数	非対角		348.85	359.99	369.92	347.81	361.00	370.82	349.19	360.78	370.89
72		演算	338.71	352.38	358.95	338.15	349.19	359.03	338.97	349.09	361.98
		通信	5.54	5.66	5.96	5.48	6.46	6.80	7.23	7.47	7.60
[単位:		同期	4.60	1.95	5.01	4.19	5.36	4.99	2.99	4.22	1.31
秒]	その他		49.62	38.70	28.78	18.53	17.94	20.52	23.29	11.78	1.63

表 4.78: TGS/チューニングの性能測定ケース（バンド並列数変化）

ケース	3-1	3-2	3-3(1-1)	3-4	3-5
node	216	432	648	864	1296
PROCS	$4 \times 6 \times 9 \times 1$	$4 \times 6 \times 9 \times 2$	$4 \times 6 \times 9 \times 3$	$4 \times 6 \times 9 \times 4$	$4 \times 6 \times 9 \times 6$
空間並列数			216		
バンド並列数	1	2	3	4	6
ML			$68 \times 102 \times 153$		
ML0			4913		
MB			110592		
NBLK			1536		
ブロック数			72		
Node 形状	$4 \times 9 \times 6$	$4 \times 18 \times 6$	$4 \times 27 \times 6$	$4 \times 32 \times 6$	$8 \times 9 \times 18$

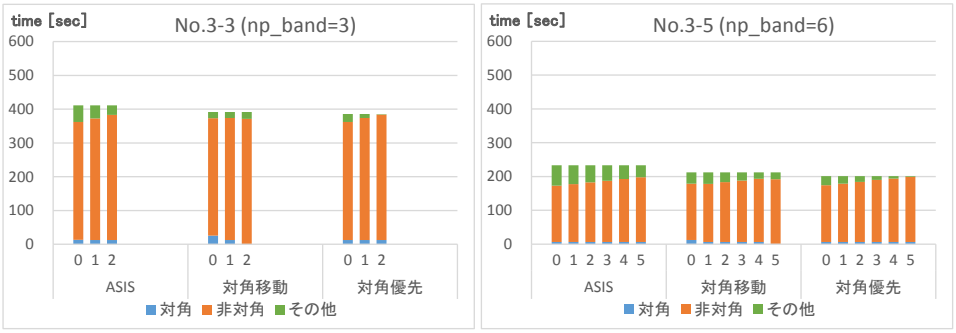


図 4.53: TGS/チューニング性能（バンド並列数変化/No.3-3,3-5）

表 4.79: TGS/チューニング性能（バンド並列数変化/No.3-5）

No.3-5	チューニング		ASIS						対角移動						対角優先					
	バンドプロセス		0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
並列数	全体		233.38	233.38	233.38	233.38	233.38	233.36	212.42	212.42	212.42	212.42	212.42	212.4	201.08	201.08	201.09	201.09	201.08	201.01
・全体	対角		6.68	6.54	6.5	6.47	6.5	6.5	12.7	6.48	6.72	6.48	6.49	0.54	6.49	6.46	6.47	6.46	6.47	6.48
1,296		演算	4.24	4.24	4.23	4.23	4.23	4.23	8.11	4.23	4.23	4.23	4.23	0.35	4.23	4.23	4.23	4.24	4.23	4.23
・空間		通信	1.27	1.26	1.26	1.26	1.26	1.26	2.44	1.26	1.27	1.25	1.26	0.11	1.27	1.26	1.26	1.25	1.25	1.26
216		同期	1.17	1.03	1.01	0.99	1	1.01	2.15	0.99	1.22	0.99	1	0.08	1	0.97	0.98	0.97	0.99	1
・バンド	非対角		166.4	171.25	176.28	181.24	186.35	191.67	166.29	171.71	176.63	181.43	186.75	191.27	167.77	172.65	178.03	183.44	187.71	192.85
6		演算	161.48	167.03	171.72	176.91	181.85	186.44	161.98	167.18	171.58	176.89	181.87	186.61	162.06	167.57	172.31	178.32	183.2	186.77
		通信	2.66	2.73	3.11	2.89	2.96	3.36	2.63	3.01	3.11	2.99	3.25	3.26	4.23	3.73	4.46	4.6	3.99	4.75
[単位:		同期	2.26	1.49	1.44	1.44	1.53	1.87	1.68	1.53	1.94	1.56	1.63	1.41	1.49	1.35	1.27	0.52	0.52	1.32
秒]	その他		60.3	55.6	50.61	45.67	40.54	35.19	33.44	34.23	29.07	24.51	19.19	20.58	26.82	21.97	16.59	11.2	6.9	1.68

また、バンド並列数が 1 に対する実行時間の短縮比率を並列効果として算出した結果を表 4.80 に示す。チューニングを行うことで並列効果も増加して、対角優先の場合が一番高いことがわかる。

表 4.80: TGS/並列効果（バンド並列数変化）

バンド並列数	1	2	3	4	6
ASIS	1.00	1.89	2.71	3.44	4.78
対角移動	1.00	1.94	2.84	3.68	5.24
対角優先	1.00	1.96	2.89	3.79	5.54

4.8.3.2.3　2018 年度実施

詳細設計 (4) では、詳細設計 (3) 開発期間 I に試作した最適化 GS カーネルの 1 つである、対角優先アルゴリズムのメモリ分散化並びに高並列での性能評価を実施した。

詳細設計 (4) 開始当初の GS カーネルのメモリ分散化状況は、オリジナルの TGS についてのみ詳細設計 (2) に完了しており、詳細設計 (3) 開発期間 II で実施した統合によって、全体を通したメモリ分散性能の評価は可能であった。2011 年の Gordon Bell 賞受賞時に導入された対角移動アルゴリズムについてはメモリ分散化を実施していない。今回試作を実施した対角優先アルゴリズムは、対角優先により付け加えられた動的負荷分散機能のメモリ分散を行えばよい。このためメモリ分散化は TGS でのメモリ分散版をベースに、試作可能である。詳細設計 (3) 開発期間 I に試作した非メモリ分散版では、動的スケジューリングを行うため、計算に必要な対角成分の波動関数を全て保持する実装をしていた。調査の結果、この対角領域の一時配列によるメモリ使用量はワーク配列と波

動関数データを合わせて 2.7[GB] と少なくない分かった。このため対角領域の保持する世代を制限することで、一時領域のメモリ使用量は、$ML0 \times NBLK \times 8 \times n = 57.6 \times n[MB](n = 1 \sim 144)$ と表され、理論上は最大で 64 世代分程度のデータを保持することが可能であるが、全ての対角成分を持つのは省メモリの観点から好ましくないと判断した。そこでメモリ分散のために、保持する対角成分一時領域の世代数を調整可能になるよう変更した。

試作した対角優先 (FGS) のメモリ分散版について、41,472 プロセスでの大規模性能評価を実施した。測定パターンは、オリジナルの TGS メモリ分散版で測定したパターン [表 4.74]、[表 4.76]、[表 4.78] に加えてバンド方向のプロセス分割を 3 → 6 に増やしたものを追加し [表 4.81]、対角優先で使用する対角成分の世代数を 3、12 のものを評価した。バンド分割数が増大すると 1 プロセスが担当する要素数は減少し演算量は増加するが、スループットに大きな影響はなかった [図 4.54]。また、2 種類の一時領域世代数パターンでの測定では、一時領域世代数を 3 世代に制限した時とメモリ分散しない 2016 年の版とでは性能に差がないことを確認した [図 4.55]。またバンド分割数 6 の場合、3〜7 世代で性能に有意な差はあらわれないこともわかった。これにより、対角成分を冗長に保持する世代数は 3〜7 程度の世代数で十分であることを確認した。また、保持する世代数の違いに伴う、計算の流れを解析し、メモリ分散前の対角優先では、行方向に計算できれば行方向に進めていくアルゴリズムになっていた。この世代管理の変更により列方向への計算の割合が増えていた [図 4.56]。

LU(コレフスキー) 分解法の GS のメモリ分散化並びに並列性能評価については、本プロジェクト内では実施しなかった。

表 4.81: FGS/評価パラメータ（バンド並列数変化）

	測定条件	
node	41,472	
PROCS	32x48x9x3	16x48x9x6
空間並列数	13,824	6,912
バンド並列数	3	6
ML	544x816x153	
ML0(1 プロセス)	4,913	9,826
MB	110,592	
NBLK	1,536	
ブロック数	72	
node 形状	48×27×32:strict	48x54x16:strict

4.8.3.3　SD のメモリ分散化と大規模問題による性能評価

4.8.3.3.1　2016 年度実施

新アルゴリズムの検討として、SD のカーネルプログラムについて、メモリ分散版を作成して性能評価並びに、外部ライブラリ EigenExa のメモリ分散版の試作を行った。

(1) SD カーネルプログラムのメモリ分散版の試作と性能評価
(2) SD-EigenExa 連携のメモリ分散化の試作

各項目の詳細な内容を以降にて記述する。

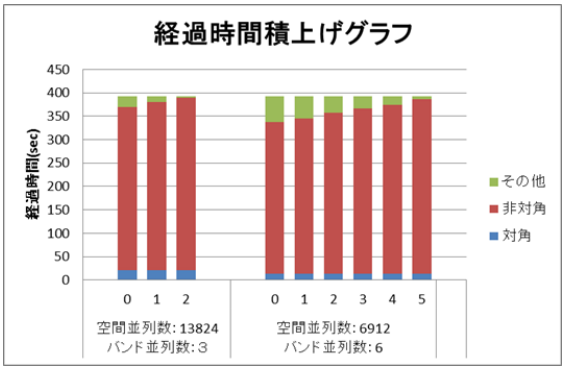


図 4.54: FGS/チューニング性能（バンド並列数変化）

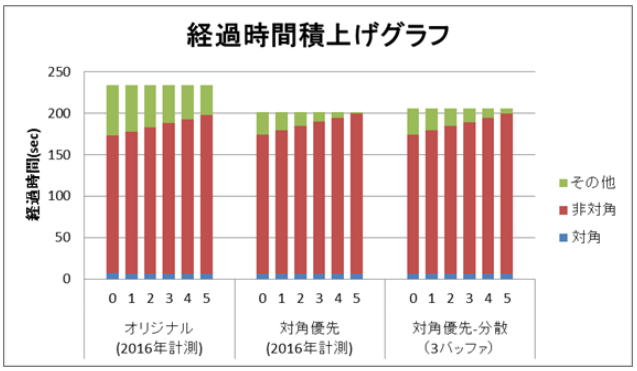


図 4.55: FGS/チューニング性能（メモリ分散による性能の変化）

(1) SD カーネルプログラムのメモリ分散版の試作と性能評価

測定条件は、空間並列数変化、ブロックサイズ変化とバンド並列数変化の 3 種類である。以降では性能評価の結果を説明する。

(a) 空間並列数変化による性能

空間方向の並列数の変化による測定条件を表 4.82 に示す。ターゲット問題に合わせてバンド並列数を 3 とし、1 プロセス当りの処理量を同等となるウィークスケールにより測定を行った。

表 4.82: SD メモリ分散版の性能測定ケース（空間並列数変化）

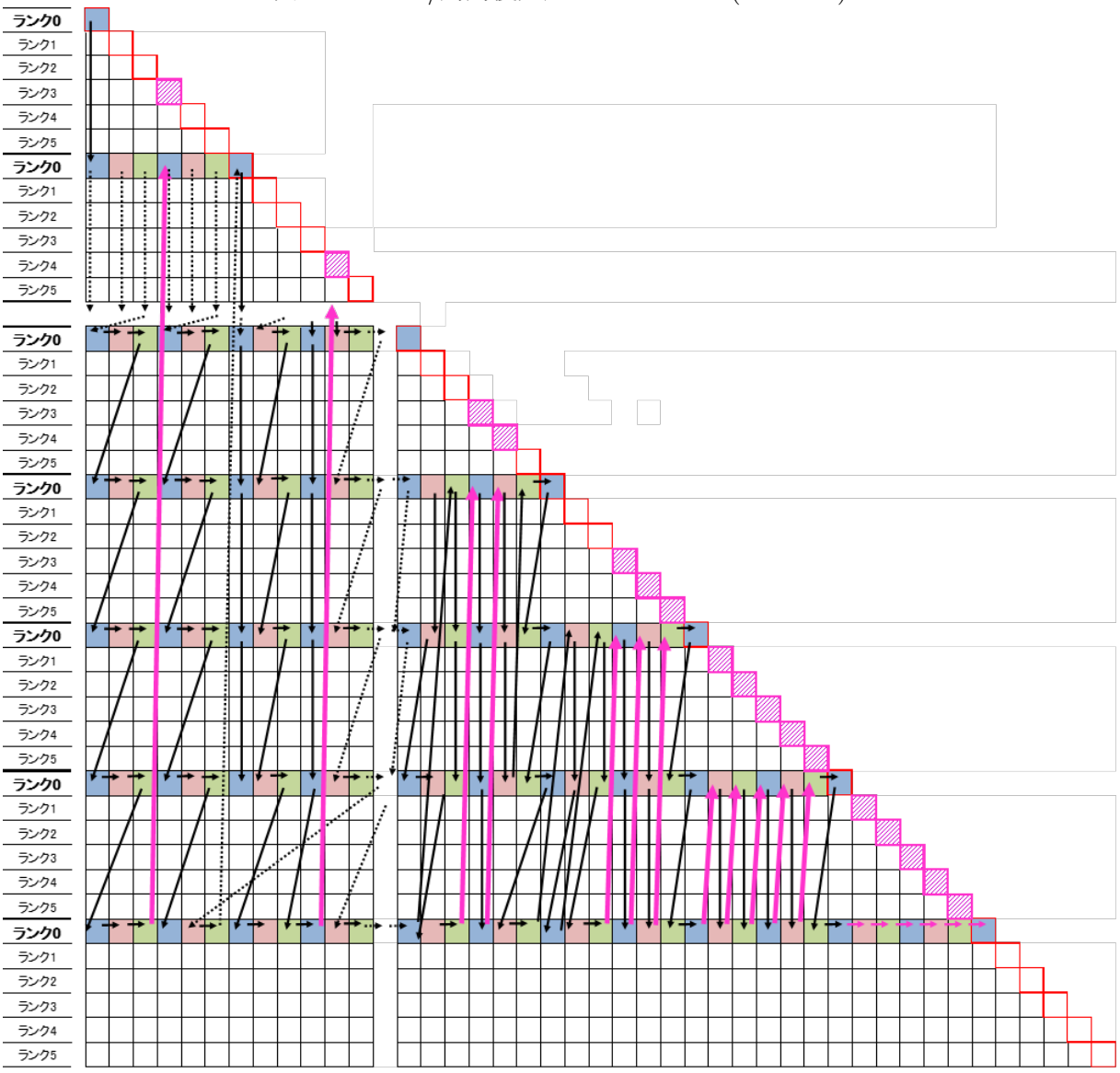
ケース	1-1	1-2	1-3	1-4	1-5
node	1296	2592	5184	10368	20736
PROCS	8 × 6 × 9 × 3	8 × 12 × 9 × 3	16 × 12 × 9 × 3	16 × 24 × 9 × 3	16 × 48 × 9 × 3
空間並列数	432	864	1728	3456	6912
バンド並列数			3		
ML	136 × 102 × 153	136 × 204 × 153	272 × 204 × 153	272 × 408 × 153	272 × 816 × 153
ML0			4,913		
MB			110,592		
NBLK			768		
ブロック数			144		
Node 形状	6 × 27 × 8	12 × 27 × 8	12 × 27 × 16	24 × 27 × 16	48 × 27 × 16

測定結果は、各並列数で同様の実行時間であった。代表してケース 1-1 と 1-5 の測定した結果のグラフを図 4.57 に示す。MatE では、対角部・非対角部の処理に空間プロセス方向の通信処理が含まれるので、空間並列数が増加すると (1-1→1-5)、実行時間が増加する。オリジナル版に対してメモリ分散版は、波動関数転送の通信が増えるので、その分の実行時間が増加するが、演算が主要なので分散前後の実行時間はほぼ同様であり、メモリ分散化による性能低下は小さい。RotV においても、波動関数転送の通信が増えるのでメモリ分散版の実行時間が増加するが、演算が主要なので実行時間はほぼ同様であり、メモリ分散化による性能低下は小さい。

(b) ブロックサイズ変化による性能

ブロックサイズ MBSIZE を、768〜48 まで変化させて測定を行った。測定した結果、NBLK=768 のケースが最速のケースであった。実行時間の主要である MatE の非対角部と、RotV の固有ベクトル転送・演算の時間変化を表 4.83 に示す。

図 4.56: FGS/対角優先処理シーケンス (ランク 0)



ブロックサイズが小さくなると DGEMM の効率が低下するので、演算時間が大きくなり全体の時間が増加する。また、空間方向の通信時間が増加する傾向にある。通信時間は表 4.83 の「通信」と「同期」を合計した値となる。MatE では非対角部にて Reduce 通信を行い、RotV では固有ベクトル転送にて Bcast 通信を行う。通信時間はデータ転送時間とレイテンシ時間の合計である。ブロックサイズが変化しても全体の総通信量は変化しないので、データ転送時間は変化しない。レイテンシ時間は、実行回数に比例するので、ブロックサイズの二乗の逆数に比例する。従って、ブロックサイズが小さくなると、通信時間が急速に増加する。通信時間の式を以下に示す。

$$
\begin{aligned}
\mathrm{t}_{all} &= (\mathrm{t}_L + \mathrm{t}_D) \times \mathrm{N} \\
&= (\mathrm{t}_L + m/\mathrm{B}) \times (\mathrm{M}/m) \\
&= \mathrm{t}_L \times \mathrm{M}/m + \mathrm{M}/\mathrm{B} \quad \propto \quad 1/m \quad \propto \quad 1/MBSIZE^2
\end{aligned}
$$

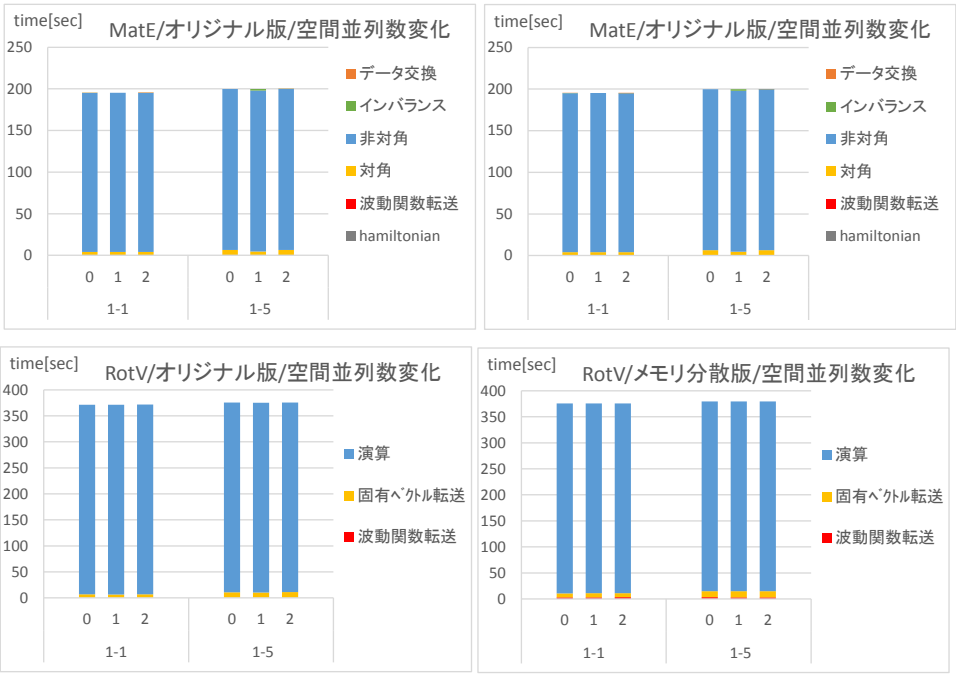


図 4.57: SD/メモリ分散版の性能（空間並列数変化/No.1-1、1-5）

表 4.83: SD/メモリ分散版の性能（ブロックサイズ変化）

	チューニング		2-1(1-1)	2-2	2-3	2-4	2-5
	ブロックサイズ		768	384	192	96	48
	バンドプロセス		0	0	0	0	0
MatE	全体		204.49	217.14	246.35	339.64	854.04
	非対角		191.63	205.35	235.01	327.66	839.25
		演算	185.41	195.33	213.32	281.41	709.34
		同期	2.18	4.24	11.13	21.36	69.02
[単位:秒]		通信	4.04	5.78	10.56	24.90	60.89
RotV	全体		376.27	392.26	455.33	613.72	1520.92
	固有ﾍﾞｸﾄﾙ転送		7.29	11.91	34.21	82.89	259.98
		同期	2.89	4.82	18.16	41.65	155.39
		通信	4.40	7.09	16.05	41.24	104.59
[単位:秒]	演算		365.30	376.10	415.65	520.53	1237.27

t_{all} ；総通信時間
N ；実行回数（=M/m）
M ；総メッセージサイズ
B ；通信バンド幅

t_L ；レイテンシ時間 (1 回)
t_D ；データ転送時間 (1 回)
m ；メッセージサイズ (1 回)
$(m = MBSIZE^2 \times 8byte)$

(c) バンド並列数変化による性能

バンド方向の並列数の変化による測定条件を表 4.84 に示す。ケース 1-1 に対して、バンド並列数を 1、2、3、4、6 と変化させたストロングスケールを測定した。

測定した結果、並列数の増加と共に実行時間が短縮した。ケース 3-3(1-1) と 3-5 の測定した結果のグラフを図 4.58 に示す。MatE のオリジナル版では、ブロック分割により各バンドランクへ割り当てるため、偶数分割の場合に非対角項の処理量が異なり、インバランスが発生している。メモリ分散版では、ブロックサイクリックにより各バンドのランクへ割り当てるため、インバランスがほぼ無く、ケース 3-5 では実行時間が改善

表 4.84: SD メモリ分散版の性能測定ケース（バンド並列数変化）

ケース	3-1	3-2	3-3(1-1)	3-4	3-5
node	432	864	1296	1728	2592
PROCS	$8 \times 6 \times 9 \times 1$	$8 \times 6 \times 9 \times 2$	$8 \times 6 \times 9 \times 3$	$8 \times 6 \times 9 \times 4$	$8 \times 6 \times 9 \times 6$
空間並列数			432		
バンド並列数	1	2	3	4	6
ML			$68 \times 102 \times 153$		
ML0			4913		
MB			110592		
NBLK			768		
ブロック数			144		
Node 形状	$6 \times 9 \times 8$	$6 \times 18 \times 8$	$6 \times 27 \times 8$	$6 \times 36 \times 8$	$18 \times 18 \times 8$

されている。RotV では、フル行列の処理を行うためインバランスは無く、波動関数転送の時間が増加する結果ではあるが、メモリ分散の有無で同様の実行時間である。

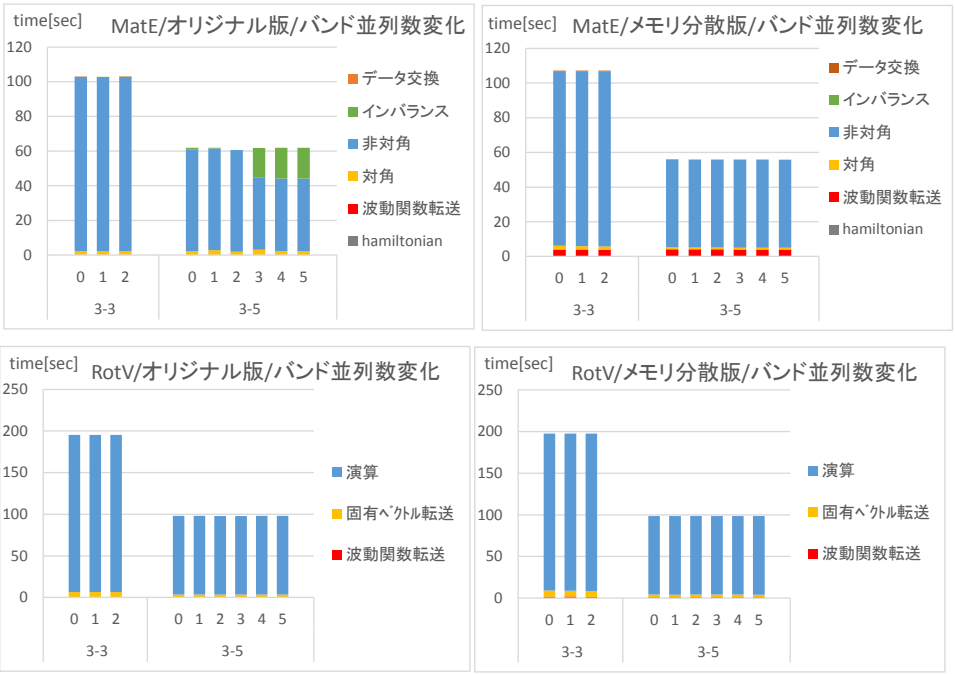


図 4.58: SD メモリ分散版の性能（バンド並列数変化/No.3-3,3-5）

また、バンド並列数が 1 に対する実行時間の短縮比率を並列効果として算出した結果を表 4.85 に示す。バンド並列についても並列効果が高いことがわかる。

(2) SD-EigenExa 連携のメモリ分散化の試作

SD のカーネルプログラムのメモリ分散版に対応して、EigenExa と連携する部分の試作を行った。EigenExa ではフル行列を扱うので、三角行列を対称行列化して、EigenExa の前後でブロックサイクリックとサイクリックのデータの持ち替えを行うよう、連携部分を作成した。EigenExa は三重対角処理による eigen_s を使い、小規模データで動作を確認した。

SD の MatE 処理では三角行列を扱い、EigenExa 連携でフル行列化を行うようにしているが、今後、MatE 処理の際にフル行列として処理することで、フル行列化の処理が不要になるので、三角行列版とフル行列版で性能を評価した。

表 4.85: SD メモリ分散版/並列効果（バンド並列数変化）

バンド並列数		1	2	3	4	6
MatE	オリジナル	1.00	1.33	2.99	3.19	4.97
	メモリ分散	1.00	1.96	2.89	3.76	5.54
RotV	オリジナル	1.00	2.00	3.00	3.99	5.98
	メモリ分散	1.00	1.98	2.96	3.92	5.94

4.8.3.3.2　2017 年度実施

新アルゴリズムの検討として、SD のカーネルプログラムについて、外部対角化ライブラリ EigenExa の連携について性能評価を行った。

(1) 外部対角化ライブラリ EigenExa の性能評価
(2) SD-EigenExa 連携のメモリ分散化の性能評価
(3) 並列数の増加に伴う eigen_scaling の実行時間の増大に関する調査
(4) EigenExa 内の bcast 通信集約の効果に関する評価

各項目の詳細な内容を以降にて記述する。

(1) 外部対角化ライブラリ EigenExa の性能評価
詳細設計 (2) では、メモリ分散化された対角化のカーネルのうち MatE、RotV の性能評価を行い、外部対角化ライブラリ EigenExa と連携する部分の試作を行った。SD-EigenExa 連携の性能を評価するにあたり、参照性能として EigenExa ライブラリの単体の基礎性能の評価を行った。評価時の EigenExa ライブラリの最新バージョンは 2.4 であるが、RSDFT と互換となっていないため RSDFT の連携結果との比較のためバージョン 2.3k2 を用いて測定を行った。図 4.59 に EigenExa-2.3k2 のスケーラビリティを示す。測定にはターゲット問題の半分の規模 110,592 元での測定を行っている。左図が EigenExa 内の 3 つのモジュールごとのスケーラビリティで、右図が通信と演算に分けたスケーラビリティである。評価区間のうち分割統治法（DC）については ScaLAPACK の通信ルーチンを呼び出しており、厳密に演算と通信を切り出しが出来ていない。演算のスケーラビリティは概ね良好であり、ターゲット問題の半分の規模の行列サイズで、想定並列数 $96 \times 96 = 9,216$ プロセスまで実行時間の短縮が期待できることが分かる。

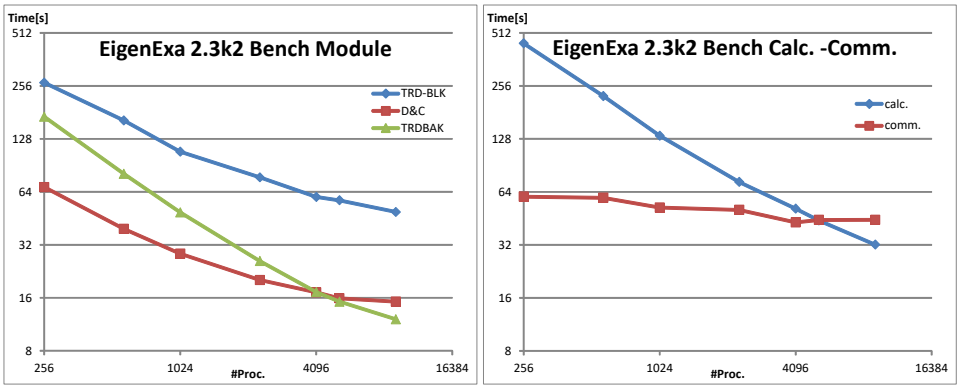


図 4.59: EigenExa のスケーラビリティ。（左図：区間ごと、右図：演算と通信）

図 4.60 に京での性能値並びに富岳での性能推定の根拠として使われた 2011 年の Gordon Bell

賞受賞時に使われたバージョン (Ver. 2012) の性能と評価バージョンの性能比較を示す。計測可能な全てのプロセス数でも通信時間が大幅に削減されていることがわかった。この通信時間の削減は、三重対角化ルーチン（TRD-BLK）の通信の集約によるレイテンシの隠蔽によるもので、ターゲット問題を京で測定した結果では、最新のライブラリで 1SCF あたり 90.5[s] 削減されることが期待できる。

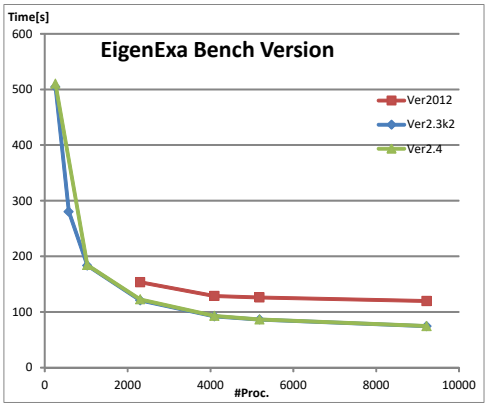


図 4.60: EigenExa のバージョンによる性能の違い。

また SD 連携では、EigenExa ライブラリで使用するプロセスを縮小することで実行時間短縮の実現を考えており、その時、EigenExa のプロセス形状が必ずしも正方形ではない可能性がある。図 4.61 に長方形のプロセス配置における EigenExa の性能を記す。測定には、ターゲット問題 221,184 の時と同程度の演算量となるように半分の問題規模 110,592 元で見積もられたプロセス数、576 にていくつかのプロセス形状パターンで測定を行った。但し、プロセス配置の扁平率が全体的に高いため EigenExa ライブラリで想定されていないのプロセス形状となり、動作が完了しないパターンも存在するので注意が必要である。これを見ると EigenExa を高速に動かすためには通信を最適化するために、なるべく正方形に近いプロセス配置にする必要がある。しかし長方形の扁平率による速度低下は大きくはないことが分かる。ライブラリを呼び出す際のランク配置の工夫をすることでより、5%程度ではあるが効率的な利用が可能になることが分かった。

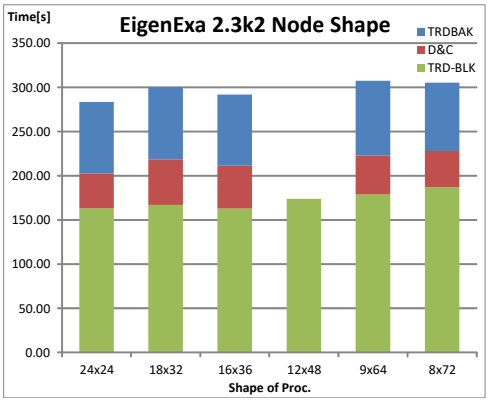


図 4.61: EigenExa のプロセス配置による性能の違い。

(2) SD-EigenExa 連携のメモリ分散化の性能評価

2016 年度試作した SD-EigenExa 連携のメモリ分散試作版に、EigenExa ライブラリを組み

込んで動作テストした。様々な問題でテスト行ったところ、処理量のインバランス、変数のオーバーフロー、初期化の時間増大、利用されない 17.6GB のメモリアロケーション、プロセス収縮率が高い時の考慮漏れなどの問題が見つかり、測定区間の整備と並行して修正を行った。また EigenExa ライブラリの最新版を組み込んだところ、最新版の EigenExa2.4 ではライブラリのモジュール構造の変更などがあり、確実な性能評価が困難であるため、まずは同性能であるとされる EigenExa2.3k2 で実行を行った。バージョン 2.3k2 版ではいくつかの分割パターンにおいてライブラリが正常終了しない場合があり、問題を数値ライブラリ WG にフィードバックすることで、障害を修正した 2.3l 版が提供された。

SD-Eigen 試作コードを用いて大規模問題の性能測定を実施した。プロセス配置には、RSDFT のプロセス数と EigenExa のプロセス数の 2 種類のパラメータが考えられるため、評価に用いた問題規模として、

(a) 1,296〜20,736 並列 5 パターン→ 1024 並列への縮約。

(b) 10,384 並列→ 256〜9,216 並列 6 パターンへの縮約。

のそれぞれについて RSDFT 空間方向の分割数を変化させた時の、3 つのモジュールパターンの性能を測定した。測定結果を図 4.62 に示す。左図が RSDFT の分割パターンを変化させた時の EigenExa 周辺の性能で、右図は EigenExa の分割パターンを変化させた時の EigenExa の性能である。

Eigen 連携のうち三角行列を処理するモジュール（imate=1）において、32×32 へ縮約するパターンで、性能ぶれが確認されたが、演算時間のぶれが通信同期待ちとして観測されたものと評価された。本区間は、2 回実行することで性能ぶれが減少したため、DGEMM の 1 回目呼び出しのオーバヘッドによるものと想定できる。性能評価には注意が必要である。

2 万並列から各分割数へ縮約させた時の性能を測定では、4,096 並列が一番速かった。これは前項の EigenExa の基礎性能と照らしあわせて、プロセス縮約の影響ではないことが想定される。SD 連携と EigenExa 参照測定データとの性能比較を行ったところ、以下のオーバーヘッドが見られた。

(a) EigenExa72×72 分割：三重対角化 Bcast

(b) EigenExa96×96 分割：三重対角化に入る前の eigen_scaling

この処理は、行列要素を規格化するための max 関数を計算しており、終了処理の DSCAL は変化なかった。

なおサイクリック化とブロックサイクリック化などの行列の前後変換処理は対角化本体処理と比べて比率が小さいことが分かった。また現在、ScaLAPACK との性能比較を試みているが、ScaLAPACK が想定した時間で終了しないため、比較方法を検討した。

(3) 並列数の増加に伴う eigen_scaling の実行時間の増大に関する調査
詳細設計 (3) 開発期間 II では、前項で見つかった並列数の増加に伴う eigen_scaling の実行時間の増大について、調査を実施した。eigen_scaling は max 関数を使用して行列要素の規格化の処理をしており、このオーバーヘッドについて切り出しを行い調査する予定であったが、第 1 段階としてインバランスの状況の解析を行った。詳細設計 (3) 開発期間 I での測定結果をランク番号ごとに再度解析したところ、オーバーヘッドが見られた 96x96 並列での eigen_scaling の実行時間はプロセスごとに 20[s]〜70[s] のばらつきがあった。eigen_scaling の

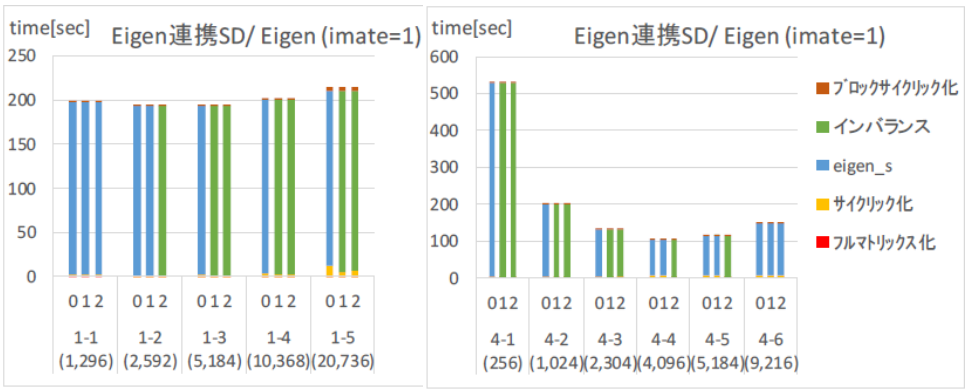


図 4.62: SD-EigenExa 連携の性能。左図：RSDFT の分割数を変化。右図：EigenExa の分割数を変化。

時間増大は、その直前に実行される、ブロックサイクリック分割からサイクリック分割へ並列方式の変換を行う RSDFT 内のモジュール trans_blkcy2cy のインバランスによるものであった (図 4.63)。このモジュールは、blacs の pdgemr2d を呼び出しており、eigen_scaling の規格化の処理における、1 要素の allreduce 通信によるこのバリア同期によって、pdgemr2d のインバランスが計上されたと想定される。今後、Eigen 連携のチューニングについて、EigenExa 内のサービスルーチンの導入について調査並びに評価を行った。

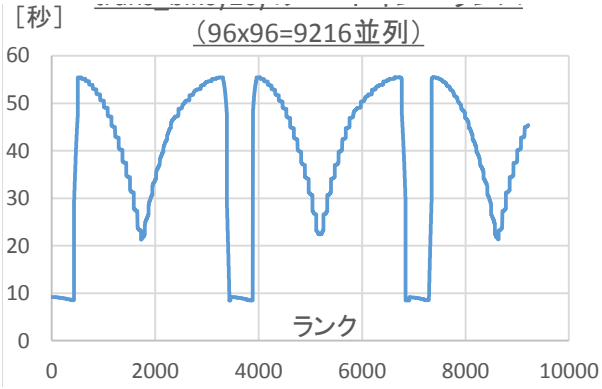


図 4.63: trans_blkcy2cy のロードインバランス

(4) EigenExa 内の bcast 通信集約の効果に関する評価
この評価は、最新版の EigenExa ライブラリの三重対角化 (TRD-BLK) の区間について、京での通信時間が通信式による見積もりと比べて、通信の集約による効果が、大きくなかったため実施した。Eigen 連携では、通信時間削減を目的に、EigenExa を 1proc/1node で実行することを想定しており、通信サイズも小さい特徴がある。今まで bcast の通信性能は、Trinaryx, Simple Spread(図 4.64) を用いた通信性能見積もり式による評価結果を行ってきた。Tofu 専用アルゴリズムが適用できない条件での通信など上記以外のアルゴリズムの性能は、経路の輻輳があるため、見積式による評価は難しい。今後、実際に適用されると推定される Bintree3d(図 4.65) による再評価を行なった。

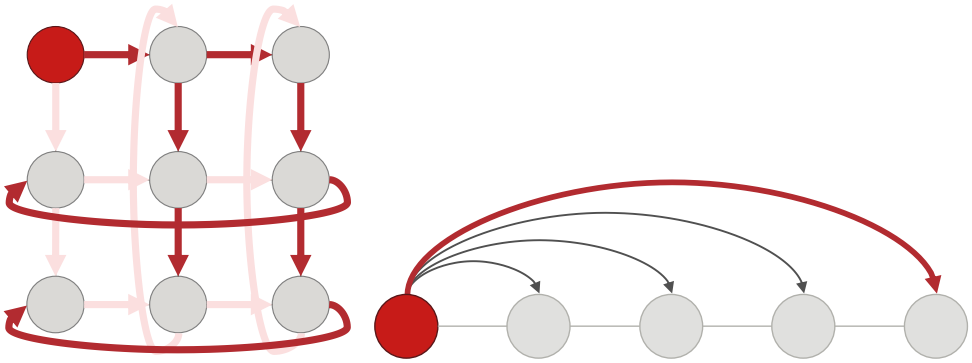


図 4.64: 今まで評価を行っていた Tofu 専用アルゴリズムの通信概要：左図：二次元 Trinaryx3、右図：一次元 Simple Spread

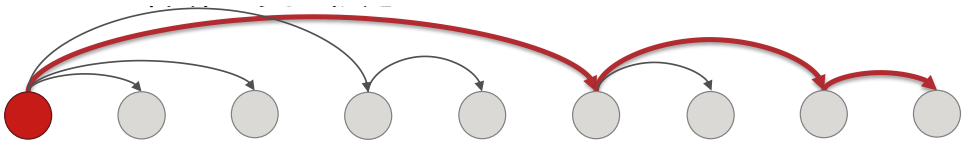


図 4.65: 一次元 Bintree3D アルゴリズムの通信の概要

4.8.3.3.3　2018 年度実施

詳細設計 (3) 開発期間 II で見つかった、EigenExa 大規模並列における、RSDFT バンド空間分割 (cyclic) から ScaLAPACK 分割 (block cyclic) の変換する pdgemr2d のオーバーヘッド増大の問題について、詳細設計 (4) では、EigenExa のサービスルーチン (dc_redist1/2) の組み込みの可能性について評価を行った。

EigenExa 内で提供されている dc_regist は、2 段階で block cyclic → cyclic の変換を行い、行と列を別々に変換する仕組みとなっていた [図 4.66]。RSDFT の対角化では使用するプロセス数の縮小も行っているため、単純に行と列方向に処理を分離することは出来ないことが分かった。

RSDFT 内のメモリ分散は、block cyclic は MatE や RotV などでの処理が高速になるように分散配置しており、cyclic は ScaLAPACK の処理が高速になるように分散配置したものとなっている。pdgemr2d の高速化を考えるのではなく、プロセスマッピングを調整して、MatE、RotV、EigenExa、プロセスマッピング変換の 4 つがバランスを保ちつつ高速になるように調整した方が良いという結論に達した。

pdgemr2d により増大する時間は、アプリ全体で 1,000 秒ほどのうちの数十秒である。EigenExa のプロセス分割を 64×64 で分割した時は、pdgemr2d 起因の実行時間の増大はなかった、EigenExa を 96×96 分割で動かす必要がないのであれば、最適なプロセスマッピングを再度検討する必要がある。また分割規模縮小については、CPMD、LU 分解、FFT などで外部ライブラリを用いる箇所で同じことが言え、同様の対応が必要と想定される問題と考える。EigenExa の Mapfile を用いたプロセスマッピングチューニングは大規模実機評価が始まってから実施した。

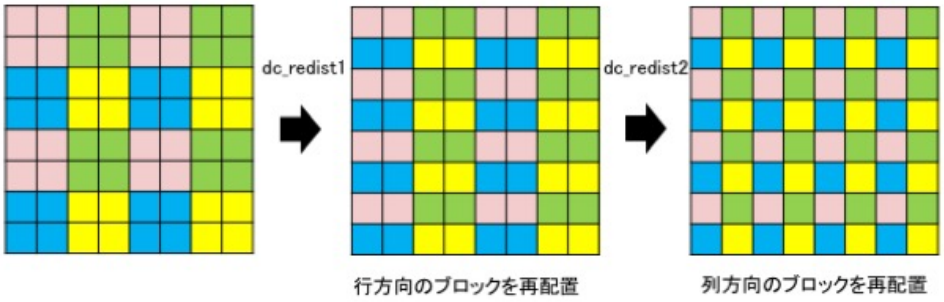


図 4.66: dc_regist の処理イメージ

4.8.3.4　CG のメモリ分散化

4.8.3.4.1　2017 年度実施

新アルゴリズムの検討として、CG のカーネルプログラムについて、メモリ分散版の試作並びにチューニングに向けた評価を行った。

(1) CG カーネルプログラムのメモリ分散化の試作
(2) CG カーネルプログラムの性能評価

以下に、実施内容の詳細を記述する。

(1) CG カーネルプログラムのメモリ分散化の試作
CG の演算量は、$O(N^2)$ であるため全体に占める割合は SD や TGS などの $O(N^3)$ 区間と比べて相対的に大きくない。このため、2012 年に京で RSDFT の性能を評価した際には、チューニングの対象としていなかった。全体のメモリ使用量の観点から CG のカーネルについてもメモリ分散化が必要であり、詳細設計 (3) では、CG の区間についてメモリ分散版の試作並びに、今後のチューニングの見積もりを行った。処理内容は CG 本体処理とハミルトニアンを構成する前処理から構成されている。今回の作業では、前処理について、運動エネルギー項の構成に限定し、オプションとしては、preconditioning_1 の処理についてメモリ分散の試作を行った。CG 処理では通信が全体の 73%を占め、袖通信と空間方向の Allreduce 通信の削減が課題となる。これらの処理は通信処理バンド数回行われており、通信を行う BCSET ルーチン内で行われる通信の集約により、レイテンシの隠蔽が可能である。

(2) CG カーネルプログラムの性能評価
詳細設計 (3) 開発期間 II では、開発期間 I で試作したメモリ分散版の性能評価を実施した。大規模問題での性能測定に際しては、初期状態生成のための時間が増大することが想定されたためソースコードの修正を行い、標準で出力されるタイマ情報に詳細タイマの追加を行った。評価パターンとしては、SD での評価に用いた測定パターン (表 4.82, 表 4.84) に加えて、バンド分割数を 2 倍に増やしたものを追加した。BCSET 内の通信集約のパラメータ MB_d は、2011 年当時は 3 で測定していた。これは、メモリ使用量の問題で大きいパターンを試すことができなかったためであるが、メモリ分散の実装により、チューニングの効果が期待できるため、MB_d=3,9,18 を 3 パターンについて性能評価を行った。hamiltonian 計算については、非局所ポテンシャル項の計算は通信が主であり負荷は小さいため、運動項（差分）の計算のみを用いて性能評価を実施した。評価結果は図 4.67〜図 4.69 の通りである。空間方向分割の weak scaling 測定では、メモリ分散の有無による性能差はなく、メモリ分散化に起

因するオーバーヘッドは小さいと評価できる (図 4.67)。しかし並列数の増大とともに、メモリ分散の有無にかかわらず、preconditioning モジュールの演算時間、通信時間が供に増大しており、それに伴う演算量の増加は見られなかった。バンド分割の strong scaling 測定では、初めて測定した分割数 12 の場合、性能が 10.8 倍向上し、並列性は良好といえる (図 4.68)。また通信の集約サイズ MB_d は、3 → 18 に増やしたところ、180[s] → 80[s] まで実行時間の短縮が見られた (図 4.69)。バンドベクトルがまとめられるため、通信レイテンシが削減できたためと推察されるが、今回の測定では、演算時間も減っており、詳細の解析・評価については、全体への統合を行った後の全体性能評価チューニングの段階で実施する。

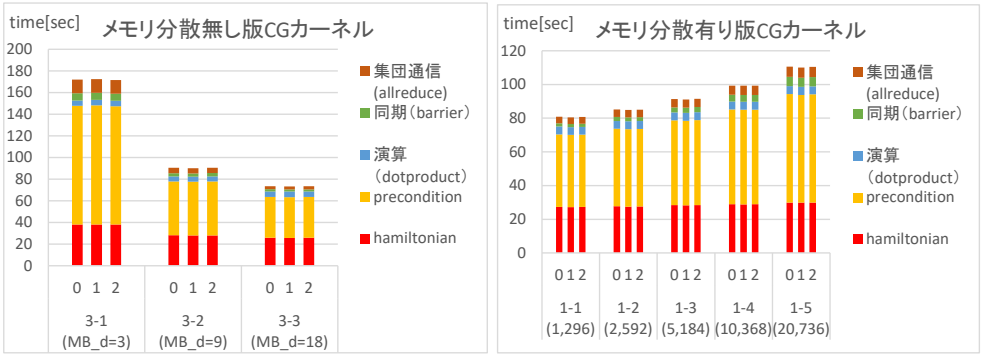


図 4.67: CG の処理別経過時間（空間並列数変化）左図：メモリ分散無し、右図：メモリ分散有り

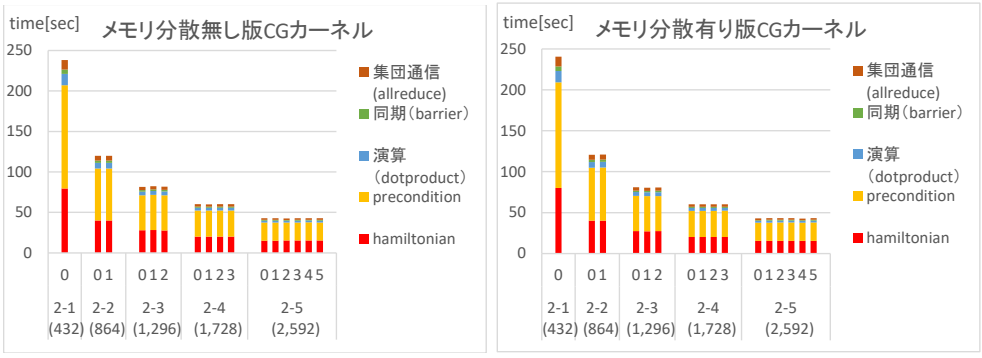


図 4.68: CG の処理別経過時間（バンド並列数変化）左図：メモリ分散無し、右図：メモリ分散有り

4.8.3.5　CPMD のメモリ分散化

新アルゴリズムの検討として、ターゲット問題の 1 つとしている CPMD カーネルプログラムについて、SCF 計算と同様に性能評価を進めている。ただし、性能・電力の予測は、従来の SCF 計算で行っている。性能評価を進めるにあたり、現在実装されている CPMD の性能の見積もりが必要であると供に、SCF の各区間同様にメモリ分散化が必要である。

4.8.3.5.1　2015 年度実施

CPMD 計算に対して、カーネルを抽出し性能評価の準備を行った。以下に、実施内容の詳細を記述する。

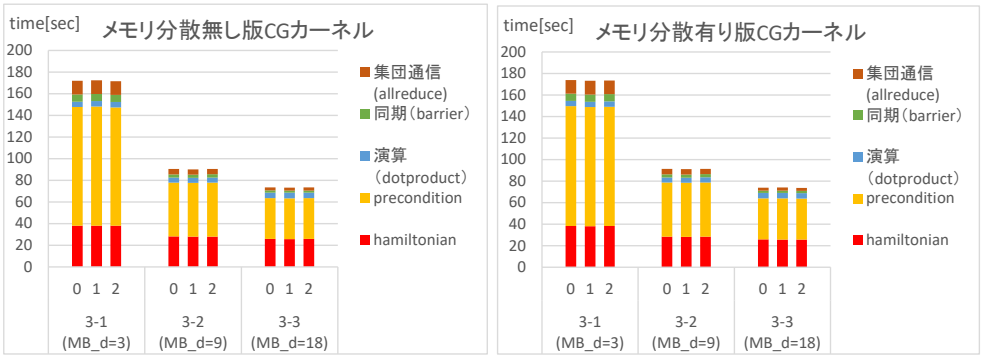


図 4.69: CG の処理別経過時間（バンドまとめ数変化）左図：メモリ分散無し、右図：メモリ分散有り

(1) 内容
CPMD 計算について、演算区間と通信区間を特定して、SCF 計算と同様に、演算区間のカーネル化と通信情報の特定を進めた。演算区間は DGEMM と DSYRK が主要となり、4 本のカーネルプログラムを作成して、富岳推定を進めた。

(2) 状況
SCF 計算において推定した DGEMM に加えて DSYRK の推定が必要となり、カーネルプログラムを用いて、FX100 測定・富岳推定を行い性能を評価した。DSYRK は、京の演算ピーク比に対して富岳の推定演算ピーク比が低い傾向であり、性能改善の確認を行なった。%していく予定である。

4.8.3.5.2　2016 年度実施

2015 年度に切り出した CPMD 計算のカーネルを用いて、通信性能の評価並びにメモリ分散に関する検討を行った。以下に、実施内容の詳細を記述する。

(1) 内容
CPMD 計算の通信区間について、SCF 計算と同様に性能を推定して、性能の傾向を確認した。

(2) 状況
通信性能について、京と富岳で違いが大きい区間があり、それぞれの見積式と算出結果を確認して、京の見積値が大きい傾向であることが判明した。ただし、今後のメモリ分散の適用により通信部分が大きく変更される可能性があり、メモリ分散が適用される際に再度検討することとした。

4.8.3.5.3　2017 年度実施

CPMD カーネルプログラムについて、メモリ分散版の試作並びにチューニングに向けた評価を行った。実施内容は以下の通りである。

(1) CPMD 計算で用いられる DSYRK のチューニング
(2) CPMD カーネルプログラムのメモリ分散化の調査
(3) CPMD カーネルプログラムのメモリ分散化の試作

以下に、実施内容の詳細を記述する。

(1) CPMD カーネルプログラムで用いられる DSYRK のチューニング
CPMD 中の DSYRK の性能について、上三角行列のサイクリック分割の最適化を実施し、性能の評価を行った。その結果、FX100 で効率が 57%→ 88%と 30%程度性能が向上すると評価された。今回の富岳での DSYRK の予測性能は DGEMM 専用モードによる推定ではなく、性能電力予測ツールによる評価であり、効率は 67%であった。

(2) CPMD カーネルプログラムのメモリ分散化の調査
CPMD カーネルプログラムのメモリ分散化の対象は、ROMatE, ROsolv, RORotV から構成される ROTORB である。これらの処理は、SD カーネルプログラムの MatE, Solv, RotV と同等の処理であり (図 4.86)、SD カーネルプログラムでのメモリ分散方法のノウハウをそのまま使用することが可能であった。CPMD カーネルプログラムのメモリ分散前の実装では、全バンドサイズ MBCxMBC サイズの Work 配列が 6 つ存在し、このメモリ分散化されていない配列のメモリ分散が課題である。CPMD 内の力の計算区間 getforce_cpmd 内にある、ハミルトニアン演算ルーチン (Hamiltonian) については、非局所ポテンシャル項の計算の中で、自プロセスの作用＋データ交換、htpsi の更新の処理を行っており、リストアクセスであり、何回も呼ばれるため負荷が高いと見積もった。

表 4.86: Subspace-diag と ROTORB の処理の比較

Subspace-diag/SCF	ROTORB/CPMD
MatE　行列生成	ROMatE　行列生成
Ψ(ML,MB)　→　Hsub(LLD,LLC)	Ψ(ML,MBC) 、Ψ(ML,MBC) →　A(MBC,MBC), B(MBC,MBC)
Solv　求解（固有ベクトル、固有値）	ROSolv 求解（ラグランジュ未定乗数法）
Hsub(LLD,LLC) →　Vsub(LLD,LLC)	A(MBC,MBC), B(MBC,MBC) →　χ (MBC,MBC)
Rotv 波動関数の更新（回転）	RORotv 波動関数の更新（回転）
Vsub(LLD,LLC)　→　Ψ(ML,MB)	χ (MBC,MBC)　→　Ψ(ML,MBC)

(3) CPMD カーネルプログラムのメモリ分散化の試作
CPMD カーネルプログラムについて、テストプログラムの切り出しを行い、CPMD メモリ分散版のテストプログラムを試作、演算量削減のチューニングとテスト実行を行った。プロセス分割は ScaLAPACK と同様な方式とし、DGEMM を PDGEMM へ置き換えを行った。コードのメンテナンスを鑑み、今回の分散方法は妥当であり、今後 PDGEMM ライブラリの高速化を推進したい。試作性能を図 4.87 に示す。

実行時間は、ブロックサイズを 36 → 72 へと大きくすることで、オリジナル版より高速になった。性能はブロックサイズのチューニング効果が大きく、非対角ブロックコピーによるチューニングの効果は小さかった。オリジナル版ではサイズの大きい DGEMM を行っていたため、メモリ分散版ではブロックサイズが小さい場合に遅くなったためと考えられる。またオリジナル版で行っていた allgatherv でのデータ持ち替えがなくなったことにより、並列数が大きい時に、通信の削減効果が期待できる。

詳細の性能評価については、2018 年度以降の統合版の評価の一環で実施した。メモリ使用

量は、CPMD で SCF の約 3 倍、バンド×バンドの行列を 6 配列確保するため、ターゲット問題を設定した上で、メモリ使用量の削減効果や通信時間の変化の評価を行った。

CPMD 内では、力の計算をする際に、今回試作対象ではない FFT が 3 回計算されており、ffte を使用している。筑波大で FFT の使用方法について議論されており、FFT 性能についても評価並びにプロセスマッピングを含むチューニングが必要である。

表 4.87: ROTORB の実行時間の測定結果：単位 [s]

プログラム種別		ベース	メモリ分散	メモリ分散	メモリ分散	メモリ分散
ブロックサイズ		-	36	36	72	72
非対角ブロックコピー		-	×	○	×	○
rotorb		0.81672	1.23073	0.98339	0.74418	0.69336
ROMatE	[1]	0.39421	0.44683	0.32232	0.28073	0.25569
	[2]	0.35397	0.44536	0.32302	0.28061	0.25357
ROSolv	[3]	0.00090	0.00468	0.00410	0.00352	0.00399
	[4]	0.01108	0.03180	0.03267	0.02331	0.02287
RORotV	[5]	0.03455	0.26978	0.26891	0.12394	0.12530

4.8.3.5.4　2018 年度実施

詳細設計 (4) では、CPMD のバンド方向メモリ分散化を進めた。詳細設計 (3) にメモリ分散化を実施した ROTORB カーネルに続き、その他の区間について詳細設計 (4) の前半で公開された統合版 (RS-CPMD) を元にバンド方向メモリ分散化を実施した。

CPMD では、SCF などと比べてメモリ使用量が大きい特徴がある。これは band× band サイズの一時配列による影響であり、空間方向のメモリ分散だけでは、メモリ使用量の削減はできない。このためこれまでは、ターゲット問題の設定や性能の見積りさえ難しい状況であった。

これに対して、CPMD 内並びにグローバルに確保されている配列を詳しく再精査し、メモリ使用量の詳細な評価を行った。この調査結果をもとに Excel ベースの CPMD メモリ使用量簡易シミュレータを作成し [図 4.70]、ターゲット問題の設定に利用した。CPMD メモリ分散評価のターゲットとして、SiO_2 の 12,288 原子〜41,472 原子のものを作成し、問題規模の評価を行い [図 4.71]、各問題について、グリッド数などのパラメータリストを算出した。また SiC/SiO_2 界面問題について、13,344 原子〜120,096 原子の問題を作成した [図 4.72]。

また詳細設計 (4) には、カーネル単位ではなく詳細設計 (3) に行われた統合作業で整備されたフルアプリ RS-CPMD をベースにバンド方向メモリ分散を実施した。このため詳細設計 (3) にメモリ分散の試作済みの ROTORB カーネルについても、変更内容の確認を実施した。

CPMD のバンド方向メモリ分散をアプリケーション全体を通して動作させるためには、ROTORB 以外のカーネルについてもメモリ分散化が必要である。CPMD の処理について、全処理に対応するカーネルを抽出し、CPMD 内の各処理についてメモリ分散作業の見積りを実施した。対象となるカーネルは、wf_force, getforce_cpmd, rotorb2 であり、それらの特徴は以下の通りである。

1. getforce_cpmd: 畳み込み計算を目的とした FFT 処理が含まれる。
2. ROTORB2: ROTOB の実装とほぼ同じである。
3. wf_force: 1 つのループから構成されるので、メモリ分散は難しくない。
4. getforce_cpmd: バンド方向のループを持たないためメモリ分散は必要無い。

また、CPMD 内の nonlocal 項（リストアクセス）と差分の計算（H ψ：行列ベクトル積計算）については、SCF でも同じ計算が行われているため、そちらでの動作チェックが済んでいれば新たな作業は発生しない。

これらのメモリ分散の試作を行う過程で、いくつかの問題が見つかった。calc_density2/getforce_cpmd の占有バンド配列 OCC と波動関数 unk の情報が一致しないという問題が見つかり、データの並べ替えによる block cyclic の分割パターンを修正することで対応した。また空間並列数を変化させた時に結果が一致しない問題があった。これは SCF ブロック内の処理の問題と見られ、CPMD カーネルのメモリ分散化によるものではないことを確認し、開発者にフィードバックを行った。これにより、全体を通してメモリ分散で動作する RS-CPMD が完成した。

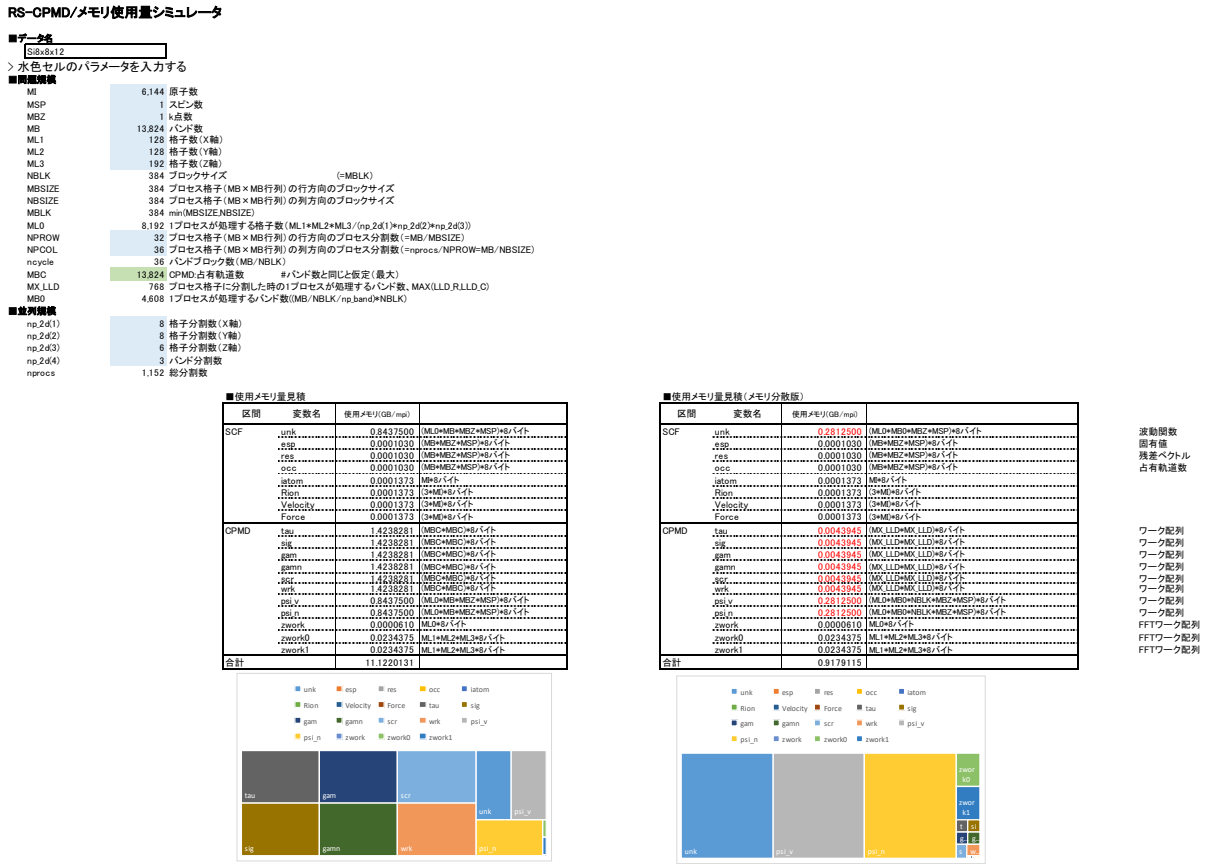


図 4.70: Excel ベースの CPMD メモリ使用量簡易シミュレータ

詳細設計 (4) 後半では、詳細設計 (3) 開発期間 II にメモリ分散化を実施した CPMD の ROTORB カーネルについて、大規模性能評価を行った。ターゲットとする問題は、開発者が作成した SiO_2 12,288 原子〜41,472 原子を用い、並列数とそのメモリ使用量いついて評価を行った。4 万原子規模では、5 万並列以上でないと動作しないことが判明したため、12,288 原子を用いた性能評価を行った。CPMD の処理は、処理ブロックごとに分割パターンが変わるため、妥当性の検証のため、バンド方向に weak scaling、空間方向に strong scaling による測定パラメータ案を作成した。

864node を用いたメモリ分散版と非メモリ分散版との比較では、[表 4.88] の通り 2 種類のブロックサイズ MBLK を用いて実行時間の内訳を測定した。ROMatE、ROSolv は、ほぼ同等の性能であることを確認した。RORotv が 2 倍ほど時間が増大したが、問題サイズに対応したブロックサイズ MBLK を 180 に調整した測定では、gemm のサイズが大きくなったため性能劣化が抑えられた [図 4.73]。バンド数を固定した時の weak scaling 測定では、[表 4.89] のパラメータを使った。ROMatE、RORotv で weak scale し性能劣化は見られなかったが、ROSolv は若干ではあるが増大した [図 4.74]。ROsolv が遅くならなかったのは、pdgemm の処理が増えている可能性はあるも

supercell	3	4	8	10	12	14	16	
atoms	648	1536	12288	24000	41472	65856	98304	
electrons	3456	8192	65536	128000	221184	351232	524288	
band	1728	4096	32768	64000	110592	175616	262144	CPMDで必要なバンド
band_x1.1	1901	4506	36045	70400	121651	193178	288358	
band_x1.2	2074	4915	39322	76800	132710	210739	314573	SCF計算用バンド（1.1倍～1.5倍）
band_x1.5	2592	6144	49152	96000	165888	263424	393216	
grid_1D	108	144	288	360	432	504	576	1方向のグリッド刻み数
grid_3D	1259712	2985984	23887872	46656000	80621568	128024064	191102976	全グリッド数

この辺がターゲット

図 4.71: CPMD のターゲット問題 (SiO_2) とその動作パラメータ

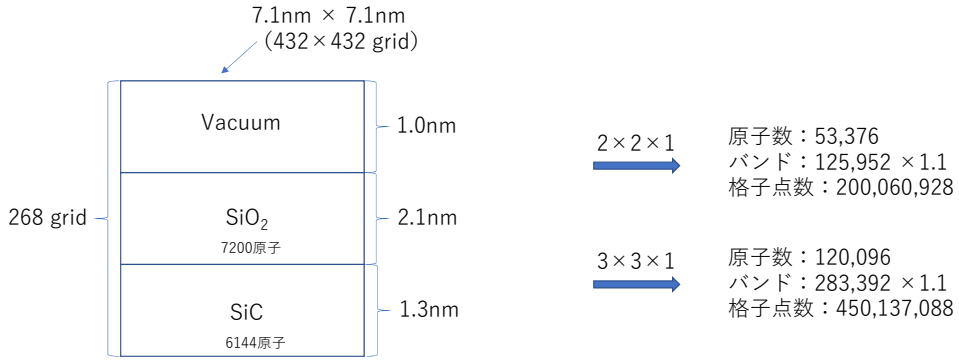


図 4.72: CPMD のターゲット問題 (SiC-SiO_2 界面問題)

のの、allgatherv がなくなった分速くなったと解釈できる。また測定パラメータ 3-5 で実行時間が増大しているのは、ノード当たりの演算量が端数処理分、他の測定ケースより多くなっているためであり、weak scaling 性能上は問題ではない。RORotv は大きな gemmm 処理であり、限界性能で動作している部分である。ブロックサイズを 70 から 100 以上にすることで性能が回復すると見込まれる。空間並列分割数を変化させた時の strong scaling 測定では、[表 4.90] のパラメータを用いた測定を行った。16 倍の並列規模での演算速度の向上比は、約 11 倍ほどであり [図 4.75]、おおむね良好といえる。

第一原理分子動力学計算の分野では、現在 1,000 原子未満が標準的な問題サイズである。1 万原子レベルの計算ができるようになれば、科学的なインパクトが大きい。また一般に 100ps 計算するためには 1,000 万 step 必要である。CPMD は $O(N)$ 法にくらべて δt を大きくとることができる利点がある。更に、ステップ数よりアンサンブル数が多い計算が良く行われており、機械学習を用いた粗視化パラメータの算出などインパクトのあるターゲット問題などに適用できる。

4.8.3.5.5　2019 年度実施

ソフトウェア調整 (1) では、CMPD 大規模問題のスケーラビリティ評価を実施した。

京を用いて、CMPD 大規模問題の高並列でのスケーラビリティ評価を行った。測定には、8 原子単位から任意サイズの入力を作成するツールを用いて、CPMD 用の 1,000 原子～4,000 原子の入力データを作成した。測定はバンド分割数を変化させ、ブロック数 (NBLK) は固定とした。性能については、codesign 版が遅くなる現象が見られたが、データ分割により通信が増えたことに起因するバンド方向の通信が目立った。バンド方向の跳び幅が小さくなるようなプロセスマッピ

表 4.88: ROTORB のメモリ分散前後の測定パターン

	1-1	1-2
node	864	
PROCS	8×12×3×3	
空間分割数	288	
NGRID	144×144×144	
NGRID0(1 プロセス)	10,368	
np_band	3	
MBC	4,200	
nprow×npcol	36×24	
MBLK	72	180
演算ブロック数 (1 プロセス)	1,200	192
node 形状	8×9×12	

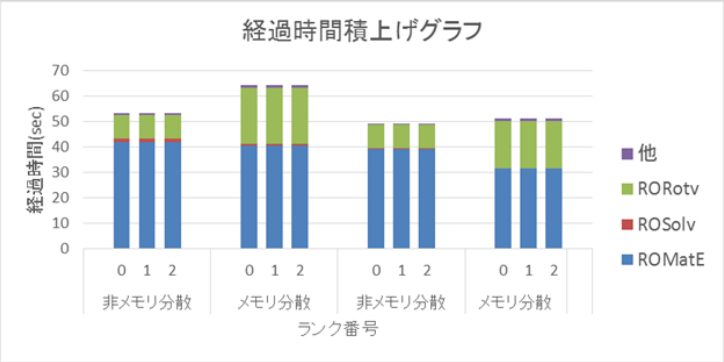


図 4.73: ROTORB のメモリ分散による実行時間の変化。左：MLBK=72、右：MLBK=180。

ングをすることで、バンド方向 bcast 通信が 4 倍高速化し、CPMD 全体の並列効率が 2.8 → 3.3 と向上した。また高並列で EigenExa を動作させることができた。EigenExa と ScaLAPACK の結果差異は SCF1,2 回目で見られなかった。上記の修正並びに EigenExa を含めた計算全てで、収束と結果の一致することが確認できた。

4.8.3.6　メモリ分散版の統合

4.8.3.6.1　2017 年度実施

詳細設計 (3) では開発期間 II にて、これまで部品として整備してきたメモリ分散の試作コードを、全体を通して動作するように、統合整備の作業を実施した。本年度統合した区間は GS と SD のカーネルプログラムであり、テスト問題のケースが問題なく動作することを確認した。

1. TGS カーネルプログラムの統合版の試作
   詳細設計 (3) 開発期間 II では、対角優先などのチューニング版については、並列性能評価が十分なされていないため、まずは全体をメモリ分散処理で動作することを目的に、オリジナルの TGS メモリ分散版の統合を実施した。GS の中でも性能評価を行っている対角優先チューニングのメモリ分散版の統合については、未統合の計算経路である CPMD などの統合並びに性能評価と併せて、2018 年度以降に実施する。

2. SD カーネルプログラムの統合版の試作
   SD カーネルプログラムの中で、MatE, RotV, Eigen 連携などのカーネルプログラムの統合

表 4.89: ROTORB のバンド分割数に関する weak scaling 測定パターン

測定ケース	3-1	3-2	3-3(2-3)	3-4	3-5	ﾀｰｹﾞｯﾄ
node	1,296	2,592	5,184	10,368	20,736	41,472
PROCS	8×9×6×3	8×18×6×3	16×18×6×3	16×36×6×3	16×36×12×3	32×48×9×3
空間分割数	432	864	1,728	3,456	6,912	13,824
NGRID	144×144×144					288×288×288
NGRID0(1 プロセス)	6,912	3,456	1,728	864	432	1,728
np_band	3					3
MBC	32,768					32,768
nprow×npcol	36×36	54×48	72×72	108×96	144×144	216x192
MBLK	576	576	576	576	576	–
演算ブロック数（1 プロセス）	1,083	1,083	1,083	1,083	1,083	–
node 形状	8×9×18	8×18×18	16×18×18	16×36×18	16×36×36	32×27×48

表 4.90: ROTORB のグリッド分割数に関する strong scaling 測定パターン

測定ケース	3-1	3-2	3-3(2-3)	3-4	3-5	ﾀｰｹﾞｯﾄ
node	1,296	2,592	5,184	10,368	20,736	41,472
PROCS	8×9×6×3	8×18×6×3	16×18×6×3	16×36×6×3	16×36×12×3	32×48×9×3
空間分割数	432	864	1,728	3,456	6,912	13,824
NGRID	144×144×144					288×288×288
NGRID0(1 プロセス)	6,912	3,456	1,728	864	432	1,728
np_band	3					3
MBC	32,768					32,768
nprow×npcol	36×36	54×48	72×72	108×96	144×144	216x192
MBLK	576	576	576	576	576	–
演算ブロック数（1 プロセス）	1,083	1,083	1,083	1,083	1,083	–
node 形状	8×9×18	8×18×18	16×18×18	16×36×18	16×36×36	32×27×48

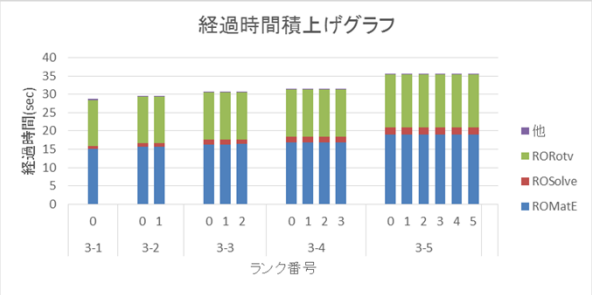


図 4.74: ROTORB のバンド分割数に関する weak scaling 測定結果

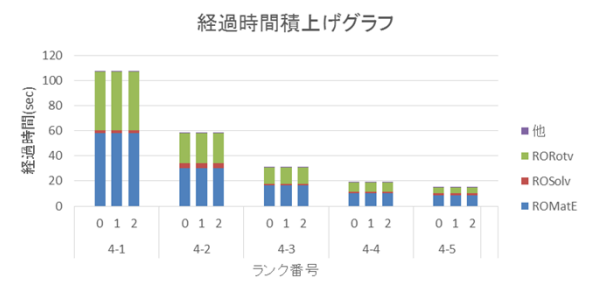


図 4.75: ROTOORB のグリッド分割数に関する strong scaling 測定結果

作業は完了した。

4.8.3.6.2 2018 年度実施

詳細設計 (3) 開発期間 II に統合を実施した RSDFT 環境の調査を行ったところ、いくつか問題があることがわかった。見つかった問題は以下の通りである。

1. SiO_2 1,536 原子の計算について、sdsyevd 内で結果に差異があった。
2. EigenExa の動作について、blacs の context の設定周辺で問題があり収束しないことがある。
3. MPI 環境で IntelMPI を使用するとデッドロックが発生した。
4. CPMD の ROTORB/ROTORB2 に非並列計算が残存していた。

SiO_2 1,536 原子での結果差異については、EigenExa に入る前の MatE 内で vtemp2, wtemp2 のメモリ解放処理が欠落していたため、エラー若しくは結果異常になったと確認され修正した。blacs の context の設定については、コミュニケータの処理に関して環境依存の手続きが見つかり修正を施した。Intel MPI については EigenExa のサポート対象外のため RSDFT の問題ではないと確認された。CPMD については、バンドメモリ分散になってない不具合の修正することで、並列性が向上した。現在の性能評価環境では、5,000 原子の問題はメモリ不足でテストしていないが、適切な計算資源を使用すれば動作することが期待される。

詳細設計 (3) 開発期間 II に行われた RSDFT の統合について、RSDFT および RSDT-BMD の高度化および公開支援として、実施機関で統合した環境の整備を行った。整備された統合環境は、入

測定ケース	3−1	3−2	3−3	3−4
Node（並列数）	576	1,152	1,728	2,304
PROCS	6×12×8×1	6×12×8×2	6×12×8×3	6×12×8×4
空間分割数	576			
バンド分割数	1	2	3	4
NGRID	128×128×128			
NGRID0(1プロセス)	3,641			
NBAND	9,216			
NBAND0(1プロセス)	9,216	4,608	3,072	2,304
NBLK	192			
NPROW	24	36	48	48
NPCOL	24	32	36	48
node形状	6×12×8	6×12×16	6×12×24	6×12×32

図 4.76: CPMD のスケーラビリティの測定パターン (Si 4,096 原子)

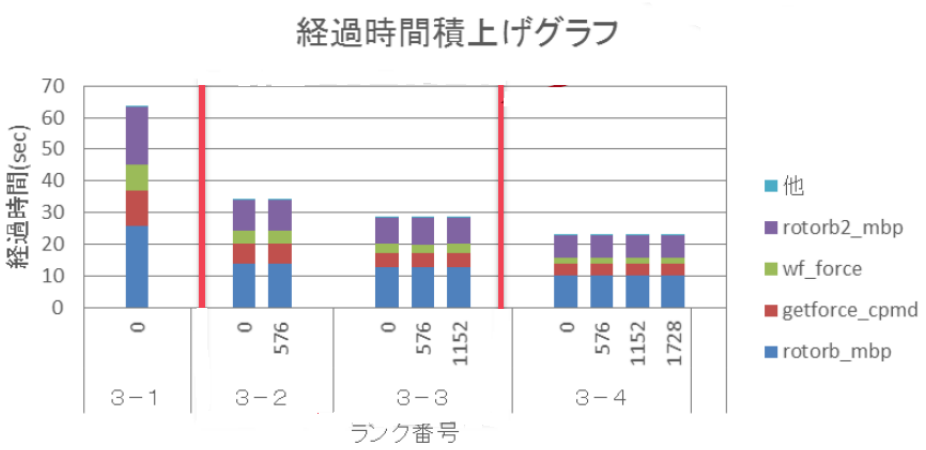


図 4.77: CPMD のスケーラビリティ測定結果 (Si 4,096 原子)

力データ制限付きで動作するものである。制限内容は、RSDFT に内在するものであり、EigenExa ライブラリの組み込みを実施した時、プロセス縮小をしなければ問題はない。9 月には github を立ち上げ、これにより、現段階の最新のメモリ分散版での実機性能測定が可能な状態になった。

詳細設計 (4) では、DGEMM カーネルについて、実機を用いた性能評価を行った。アプリ全体については、実機を用いて小規模性能測定に向けたテスト問題として、ターゲット問題である 100,000 原子系、300,000 バンドの SiO_2 問題について、1 ノード分切り出した SiO_2 1,536 原子の GGAOBE 計算の入力データ作成した。この問題は、京 500 ノードで数十秒で動作するが、バンド方向には分割せず、グリッド方向にターゲット問題と等価な問題として切り出されたサンプル入力である。この問題を用いて、京並びに FX100 上で、実機測定に向け適切か検証を実施した。このサンプル問題ではバンド分割されていないため、300,000 元を 1 ノードで対角化する必要があり、GS の処理区間では問題なく動作するが、対角化の処理区間でメモリ不足になることがわかった。対角化の実機評価に対応するため、バンド数は原子数の 3 倍であると仮定してメモリにのる規模を調べたところ、192atom/1node、若しくは 288atom/2node が計算可能な問題として妥当であることが

分かった。

RSDFT では、EigenExa による対角化のハウスホルダー変換が gemv 相当の計算であり、37MW ぎりぎりと消費電力が大きく評価されていた。但し 1SCF あたり 18.6 秒/1,051 秒程度の実行時間である。gemv 操作は、バンド幅ネックであるがその裏で演算もしており、L1,L2 もビジーである。定格などの問題で、ノードあたり 223W が上限であり、性能電力予測ツールによるとわずかではあるが上回っている。試作機を用いた評価では、EigenExa の tread1 処理を切り出しを行い、電力対策の必要性を判断することになった。但し EigenExa 内で ScaLAPACK を使用するため、ライブラリ評価環境の整備の問題で、実機評価は 2019 年度に実施した。

18 秒程度の計算時間とはいえ、一瞬でも電力がオーバーしたと判断されると、それ以降の大部分の計算は周波数を下げて行われる可能性がある。このため電力制御や電力チューニングなどの対応を考える必要がある。

なお、CG カーネルの性能評価並びにチューニングは統合版で実施した。

4.8.3.6.3　2019 年度実施

ソフトウェア調整 (1) では、以下 2 項目の作業を実施した。

1. 対角優先 GS カーネルの RSDFT アプリへの統合
2. メモリ分散版 RSDFT のサンプル問題の検証

以下に、個別の作業内容について記載する。

1. 対角優先 GS カーネルの RSDFT アプリへの統合

   RSDFT のメモリ分散のために開発した対角優先 GS カーネルを、RSDFT アプリへの統合を行った。input parameter(fort.1 内の GS) を、アルゴリズムを 13 に設定することで利用できる。対角移動 (オリジナル高橋)GS と対角優先 GS で block サイズを変化させると結果が異なる問題が見つかった。この差異は 1 並列でも発生したが、直交化処理により 0 になる絶対値が少ない成分でのみ見られた誤差であったため問題はない。また SD 処理内で ScaLAPACK から処理が戻ってきた時に、最終桁で誤差が発生し、収束までの SCF 繰り返し回数が 1 回程度ずれることがあった。分割が正方形でない時の ScaLAPACK の誤差と分かり、RSDFT の問題でないことを確認した。対角移動 GS については、各プロセスが全データを持つことが前提となっているためメモリ分散化は行っていない。

2. メモリ分散版 RSDFT のサンプル問題の検証

   RSDFT のアプリ全体性能を少ノード実機検証に向けて、動作検証できる小規模問題の作成を行った。当初作成したデータは 100,000 原子系、300,000 バンドの SiO_2 問題を 1,536 プロセスに切り出した入力であったが、バンド分割されていなかったため、300,000 元を 1node で対角化する必要があった。対角化処理の評価にもに対応するため、メモリにのる問題を京並びに FX100 を用いて調査したところ、192atom/1node、若しくは 288atom/2node が少ノード実機環境で計算可能な問題規模であると見積もった。

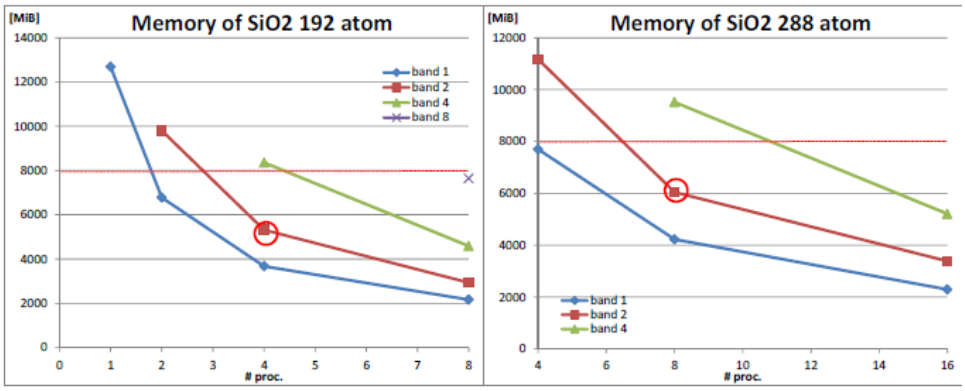


図 4.78: 京を用いたアプリ全体のメモリ消費量と実行時間の測定結果 (SiO_2)

4.8.3.7　実機を用いた動作・性能評価

2011 年に Gordon Bell 賞を受賞した際は、メモリ量が 16[GB/proc.] の京の上で、バンド方向のプロセス分割に対し、メモリ未分散版の RSDFT を用いて計算が行われた。本 WG は、同じ規模の問題を 8[GB/proc.] のメモリを搭載する富岳上で、より少ノードでスループットを向上させることが目的のミッションであると言える。これまでは、メモリをバンド分割方向にメモリを共有する実装になっていたが、本目標達成には、バンド方向のプロセス分割 (2011 年当時、3 分割程度) のメモリ分散の実装が必須であり、これまで、その設計・実装・評価に取り組んできた。本節では、実機を用いた RSDFT の動作検証並びに性能評価についてまとめる。

4.8.3.7.1　2019 年度実施

2019 年度のソフトウェア調整 (1) では、以下の 4 項目の作業を実施した。

1. 試作版 CPU を用いた RSDFT フルアプリの小規模問題の検証
2. 富岳共用前評価環境 I を用いて DGEMM カーネル性能の検証
3. ブーストモードで電力超過懸念がある EigenExa の tread1 の電力評価
4. 富岳共用前評価環境を用いたアプリ並列実機検証

以下に、個別の作業内容について記載する。

1. 試作版 CPU を用いた RSDFT フルアプリの小規模問題の検証

   試作版 CPU を用いた RSDFT フルアプリの小規模問題の検証を行った。192 原子の入力データを master 版、codesign 版 RSDFT を使用して、FX100 並びに実機で測定を行った。対角化、FFTW は含めないという条件での測定にて、FX100 と試作版 CPU で結果が一致することを確認した。また master 版と codesign 版では計算されたエネルギーの値が異なり、codesign 版はメモリ使用量が 1.5 倍ほど必要であった。結果差異が起こりうる原因として、初期波動関数の違い、並びに、波動関数のインデックスの並び替え、また gram shcmidt のアドレスの付け替えなどにより発生しえ、収束回数が数回変わってもおかしくないものである。この問題は、CG 処理内で通信を波動関数束ねて転送する処理について、パラメータ (MBD) を使うと計算結果が変わり、収束しなくなることが分かった。MBD=1 では発生せず、新しい RSDFT ではその問題が解消されていたので、この 1 年内に解決して、codesign

版に反映していなかったものと思われる。CG 処理内の MBD の一部を逐次処理に変更することで、master 版では 2 種類の thread 並列で結果が一致し、SCF の収束することを確認した。また OpenMP を使用した時、FX100 と富岳試作機の計算結果の違いも見つかった。CG 処理内の preconditioning の箇所がマスタースレッドで非同期通信処理をしている箇所で発生しており、該当箇所を非スレッド並列実行することで、Total Energy の結果が一致することを確認した。本問題は、threads 並列に関する single ループの barrier 同期に関するシステム側の不具合の影響と判明した。また codesign 版と master 版で複素版と実数版で結果が異なる現象も見つかった。master と codesig は、初期波動関数は同じでないなどの差異はあるが、収束すれば同じになるものである。また grid 数がプロセス数で割り切れない時の考慮漏れがあり、計算が進まない問題が見つかった。FFT などに起因する制限である。利用者に注意喚起するために、実行時 Warning を出力すると供に、ドキュメントに制限事項として記載することになった。
性能は、FX100 と比べてその他のコストが 2 倍程度増加していた。増大した区間は、calc_PBE_c と phase_control であった。この区間について SWP の促進などの最適化を実施したところ、60 対角優先アルゴリズムについても問題なく動作し、7%〜11%程度高速化することが分かった。CPMD については、Si 8 原子、192 原子の系を用い、master 版、codesign 版を FX100 並びに試作版 CPU を用いて評価した。それぞれ数桁程度の精度であり、実行時間は、1 ノード実行で、CPMD 全体で試作版 CPU が FX100 の 2 倍以上の性能であった。特に ROTORB、ROTORB2 区間はコストが大きく A64fx の性能が FX100 の約 2.5 倍と優位であった。FX100 と A64fx で見られた精度の差異について、Intel との比較評価も実施した。Intel、FX100、A64fx でそれぞれ異なる結果になった。これら CPMD の精度について、研究フェーズで影響がないか実問題を用いて確認した。Si 8 scfconv の問題で、精度を上げた計算を行い、発散や振動などは見られずほぼ等価結果を得た。この件は、スプライン関数補間を行うかについてのコンパイルオプション-D_SPLINE_の有無による差異であった。

2. 富岳共用前評価環境 I を用いて DGEMM カーネル性能の検証

富岳共用前評価環境 I を用いて DGEMM カーネル性能の再評価を実行した。初回計算時に性能が低い傾向が見られたため、1 回目を捨て実行を入れることで、昨年評価時とほぼ同じ性能が得られることが分かった。しかし、実行時間が短い区間について昨年度の評価性能と差異があり、複数回まとめて測定することで、その差異が軽減したため、カウンタの精度の問題であると結論づけた。サイズが小さく実行時間が数十 [μ s] の NN 型 DGEMM でのみ性能差がみられたため、他にも問題がある可能性懸念され記録に留めた。

3. ブーストモードで電力超過懸念がある EigenExa の tread1 の電力評価

ブーストモードで電力超過懸念がある EigenExa の Householder 変換をする区間 tread1 について、試作 CPU による実アプリを用いた EigenExa の動作確認を行った。RSDFT に組み込んだバージョン 2.3d で、実機並びに FX100 上で ScaLAPACK と同等の結果になることを確認した。また EigenExa-2.4b など公開版の最新のものを使用すると、富岳共用前評価環境 II では、RSDFT 本体が deadlock する問題があった。これは、DGEMM に関する結果異常障害によるものであり、2020 年 2 月版でライブラリが修正されることで、動作するようになった。

RSDFT DGEMM performance							CMG ('19.10.2)			CMG ('19.10.24)			Reference			
id	TA	TB	M	N	K	FLOP	elapsed	FLOPS	%	elapsed	FLOPS	%	Previous	Excel	FX100	京
1	T	N	1536	1536	4913	2.32E+10	3.60.E-02	6.44E+11	83.8%	3.60.E-02	6.44E+11	83.9%	84.2%	78.3%	88.3%	89.0%
2	T	N	768	768	4913	5.80E+09	1.01.E-02	5.74E+11	74.7%	1.02.E-02	5.69E+11	74.1%	75.8%	68.1%	80.5%	84.4%
3	T	N	384	384	4913	1.45E+09	2.76.E-03	5.25E+11	68.4%	2.77.E-03	5.23E+11	68.1%	69.3%	62.1%	75.6%	80.5%
4	T	N	192	192	4913	3.62E+08	8.57.E-04	4.23E+11	55.1%	8.43.E-04	4.30E+11	56.0%	53.2%	39.0%	53.1%	73.2%
5	T	N	96	96	4913	9.06E+07	3.17.E-04	2.85E+11	37.1%	3.06.E-04	2.96E+11	38.5%	35.7%	17.6%	26.5%	58.2%
6	T	N	48	48	4913	2.26E+07	1.20.E-04	1.89E+11	24.6%	1.14.E-04	1.98E+11	25.8%	23.2%	11.9%	19.2%	22.9%
7	T	N	24	24	4913	5.66E+06	5.38.E-05	1.05E+11	13.7%	5.26.E-05	1.08E+11	14.0%	13.7%	8.1%	14.2%	6.0%
8	T	N	12	12	4913	1.41E+06	3.41.E-05	4.15E+10	5.4%	3.23.E-05	4.39E+10	5.7%	5.4%	5.6%	10.4%	1.7%
11	N	N	4913	1536	1536	2.32E+10	3.38.E-02	6.85E+11	89.2%	3.38.E-02	6.86E+11	89.3%	88.6%	80.4%	89.9%	87.3%
12	N	N	4913	768	768	5.80E+09	8.99.E-03	6.45E+11	83.9%	8.98.E-03	6.45E+11	84.0%	83.7%	77.2%	87.8%	86.2%
13	N	N	4913	384	384	1.45E+09	2.35.E-03	6.17E+11	80.3%	2.31.E-03	6.27E+11	81.6%	80.3%	70.5%	83.0%	83.4%
14	N	N	4913	192	192	3.62E+08	6.74.E-04	5.37E+11	70.0%	6.51.E-04	5.57E+11	72.5%	70.4%	55.5%	70.1%	75.6%
15	N	N	4913	96	96	9.06E+07	2.31.E-04	3.92E+11	51.0%	2.16.E-04	4.18E+11	54.5%	50.4%	54.6%	69.6%	60.6%
16	N	N	4913	48	48	2.26E+07	9.46.E-05	2.39E+11	31.2%	6.28.E-05	3.60E+11	46.9%	46.8%	54.4%	69.3%	26.4%
17	N	N	4913	24	24	5.66E+06	5.23.E-05	1.08E+11	14.1%	2.75.E-05	2.06E+11	26.8%	27.9%	37.2%	51.1%	11.2%
18	N	N	4913	12	12	1.41E+06	2.32.E-05	6.11E+10	8.0%	1.27.E-05	1.11E+11	14.5%	12.6%	22.0%	33.5%	7.2%

図 4.79: 詳細設計 (4) での実機検証との DGEMM 性能比較。赤枠内の性能が異なる。

eigen_FS 大規模実機検証と RSDFT 性能実証の時期が重なるため、数値ライブラリ WG とは検証協力を行って実施した。

4. 富岳共用前評価環境 II を用いたアプリ並列実機検証

富岳共用前評価環境を使用して、RSDFT アプリの並列時の動作検証を実施した。この検証により 6 件の問題が見つかり、この問題を修正することで、1,152cpu まで実行し、3,072 原子までの小さな問題で問題なく動作することを確認した。この評価により見つかった問題は以下の通りである。

(a) バンド数がブロックサイズで割り切れない時の考慮漏れ
本問題により、配列の右辺と左辺で大きさが異なる場合がある不具合が見つかったため修正した。CPMD などのサンプル問題では、必ず割り切れるように設定されていたため顕在化しなかった。

(b) 内部プロセスマッピングの考慮漏れ
ScaLAPACK へのコミュニケータのプロセスマッピングで考慮漏れが見つかり修正した。

(c) 返り値未定義による不安定動作
RSDFT を-Knoopenmp で構築すると SCF 計算が開始しない問題が見つかった。原因は、通常計算する sweep 処理を、今回計算していなかったため、ierr の値が不定になることであった。reurn の前に ierr=0 と定義することで対応した。コード内の他の箇所については初期化されていることを確認した。

(d) -O2 によるコンパイラ abort
RSDFT を-O2 で構築するとコンパイル時の abort が発生した。本件はコンパイラの最適化の考慮漏れで修正された。

(e) 試作版 CPU における性能ぶれ
評価環境 I でのみ SCF が step ごとに 10 倍程度以上の性能ブレが発生した。本性能問

題は、4cpu のうち 3cpu で再現し、試作版 CPU 初期ロットによるものと結論づけた。

(f) DGEMM の結果異常に起因する deadlock
並列数によって、SCF が deadlock する現象が見られたこれは、DGEMM の不具合により、ScaLAPACK の結果が変わり収束しなくなったことによると考えられ、2020 年 2 月に本問題が修正されることで、問題なく動作するようになった。

また性能上、バンド分割により 1GB 程度のメモリ使用量増大と 2〜5 割程度実行時間が増大する傾向が見られた。性能については、バンド分割によりバンド方向の bcast が発生することによるものであり、このオーバーヘッドは妥当であると結論づけた。

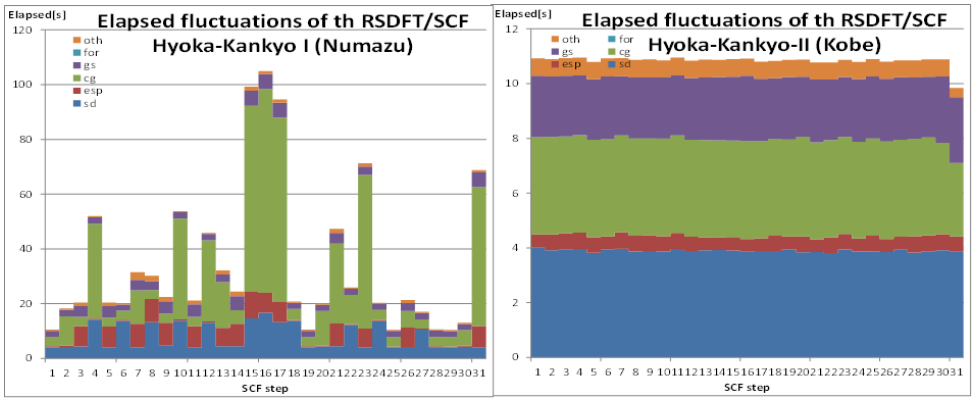


図 4.80: 富岳共用前評価環境 I(左図),II(右図) での SCF 実行時間のぶれ

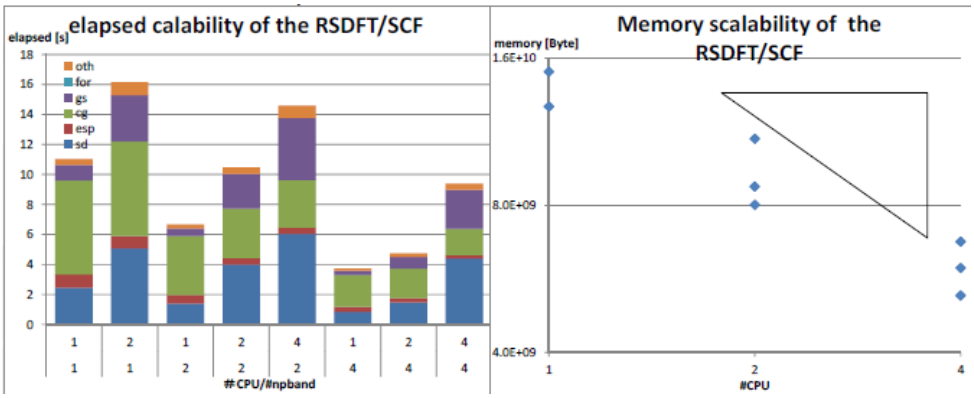


図 4.81: 富岳共用前評価環境での小規模フルアプリケーションのスケーラビリティ

4.8.3.7.2　2020 年度実施

2020 年度ソフトウェア調整 (2) として、以下の 4 項目の作業を実施した。

1. 意図しない制限時間タイムアウト (デッドロックなど) 問題の解決
2. メモリ削減
3. 代表プロセス方式の性能評価
4. 電力評価

2020 年 4 月から富岳共用前評価環境 III が開始され、システム開発と並行して、ターゲット問題 27 ラックを動かすことが可能な比較的大規模な環境を用いて、動作検証並びに性能評価を実施した。既に、2019 年度の富岳共用前評価環境 II において、3 ラックほどの小規模システムでの動作検証が行われ、前節で記したように、アプリ側、システム側の課題がいくつか見つかっていた。以下に、個別の作業内容について記載する。

1. 意図しない制限時間タイムアウト (デッドロックなど) 問題の解決

   2020 年度から始まった富岳共用前評価環境 III では、大規模な問題設定を用いて、多ノードを使用をした評価が可能になった。しかし、その評価の過程で、意図しないタイムアウトやメモリ枯渇という課題が顕在化した。これらの問題は、計算する問題規模や並列数を変更するごとに発生した。意図しないタイムアウトについてまとめると、以下の 3 項目に分類できる。

   (a) 不要な初期化処理の削減 (アプリ側の修正)
   (b) blacs_gridmap に与える usermap の重なりによるデッドロック (アプリ側の修正)
   (c) メモリ増大に伴うデッドロック (システム側の修正)

   以下に、それぞれ詳細について述べる。

   (a) 不要な初期化処理の削減 (アプリ側の修正)

   本 WG で評価を行うターゲット問題は、繰り返し単位を 1SCF として評価を行い、初期化区間の評価はしない。この理由は、実問題では初期化は 1 回だけ実行され、あとは 200 回程度の SCF の繰り返し収束計算を行うため、このコストの評価は必要ないためである。しかし実機評価では毎回初期化を行う必要があるため、初期化に関わるコストが多く占め、それだけで 1 時間ほどの資源を消費しタイムアウトの原因になり得る。今回、初期化処理の入力パラメータを調整したり、実計算でのみ必要となる精度に影響する処理を省くことで対応した。これにより SiO_2 の 3,072 原子で 12rack(4,608cpu) まで動作することを確認した。

   (b) blacs_gridmap に与える usermap の重なりによるデッドロック (アプリ側の修正)

   上記問題を回避した後も、SiO_2 3,072 原子、24[rack](9,216[CPU]) の評価では初期化処理でタイムアウトが発生した。発生箇所は、袖通信の isend/irecv であった。しかしカーネルでの評価ではデッドロックは発生せず、MPI_barrier を挿入して、袖通信に到達するプロセスを調べたところ、この通信の前に行われていたサブコミュニケータ生成処理で完了しないプロセスがあることが分かった。本事象は 1cpu を用いて半分以下のプロセス数のサブコミュニケータを切り出した際にも発生したことから、少ノードでの再現環境を切り出し、富士通通信ライブラリ開発者と協力して原因を調査したところ、blacs_gridmap での内部処理の MPI_Comm_create の処理で、一部のプロセスが通過していることが判明した。blacs_gridmap に与える usermap 情報に重なりがないように調整することで回避できた。本調査に関連して、SD の呼び出しにおいて、処理のまとめ上げの効果が考慮されない問題が見つかり、袖通信が多く発生する問題をコード修正により対応した。

(c) メモリ増大に伴うデッドロック (システム側の修正)

ここに至るまで、様々な箇所で頻繁に MPI 通信のデッドロックを検出した。検出箇所一定ではなく、様々な箇所で発生した。その多くは、メモリオーバーによるもので、一部のノードでメモリ枯渇をおこし、MPI のプロセスが応答しなくなるという問題であった。プロセス数と問題規模の関係で、そのメモリ不足が発生するタイミングが変わるためデッドロックが発生する箇所が変わったと考えられる。これらの問題は、他のアプリケーションでも見られており、とくにメモリリークによりメモリが増えた際に、あるタイミングで一部のプロセスが応答しないという現象として顕在化している。下節に記すメモリ解析、並びに、重点アプリの 1 つ FFB のメモリ解析を行った結果、メモリ使用量にプロセス間インバランスが存在し、メモリリークによる使用量増大の効果と重なってデッドロックを引き起こしていることが分かった。このメモリ使用量のプロセス間インバランスについては、富士通システムソフトウェア開発部隊が調査を行い、OS が使用するノーマルページ領域が枯渇した際に、ユーザが使用するラージページ領域を侵害し、ラージページ領域がフラグメント化することによるメモリ使用量のインバランスが発生していたことが分かった。本件は、Redhat OS の仕様であり、修正は難しいため、メモリ使用量が超過した時にデッドロックではなく、適切にメモリ枯渇を検出できるよう、システム開発にフィードバックした。

2. メモリ削減

タイムアウトの問題が解決することで、ほぼターゲット問題を実行するのに必要なシステム規模 (27rack) の実行が可能になった。しかし計算できた問題規模が SiO_2 3,072 原子までであり、ターゲット問題の 110,592 原子には 2 桁届かない。その理由はメモリ枯渇によるもので、下に記す要因で発生すると整理できた。

(a) 問題設定によるメモリ使用量増加
(b) integer4 オーバーフローによるメモリ枯渇
(c) プロセス数に比例する通信サイズの allreduce によるメモリ枯渇
(d) SD の RotV 処理におけるメモリ枯渇
(e) SD の MatE 処理におけるメモリ枯渇
(f) FFT のメモリ枯渇
(g) MPI ライブラリが使用するメモリ使用量の削減
(h) allocate の削減、通信バッファの削減

メモリ使用量の解析にあたっては、allocate 処理をフックするラッパー関数を作成しアプリ性能評価に用いた。また、/proc に出力されるメモリ量をトレースする方法を用いて実測による評価を実施した。allocate 処理でのメモリ使用量の解析から、一部のテンポラリ配列が解放されず使われ続けられていることがわかった。これらのテンポラリ配列の多くは、CPMD 計算などで何度も allocate/deallocate を繰り返す時のオーバーヘッドを削減するために導入されたものであり、オプションで切り替えられるように対応した。/proc 内にある info を参照することで、実際に使用するメモリ量を調べると、初期化にて確保されたメモリの一部がそのまま SCF でも確保されたままであることが分かった。またメモリ使用量に 100MB ほどのプロセス間インバランスがあることが分かり、システムの確保する領域によるラージペー

ジ領域のフラグメント化が発生していることが分かった。ノーマルページを使用時の比較も行い、8%程度の削減は見込めることが分かった。またメモリ使用量の調査には、並列パターンを変更する (バンド分割数=1〜6 など)、ステージングなどにより性能ブレを抑える、normalpage での比較を行う、問題規模の変更が難しいため grid 数はそのままで原子数だけ変更する (vacancy の導入) などを行い、比較評価を実施した。

(a) 問題設定によるメモリ使用量増加

京で実行した際のメモリ使用量の記録と、富岳のメモリ使用量の比較を行った。[表 4.91] は、いくつかの問題規模での京と富岳のメモリ使用量の比較である。京と富岳の間では使用量の差は小さく、その差は妥当であったが、SiO_2 計算では Si での計算と比べてメモリ使用量が大きい傾向があり、問題設定によって、メモリ使用量が大きく異なるようであった。評価用のベンチマークで使用していた問題は SiO_2 であったため、3,072 原子程度でメモリが枯渇していた。これは SiO_2 の方がグリッドの刻み幅が小さいことによるものであり、結晶の種類によりグリッド精度が変わるためと考えられている。ターゲット問題は Si 110,592 原子を想定しており、SiO_2 ベンチマークを用いた評価からターゲット問題で使用する Si を用いた評価に切り替えることで、16,384 原子程度までの計算が可能になった。SiO_2 については、3,072 原子までしか計算できなかったが、下記で挙げるメモリ削減の対応をすることで、未評価ではあるものの、より大規模な問題の計算ができるようになる。

表 4.91: 京と富岳のメモリ使用量の比較、青罫は比較的同規模での計算に相当する。

計算対象				計算資源				
組成	原子	NBAND	NGRID	マシン	分割数	メモリ[MiB]/proc.		
SiO2	1,536	4,496	168^3	京	512	4,133.4	4,011.2	4,043.2
SiO2	3,000	9,000	180^3	京	1,728	7,269.4		
SiO2	3,000	9,000	210^3	京	1,000	9,189.4	8,843.2	
SiO2	5,184	15,200	216^3	京	3,456	13,707.2	14,245.4	
SiO2	5,184	15,200	252^3	京	5,832	14,213.4	(LOM)	
SiO2	3,072	9,216		富岳	4,608	4,048.4		
SiO2	3,072	9,216	192x192	富岳	9,216	3,915.3		
SiO2	3,072	9,216	x384	富岳	18,432	4,075.7		
SiO2	3,072	9,216		富岳	36,864	4,550.2		
Si	4,096	9,216	128^3	京	1,152(576x2)	1,352.5		
Si	4,096	9,216	128^3	京	1,728(576x3)	1,160.5		
Si	4,096	9,216	128^3	京	2,304(576x4)	1,004.5		

(b) integer4 オーバーフローによるメモリ枯渇

subspace_diag_module 内に、メモリ分散されたバンド数の自乗 (MB*MB) オーダ要素数の配列が存在し、配列要素変数に integer4 を用いていたためオーバフローし、配列サイズが適切でなくなり、Si 32,768 原子規模の計算でメモリ枯渇が発生した。大規模な問題の評価で初めて顕在化した問題であるが、2011 年に京で測定した際には修正済みであり、今回の評価で過去のコード元に巻き戻されていた。コード修正によりメモリ枯渇を回避した。

(c) プロセス数に比例する通信サイズの allreduce によるメモリ枯渇

初期化処理の ps_proc2_module のモジュール内で、MB*GRID プロセス数のサイズの MPI_Allreduce を行う処理があった。大並列下では通信量が増大し、MPI 通信でのメモリ枯渇を引き起こした。本箇所はメモリ分散処理がなされておらず、原子ループを入れ替えて分散処理を行うことで対応した。MPI_Allreduce 通信を分割処理されるため性能は劣化する。

(d) SD の RotV 処理におけるメモリ枯渇

Si 65,536 原子規模で、SD 処理の前半部の RotV 区間で計算する、$A = A * B$ の回転を行う行列行列積処理でメモリ不足が起こることが分かった。A に相当する行列は波動関数であり、メモリ使用量の大半を占める配列である。本処理は、行列 A が入力と出力で共通であるため IN_PLACE 処理できず、バッファを含めて波動関数 2 つ分のメモリ空間を必要としたためと考えられる。処理を 2 分割することで、波動関数の 1.5 倍のメモリ使用量に抑えることができた。但し、本区間は RSDFT における高性能区間であり、性能劣化が予測されるため、他箇所の改良に伴い、コントロールパラメータにより元の 2 倍確保しても動作するように改良した。

(e) SD の MatE 処理におけるメモリ枯渇

SD 処理の後半部の MatE 区間でも RotV 区間同様、波動関数 2 つ分の配列を確保しており、RotV 区間のメモリ削減の対応によりメモリ枯渇が顕在化した。本区間は、単純な配列分割処理による対処ができないため、block サイズごとに処理を分割した。実行時間は伸びるがメモリは、1/block サイズに削減できる。本処理も RSDFT 内の高性能区間であり、性能劣化の影響が大きいため、他のメモリ削減ができた際に、メモリを冗長に使い高速化できるように戻せるよう修正した。

(f) FFT のメモリ枯渇

FFTW では初期化処理として、高速なアルゴリズム選択のためのプレ実行や exp を格納する一次配列を用意する plan 生成の処理を行う。この初期化処理は 1 回のみ行い、そこで用意された一次配列を使い回して、実計算を行うのが一般的な使用方法である。しかし一次配列を長期にわたって保持するため、メモリ枯渇の原因になる。RSDFT にて FFT の呼び出す箇所はハミルトニアンの初期化と更新時のみであり、使用頻度は多くない。このことから、毎回 plan を生成し、破壊する実装に切り替えた。FFT 終了による削減可能メモリは 0.8GB である。しかしメモリ使用量を/proc 情報から調査したところ、plan の破壊によるメモリ削減は見られないことがあった。これは使用するラージページがフラグメント化し、必ずしも毎回ページ解放がなされなかったためと考えられる。なお allocate のフック関数によるメモリ使用量の調査方法では、FFTW ライブラリなど構築済みライブラリで獲得するメモリ allocate 処理はフックできない問題がある。また今後の更なるメモリ削減案として、FFTW の r2c/c2r 化、in-place 化が考えられる。

(g) MPI ライブラリが使用するメモリ使用量の削減

RSDFT は富岳上で、集団通信が全実行時間の 3 割を占める。また通信時にデッドロックが発生することが多かったため、通信処理におけるメモリ量増大が大きいと考え、通信処理に伴うメモリ使用量の確認を行った。MPI 統計情報により出力されるメモリ量は CMG 当たり 510MB となっており、この中に通信 buffer を含まないことを確認した。つまり 510MB 以外にもユーザが確保する通信バッファ相当のメモリが必要になることが分かる。MPI ライブラリが使用するメモリ量は MCA オプションにより制御可能であり、小規模でのテストの結果では、--mca common_tofu_max_fastmode_procs 0 を指定すると性能の劣化がなく、CPU あたり 1.2GiB ほどメモリが削減できることが分かった。但し、--mca common_tofu_memory_saving_method 2 --mca common_tofu_shared_recv_buf_size 65536 指定時や、9,216 ノードほどの大並列実行時に、--mca common_tofu_max_fastmode_procs 0 並びに、--mca common_tofu_large_recv_buf_size 1024 を指定した際に、SEGV が発生して使用できなかった。

(h) allocate の削減、通信バッファの削減

ここまでの対応より、ターゲット問題の 2/3 の規模原子数の計算まで計算できることができた。更にメモリ allocate のフック機能を用いた調査を進めると、FFTW で一次配列として確保される 3 配列 152MB/CMG、並びに使い回しをしている通信バッファ 354MB/CMGx2 の削減が可能であることが分かった。これらの配列を削減を行うことにより、2020 年 10 月にターゲット問題が動作し、性能評価が実施できた。

なお allocate メモリ量の見積もり値とメモリ使用量の実測値を比較すると、800[MiB/CMG] 程度な乖離があることが分かった。ターゲット問題の半分の原子数でのメモリプロファイル並びにメモリ使用量の時間変化は [図 4.82] の通りである。

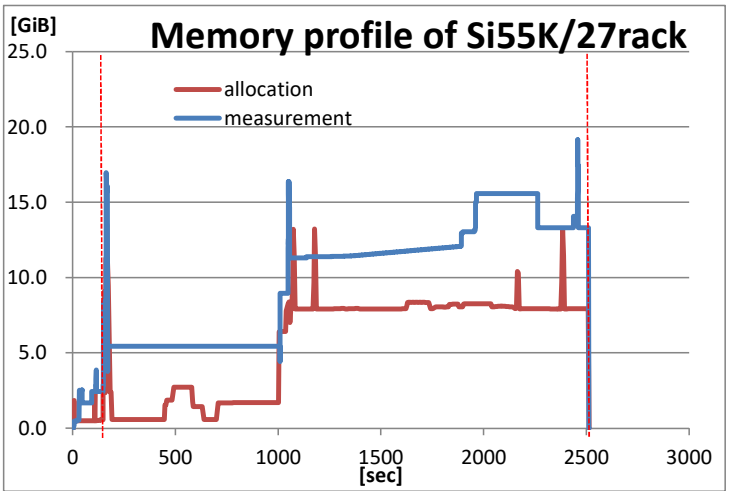


図 4.82: ターゲット問題の半分の原子数でのメモリ使用量プロファイル結果。allocate 処理からの見積もりと/proc からの実測を比較した。

この他にシステムソフトウェアに関連して、以下の 4 つのシステムソフトウェアでのメモリリークが見つかっている。

(a) FFB, GKV, GT5D, LatticeBoltzmann で見つかった Rendezvous 通信のメモリリーク
(b) RSDFT 実問題長時間実行, Horovod で見つかった、MPI_ALLREDUCE で MPI_IN_PLACE 指定時のメモリリーク
(c) LQCD, GT5D で見つかったコミュニケータ作成・破棄のメモリリーク
(d) GT5D で見つかった MPI_SEND_INIT でのメモリリーク

これらの修正により、今後より大きな問題規模の計算も可能になる。しかし、システムメモリ侵害によるフラグメント化の問題並びに MCA パラメータによる通信メモリの削減は、プロジェクト終了までに解決しておらず、運用フェーズでのシスム改善が望まれる。

3. 代表プロセス方式の性能評価

通信の性能チューニングとして、EigenExa で使用するプロセス数の絞り込みに対し、プロセスマッピングの最適化を行った。これは、バンド方向とグリッド方向のハイブリッド並列に用いるサブコミュニケータとは別で、独立に ScaLAPACK の行列プロセス並列の両サブコミュニケータにおける通信を同時に最適化する手法である。blacs の gridmap を用いて、行方向、列方向のコミュニケータを連続化することで実現する。ただし汎用的なコーディングは難しく、ターゲット問題を想定し、64x64、72x72、96x96 の大規模な問題に対して、6 次元プロセスマッピングを行った。評価の結果、どの分割パターンでも、オリジナルの 4procs./node 割り当てと比較して、1proc/node、6 次元マッピング有の方が通信時間を削減できることを確認した。64x64 では 1.57 倍高速化し、EigenExa 内の通信区間並びに ScaLAPACK が使われる DC の区間の性能は、性能分析予測シートによる通信予測性能と比較して向上しているのを確認した [図 4.83]。

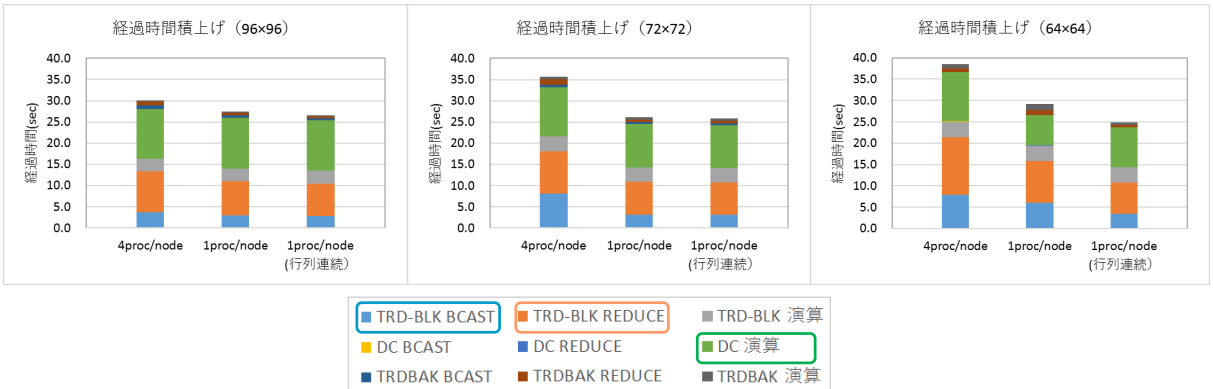


図 4.83: EigenExa のプロセスマッピングによる通信時間の変化。

また、Si 65,536 原子/24rack を用いて、代表プロセス方式による集団通信性能を評価した。この評価に用いた評価問題はターゲット問題とは異なるため、比較には通信時間ではなくスループットによる評価を行った。この評価による代表的な集団通信の性能を通信時間に換算すると、推定時間の 92.3%〜99.0%ほどであり、代表プロセス方式による通信性能見積もりは問題がない [表 4.92]。但し GS 内の iBcast 相当の通信は演算とオーバーラップさせているので、注意が必要である。

表 4.92: 代表プロセス方式における見積もりと実測の通信性能の比較。

区間	通信種別	要素数	メッセージ長 (byte)	回数	モデル式				1-3要素 Allreduce 推定時間	推定時間 (sec)	計測時間 (sec)
					オーバヘッド	Reduce	Bcast	推定時間			
SD – mate	REDUCE	262,144	2,097,152	6,144	0.00008	0.001296	—	8.453	—	8.453	—
		262,144	2,097,152	2,112	0.00008	0.001296	—	2.906	—	2.906	—
		262,144	2,097,152	64	0.00008	0.001296	—	0.088	—	0.088	—
		合計	—	—	—	—	—	—	—	**11.447**	**11.338**
SD – rotv	BCAST	262,144	2,097,152	16,384	0.00008	—	0.001086	19.096	—	**19.096**	**17.617**
GS	ALLREDUCE	262,144	2,097,152	8,064	0.00008	0.001296	0.001086	19.848	9019.500	19.848	—
		65,536	524,288	64	0.00008	0.001196	0.001000	0.146	17.896	0.146	—
		16,384	131,072	128	0.00008	0.001171	0.000978	0.285	8.949	0.285	—
		4,096	32,768	256	0.00008	0.001165	0.000973	0.568	4.476	0.568	—
		1,024	8,192	512	0.00008	0.001163	0.000972	1.134	2.241	1.134	—
		256	2,048	1,024	0.00008	0.001163	0.000971	2.267	1.127	1.127	—
		64	512	2,048	0.00008	0.001163	0.000971	4.534	0.577	0.577	—
		16	128	4,096	0.00008	0.001163	0.000971	9.069	0.315	0.315	—
		合計	—	—	—	—	—	—	—	**24.00**	**22.974**

通信性能最適化については、マップファイルを用いたプロセスマッピングによる最適化を実施した。バンド分割を b 軸=3 への割り当てを適用した。このマッピング処理により、SD 区間は高速化したものの、他の区間が遅くなり、ターゲット問題では、1,068[s/SCF] から 1,075[s/SCF] へと遅くなった。またバンド分割を 4 へと変更し、CMG に割り当てる変更を行った。これはマップファイルを用いなくてもバンド分割を優先的にプロセスマッピング (#PJM --mpi rank-map-bynode) することで実現できる。これにより、バンド方向の通信が CMG 内に閉じ込めることが可能になり、ターゲット問題で 933.7[s/SCF] と高速化した。

4. 電力評価

RSDFT では、EigenExa で用いられる Householder 変換を行う tread1 区間で、演算器並びにメモリを多く使用するため電力超過の懸念があった。実際のターゲット問題に相当する大並列での EigenExa を用いた対角化計算では、最大でも 96x96 など一部のノードを使用しないため、電力超過の心配はないと考えていたが、実問題で問題ないか電力評価を実施した。ピーク電力が問題になることから、評価にはターゲット問題の weak scaling な問題設定ではなく、ターゲット問題より最大電力大きいと想定される問題である、Si 16,384[原子/rack] の問題を用いて、2.2GHz での測定を行った。ジョブ統計情報による最大電力は、156[W/cpu], 59.9[KW/rack], 24.8[MW/system] と計測された。
また全ラックに取りつけられている 1 分分解能の電力計並びに、一部ラックに取り付けられている 1 秒分解能の電力計を用いて RSDFT の電力測定を行なった。電力値はどちらもほぼ同じであり、1 秒分解能の電力計では、各カーネルの電力使用量を詳細に分離できた [図 4.84]。これにより、想定していた EigenExa 内の tread1 での電力超過は発生していないことを確認した。
RSDFT のターゲット問題では、27rack を 3 次元プロセスマッピングを行って実行するため、単純にラックを占有できない。このためラック電力を評価すると、充填されたラックとそうでないラックへの偏りができる。システム電力を計算するときには、充填されたラックの電

力のみを集計する必要がある。また、ターゲット問題の実測は、1SCF で行うため、平均電力には大部分を占める初期化の電力、並びに、ステージングなどの電力が含まれる。このため主要カーネル区間での電力評価には、初期化などの処理を除く必要があり、複数回 SCF を実行したものとのエネルギー差を用いて平均電力の見積もりを行う必要がある [図 4.85]。

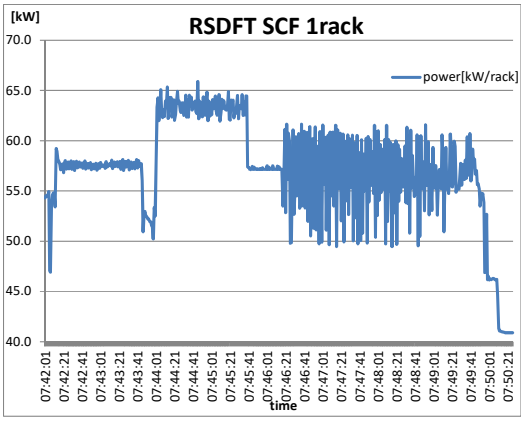


図 4.84: 1 秒精度のラック電力を用いた、RSDFT 1SCF の消費電力の変化。

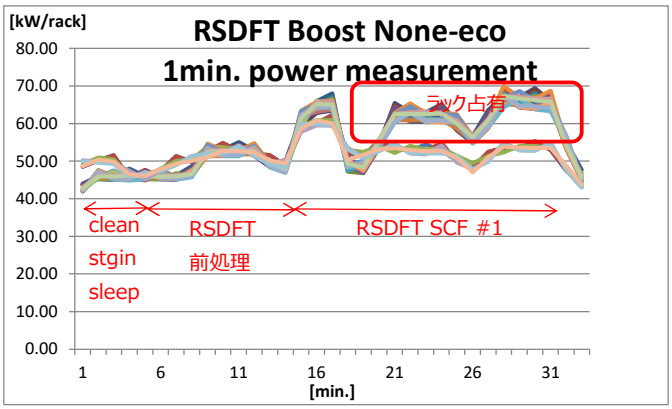


図 4.85: ターゲット問題を測定した時の 1 分精度の全ラック電力の消費電力の変化。

4.8.4 コデザインによるアーキテクチャへの要請

本コデザインを通してシステムへフィードバックした実装要請・改善項目を整理して以下に示す。

(1) 中メッセージの通信アルゴリズムの改善
(2) 長メッセージのノード内通信の改善
(3) ノード内通信のパイプライン処理の効率化
(4) ノード内複数の代表プロセスによる通信
(5) 複数 CMG 利用の DGEMM 性能の最大化
(6) MPI_IN_PLACE を使った Allreduce/Reduce 通信の Tofu 専用アルゴリズムの適用
(7) 特定のプロセス分割における EigenExa の異常終了
(8) EigenExa ライブラリ内で、PARALLEL リージョンから逐次版 DGEMM を呼び出す
(9) 通信メモリ使用量を削減する MCA オプションでの SEGV
(10) プロセス間のメモリ使用量インバランス
(11) OOM キラーで deadlock が発生時検出しメッセージを出力する
(12) SSL2 ScaLAPACK の pdgemr2d 関数の 54rack 以上での MRQ オーバーフロー

それぞれの詳細を以降に示す。

(1) 中メッセージの通信アルゴリズムの改善
・課題：
新代表プロセス方式における Allreduce のスループットが、旧代表プロセス方式及び京と比較して、1[KiB]〜1[MiB] の中メッセージ領域において性能が低い。

・通信ライブラリ WG における検討結果：
Allreduce の Reduce+Bcast のアルゴリズムにおいて、Bcast 部分に 6TNI 活用アルゴリズムによる性能改善を実施するとし、機能仕様書（通信ライブラリ編）2.1.5.1. 項「Bcast,Allreduce」へ反映した。

(2) 長メッセージのノード内通信の改善
・課題：
数 100[MiB] 以上の長メッセージ領域において、Allreduce のノード内通信の時間が、ノード間通信の時間と比較して数倍と大きい。

・通信ライブラリ WG における検討結果：
双方向 Trinaryx3 アルゴリズムによる性能改善を 2016 年度以降に検討とし、機能仕様書（通信ライブラリ編）2.1.3. 項「計算ノード内プロセス間通信性能改善」、2.1.5.1. 項「Bcast,Allreduce」へ反映した。

(3) ノード内通信のパイプライン処理の効率化
・課題：(参照：図 4.47)
1) 代表プロセス方式の Reduce のノード内通信時間の見積式 A において、コピーの項が不要と考えられ、見積式の見直しが必要である。
2) 見積式において、パイプライン処理のセグメント長がメッセージサイズに対して小さいことが前提となっているが、セグメント長が 64[KiB] が仮定され、メッセージサイズが数 MiB 以内で過小評価となるため、セグメント長の見直しが必要である。

・通信ライブラリ WG における検討結果：
1) 基本設計時点のモデル式を見直し、Reduce の式 A のコピーを除いた。この結果を元に、基本設計にて予測した性能を修正し、詳細設計（1）において性能予測を進めた。
2) 最適なセグメント長については実機評価後に決定した。

(4) ノード内複数の代表プロセスによる通信
・課題：
eigen モジュールのノード内プロセス数の検討において、ノード内に複数の代表プロセスによる通信が前提となる。

・通信ライブラリ WG における検討結果：
Trinaryx3 アルゴリズムの 1 代表プロセス方式を 2016 年度以降に検討、bintree3d アルゴリズムについては 2 代表プロセス方式を検討し、機能仕様書（通信ライブラリ編）2.1.5.1. 項「Bcast,Allreduce」へ反映した。

(5) 複数 CMG 利用の DGEMM 性能の最大化
・課題：
1MPI プロセスを複数 CMG により構成する場合が以下のようにあり、複数 CMG 利用の DGEMM 性能が最大となることが課題である。
1)RSDFT では現システムのメモリ容量を最大限利用するため、問題規模によってはメモリ容量不足となる場合があり、1MPI プロセスを複数 CMG により構成することで回避する方法がある。
2) ノード内の通信性能が高くない場合、1MPI プロセスを複数 CMG により構成することにより、通信時間が改善する場合がある。

・数値計算ライブラリ WG における検討結果：
複数 CMG 利用時は、CMG 単位とコア単位の 2 段階によるブロック分割方式を採用し、機能仕様書（数学ライブラリ編）1.1.1.9. 項「スレッド並列 DGEMM の実装：CMG を跨ぐ場合」へ反映した。

(6) MPI_IN_PLACE を使った Allreduce/Reduce 通信の Tofu 専用アルゴリズムの適用
・課題：
Allreduce/Reduce 通信にて、メモリ削減のために MPI_IN_PLACE を使うと、Tofu 専用アルゴリズムが適用されないために、性能が低下する。

・通信ライブラリ WG における検討結果：
1) メモリ削減にはならないが、MPI_IN_PLACE を使う場合も Tofu 専用アルゴリズムが適用される仕様とし、機能仕様書（通信ライブラリ編）2.1.5.1.4.2. 項「MPI_IN_PLACE の扱い」へ反映した。
2) メモリ削減については、長期的課題とした。

(7) 特定のプロセス分割における EigenExa の異常終了
・課題：
特定のプロセス分割で EigenExa-2.3k2 を動かすと、固有値計算が終了しない。

・数値計算ライブラリ WG における検討結果：
1)EigenExa-2.3k2 については二巾の分割数でたまたま動いていたので、計算結果が異常になる可能性があることが分かった。
2) 障害を修正し、EigenExa-2.3l がリリースされた。

(8) EigenExa ライブラリ内で、PARALLEL リージョンから逐次版 DGEMM を呼び出す
・課題：
開発を進めている DGEMM は thread 並列版を中心に進めており、セクターキャッシュ、共有 L2 有効利用の観点から、一般的には thread 並列版の使用を推奨する。

・数値計算ライブラリ WG における検討結果：
1)EigenExa ライブラリの逆変換ブロックから呼び出す DGEMM を thred 並列版 DGEMM に置き換えるためには、タスクスケジューリング、集団通信を用いた実装を非ブロッキング集団通信に変更することで実現可能である。
2)thread 並列版の DGEMM を呼び出す EigenExa を試作し評価を行う。
3) 逐次版 DGEMM と thread 並列版 DGEMM の利用シーンを、プログラミングガイドにまとめる。

(9) 通信メモリ使用量を削減する MCA オプションでの SEGV
・課題：
--mca common_tofu_memory_saving_method 2 --mca common_tofu_shared_recv_buf_size 65536 指定時や、9,216 プロセスほどの大規模実行時に、--mca common_tofu_max_fastmode_procs 0 並びに、--mca common_tofu_large_recv_buf_size 1024 を指定した際に SEGV が発生する。

・運用保守における検討結果：
調査を実施した。

(10) プロセス間のメモリ使用量インバランス
・課題：
プロセス間で 100MB ほどメモリ使用量にインバランスがあり、デッドロックなどを引き起こす可能性がある。
・運用保守における検討結果：
システムが使用するノーマルページ領域が枯渇した時に、ラージページ領域を使用し、フラグメント化が発生する。OS の仕様であり、根本的な対応は困難。

(11) OOM キラーで deadlock が発生時検出しメッセージを出力する
・課題：
上記メモリ使用量インバランスにより、デッドロックなどが発生した際、メモリ起因であることの切り分けが難しい。OOM キラーが一部ノードで動いた際に検出できるようにする。
・運用保守における検討結果：
他エラーを優先して表示しているため、表示の優先度を調整する。

(12) SSL2 ScaLAPACK の pdgemr2d 関数の 54rack 以上での MRQ オーバーフロー
・課題：
54rack 以上で pdgemr2d を用いてプロセス絞り込みを行うと MRQ オーバーフローが発生する。
・数値計算ライブラリ WG における検討結果：
ScaLAPACK の isend/recv にともなう MRQ オーバーフローであった。send/recv の利用、通信の分割や isend/isrecv の導入など他実装も検討したが、OMP_STACKSIZE パラメータで回避する。

表 4.93: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (RSDFT)

実装要請課題	対応
1 KB 1MB サイズの中メッセージ MPI_Allreduce の通信アルゴリズムの改善	MPI_Allreduce を MPI_Reduce+MPI_Bcast で実現する処理方式において、MPI_Bcast のノード内処理内容見直しと 6TNI 対応により性能改善
長メッセージのノード内通信の改善	Trinaryx6 アルゴリズムにより性能改善
ノード内通信のパイプライン処理の効率化	基本設計時点の見積もり式を見直し済み。最適なセグメント長については実機評価後に決定する
ノード内複数の代表プロセスによる通信	Trinaryx6 アルゴリズムや bintree3d アルゴリズムを用いた代表プロセス方式を採用
1 プロセスで複数 CMG 使用時の DGEMM 性能の改善	複数 CMG 利用時は、CMG 単位とコア単位の 2 段階によるブロック分割方式を採用

CMG 間 memcpy を用いた MPI_Bcast の高速化	複数スレッドの活用や 1 コピー方式の採用による性能改善を行う。
メモリコピー回数の削減による、代表プロセス方式の MPI_Allreduce の通信スループット向上	ノード間通信に 6TNI を活用したアルゴリズムを適用することで性能改善
MPI_IN_PLACE を使った Allreduce/Reduce 通信の Tofu 専用アルゴリズムの適用	MPI_IN_PLACE 指定時には、MPI_Allreduce は受信バッファ相当、MPI_Reduce は受信バッファの 2 倍の大きさの通信バッファを使用し、Tofu 専用アルゴリズムを適用する
EigenExa ライブラリの既知の問題点の解消	EigenExa-2.3k2 では特定のプロセス分割で実行した場合に固有値計算が終了しない、あるいは異常な計算結果を出力する可能性があることが判明し、解決を要請。EigenExa-2.3l のリリースで解決された。
EigenExa ライブラリで PARALLEL リージョンから逐次版 DGEMM を呼び出す機能	thread 並列版の DGEMM を呼び出す EigenExa を試作し、DGEMM の利用シーンをプログラミングガイドにまとめる
C 言語の関数引数がポインタ型である場合の最適化促進	C 言語のポインタ型への restrict 修飾子指定、および異なるポインタ引数が同じ領域に値を定義しないことを指示する翻訳時オプション (-Krestp=arg) もしくは最適化制御行 (!OCL NORECURRENCE) の指定により最適化を促進させる

4.8.5 富岳向け性能推定

4.8.5.1 富岳向けターゲット問題計算性能の推定

富岳向け性能推定は、4.1.5 節に説明される手順により実施した。
本課題の富岳におけるターゲット問題の実行性能の見積方法と性能予測の結果を以下に示す。

(1) 見積方法
計算の主要となる SCF 計算部分について、演算区間と通信区間をそれぞれ特定した。演算区間は京においてベースライン測定を実施して演算時間を推定し、通信区間は見積式により通信時間を推定して、演算と通信の合計を経過時間とした。
評価区間は、ターゲット問題と同様の規模の問題を京の全ノードを用いて以前に測定した結果から、97%を占める SCF 計算部分として設定した。更に、パラメータ毎に詳細区間を特定して、演算が 64 区間、通信が 63 区間を評価区間とした。この評価区間に対して見積式を適用すると、コスト比率が 91%を占め、見積結果が妥当であることを確認した。
そこで、この見積方法を使い、富岳推定用のターゲット問題・並列数等を設定して、京と富岳の性能を推定した。

- 演算性能の見積方法
  演算の主要となる DGEMM の呼び出し部分を FX100 にて詳細プロファイラ情報を取得して、性能電力予測ツールを用いて経過時間を推定する。推定した結果と演算量から各区間の実行時間を見積もる。但し、対角化の外部ライブラリ（EigenExa）の性能について、京での実効演算性能を仮定した評価を行っており、同等の精度での評価を行っていない。
- 通信性能の見積方法
  以下の手順により通信時間を見積もる。
  (a) プログラムの通信区間を特定し、通信情報として通信種類・データ型・通信サイズ・通信回数を特定する。
  (b) メッセージ長とハードウェア性能値から、通信アルゴリズムによる 1 回当たりの通信時間を算出する。（通信アルゴリズムによる見積は、空間方向が代表プロセス方式 (第 4.8.1 節参照)、その他は通常の通信方式を利用する）
  (c) 通信 1 回の実行時間に通信回数を掛けて見積時間とする。
- 全体時間の見積方法
  課題実行全体の見積は、ケース 1 回当りの実行時間に、実行するケース数を掛けて全体時間・演算性能を算出する。
  （全体時間）＝（1 回当りの実行時間）×（ケース数÷同時ケース数）
  （同時ケース数）＝（全ノード数 × ノード内プロセス数 /1 回の MPI 並列数）

(2) 富岳での性能予測結果

- 実行条件
  富岳のシステム構成における RSDFT の実行条件は、3 次元ノード形状 $8 \times 48 \times 27$ を使って、1CMG 当り 1 プロセスを割り当て、全体で 41472 プロセスにより 1 ケースを実行するものである。
  課題実行としては、1SCF の計算を 200SCF × 24 ケース分を同時 16 ケースで実行する。

- 課題実行性能（基本設計時の予測）
  基本設計における京と富岳の予測性能を表 4.94 に示す。

表 4.94: RSDFT の性能予測結果 (基本設計時)

機種		課題実行	京倍率	1SCF 時間 [秒]			実行効率	電力効率
	電力制御	[日]	[倍]	全体	演算	通信	[%]	[GF/W]
京	-	126.1	-	4538.3	3783.0	755.3	69.6%	-
富岳	電力制御なし	4.3	29	1226.8	770.1	456.6	47.6%	8.0
(通常モード)	メモリ電力制御	4.5	28	1294.9	838.3	456.6	45.1%	8.8
富岳	電力制御なし	4.1	31	1154.3	697.7	456.6	45.6%	6.7
(ブーストモード)	メモリ電力制御	4.5	28	1290.1	833.5	456.6	40.8%	7.0

- 課題実行性能（詳細設計 (1)：パラメータ修正・一部のコデザイン結果反映による予測）
  通信性能の見積式において、1〜3 要素の Allreduce の時間の見直し、コピー・リダクション性能の見直し、及び、コデザイン検討課題 (3) の 1) の Reduce のノード内コピーの削減を反映して、性能見積を修正した。これにより通信時間が減少し、基本設計に対して性能が向上し、消費電力が低下した。
  京と富岳の予測性能を表 4.95 に示す。

表 4.95: RSDFT の性能予測結果 (詳細設計 (1)：パラメータ修正・コデザイン検討課題 (3) の 1) 反映後)

機種		課題実行	京倍率	1SCF 時間 [秒]			実行効率	電力効率
	電力制御	[日]	[倍]	全体	演算	通信	[%]	[GF/W]
京	-	126.1	-	4538.3	3783.0	755.3	69.6%	-
富岳	電力制御なし	4.3	29	1217.4	770.1	447.3	48.0%	8.6
(通常モード)	メモリ電力制御	4.5	28	1285.5	838.3	447.3	45.4%	9.3
富岳	電力制御なし	4.0	31	1145.0	697.7	447.3	45.9%	6.9
(ブーストモード)	メモリ電力制御	4.5	28	1280.8	833.5	447.3	41.1%	6.9

- 課題実行性能（詳細設計 (2)：2016 年度ハードウェアパラメータ見直しによる予測）
  2015 年度の構成 (HBM) に対して、各レイテンシの見直しがあり、性能見積を行った。
  京と富岳の予測性能を表 4.96 に示す。

表 4.96: RSDFT の性能予測結果 (詳細設計 (2)：2016 年度ハードウェアパラメータ見直し後)

機種		課題実行	京倍率	1SCF 時間 [秒]			実行効率	電力効率
	エコモード	[日]	[倍]	全体	演算	通信	[%]	[GF/W]
京	-	126.1	-	4538.3	3783.0	755.3	69.6%	-
富岳	無効	4.3	30	1228.2	788.5	439.7	47.6%	9.0
(通常モード)	エコ&FLA only	6.3	20	1792.2	1352.5	439.7	32.6%	8.2
富岳	無効	4.1	31	1153.9	714.2	439.7	45.6%	7.5
(ブーストモード)	エコ&FLA only	5.8	22	1656.9	1217.2	439.7	31.7%	7.1

- 課題実行性能（詳細設計 (3) 開発期間 I：2017 年度前半評価精度精緻化による予測）

  2017 年度前半の詳細設計 (3) 開発期間 I では、詳細設計 (2) の性能見積もりに対して、評価精度の精緻化を行った。評価精緻化の内容は以下の 4 項目である。

  (a) TGS（従来法）のアルゴリズム変更に伴う通信の隠蔽効果
  (b) TGS（従来法）のアルゴリズム変更に伴う処理ブロック数の変化

(c) SD で使用される対角化外部ライブラリ（EigenExa）の演算性能の精緻化
(d) SD で使用される対角化外部ライブラリ (EigenExa) の通信の集約効果

各項目の詳細な評価精緻化の内容を以降にて記述する。

(a) TGS（従来法）のアルゴリズム変更に伴う通信の隠蔽効果
詳細設計 (2) までの性能評価と並行して、第 4.8.3 節に記したように、GS、SD、CG などの主要な評価区間のメモリ分散化を行ってきた。メモリ分散化の導入に伴い、今までプロセス間で共有していた密度汎関数などの行列情報を計算の前に収集し、計算の結果を分配する必要があるため、新たな通信の発生が予想される。但し TGS では、計算結果の分配を従来から行っており、通信量・回数に差異はない。
また、TGS については、第 4.8.3.2 節で評価した新規アルゴリズムである対角優先の手法を採用することにより、通信の隠蔽が可能である。TGS（従来法）では、計算ブロックのリカレンスと対角部の三角行列積計算のインバランスを削減するため、対角部のみ別のプロセスに移動させて計算する手法を採用した（対角移動）。評価では、演算の順序を変更し、先にリカレンスのある処理ブロックを計算することで、他のブロックを計算している間に計算結果を ibcast で通信することが可能になり、通信を隠蔽できる（対角優先）。リカレンスの概要を図 4.86 に示す。
詳細設計 (3) では、詳細設計 (2) までの従来手法の TGS（対角移動）の評価を第 4.8.3.2 節で評価した新規手法（対角優先）への変更による通信時間の差異を評価した。ibcast による計算結果の配布は初回の計算など一部通信については隠蔽できない。第 4.8.3.2 節で評価した京での実測で通信時間は、91.9%〜94.1%隠蔽されており、ここでは 90%通信時間が隠蔽できるとして評価を行った。
またバンド方向の ibcast 通信により、空間方向の allreduce 通信の帯域が低下する可能性がある。京での実測結果によると空間方向の allreduce 通信は処理ブロックにより異なり、対角ブロックで 0.0%〜0.4%通信時間が増加し、非対角ブロックでは 13.7%〜14.6%通信時間が増大していた。今回の評価では通信時間は、対角ブロックの計算では 1.0%、非対角ブロックの計算では 15%、空間方向の allreduce 通信時間が増大すると仮定した。これらの効果を総合して、対角優先アルゴリズムによる通信隠蔽効果により、 1SCF あたり 36.7[s] 減少し、全体性能へ与える影響は +2.98% であった。

(b) TGS（従来法）のアルゴリズム変更に伴う処理ブロック数の変化
TGS における新規手法（対角移動）の採用により、対角・非対角の各処理ブロックの計算順序が変わり、各プロセスの担当する処理量が変わる。実際の処理例を図 4.87 の通りである。ランク#0 が処理する対角ブロック数は、

$$(2 \cdot M_b/M_{blk}/np_b - 1) \rightarrow M_b/M_{blk}/np_b \tag{4.10}$$

と書き下せ、ターゲット問題では対角処理ブロック数は 95 → 48 に減少する。またランク#0 が処理する非対角ブロック数は、

$$M_b/M_{blk}\dot{(}M_b/M_{blk}/3 - 1)/2 \tag{4.11}$$

と書き下せ、ターゲット問題では非対角処理ブロック数は 3,384 で変わらない。ま

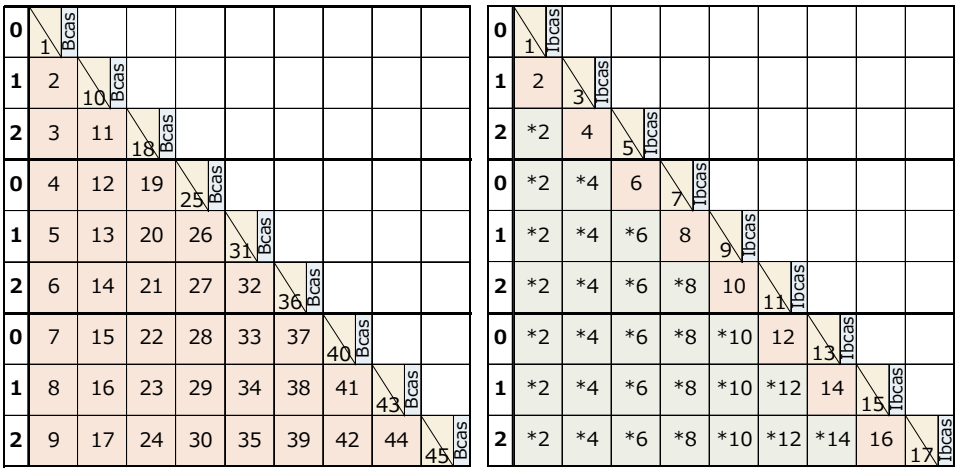


図 4.86: TGS の従来方式（対角優先）と新規方式（対角移動）での処理順序。

た対角処理ブロックの待ち回数は、

$$M_b/M_{blk} - (2 \cdot M_b/M_{blk}/np_b - 1) \rightarrow M_b/M_{blk}/np_b \cdot (np_b - 1) - 1 \qquad (4.12)$$

と書き下せ、ターゲット問題では対角処理ブロックの待ち回数は 49 → 95 に増加する。但し、M_b はバンド数、M_{blk} はブロックサイズ、np_b は、バンド分割数である。これらの効果を総計して、 1SCF あたり全ブロックの処理時間は 3.1[s] 減少し、全体性能へ与える影響は+0.25%である。

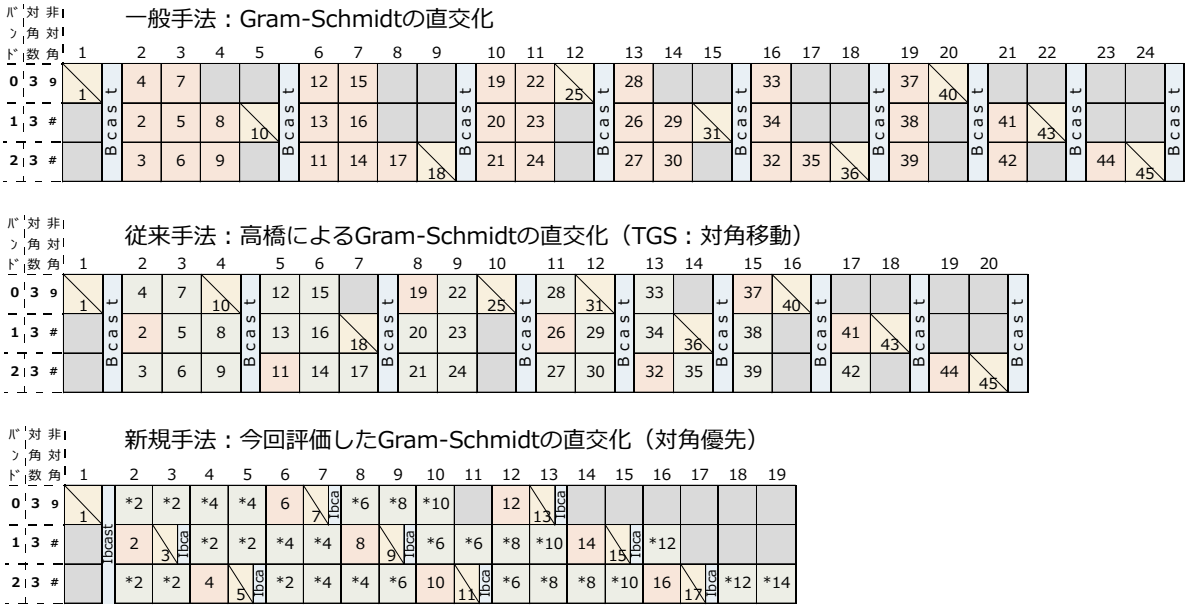


図 4.87: 一般的な GS 手法、TGS 従来方式（対角優先）、TGS 新規方式（対角移動）のプロセスごとの処理順序の例。

(c) SD で使用される対角化外部ライブラリ（EigenExa）の演算性能の精緻化
RSDFT では、対角化処理（SD）で外部対角化ライブラリを使用しており、RSDFT の性能見積もりでは EigenExa を用いて性能を評価している。外部ライブラ

リ EigenExa は、RSDFT 本体のようにカーネルを切り出しているわけではない。詳細設計 (2) での見積もりでは、EigenExa ライブラリの 3 区間について、演算は $O(N^3)$ で変化するとし、富岳での性能は、京のピーク性能が達成できると仮定して、ターゲット問題の性能を評価した。

詳細設計 (3) では、上記 3 区間について、ターゲット問題と同じ演算量になるように 1 プロセスの問題を設定し、FX100 にて実測した結果を性能電力予測ツールで処理して性能評価した。本手法により予測される演算性能は、 20.3%→ 16.6%に低下した。 このため、1SCF あたりの実行時間は 10.8[s] 増加し、全体へ与える性能への影響は-0.88%であった。

また今まで、固有値ライブラリは電力推定対象外区間であったが、新たに消費電力の評価を行うことで他の区間に比べて電力を多く消費することが判明した。 最大電力が 3〜4W 程度、平均電力が 1W 程の上昇を見込んでいる。これは EigenExa のうち三重対角化の処理を行う区間（TRD-BLK）の、メモリビジー率が 46.9%と高く、メモリ性能律速になっているためと考えられる。

(d) SD で使用される対角化外部ライブラリ (EigenExa) の通信の集約効果

詳細設計 (2) までの評価では、外部対角化ライブラリ EigenExa の評価には、2011 年の Gordon-Bell 賞に受賞時の eigen_s ライブラリ測定結果から通信種、通信回数、通信量をカウントし、その際導出した見積もり式からターゲット問題での通信回数、通信量をから通信時間を算出した。eigen_s ライブラリは、京並びに富岳の開発と並行して、EigenK、EigenExa として、開発並びに性能チューニングが進んだ。特に三重対角化区間 TRD-BLK については、通信の集約による通信回数の削減が試みられ、通信レイテンシの削減などによる高性能化が進められた。

詳細設計 (3) では、集約された通信箇所について、通信量と通信回数を算出のし直しを行った。 この結果、通信時間は 1SCF あたり 13.3[s] 減少し、全体へ与える性能への影響は+1.08%であると評価された。

以上の評価精度の向上により、 通常モードエコモード無効での京に対する性能倍率は 30 倍→ 31 倍に向上した。京と富岳の予測性能を表 4.97 に示す。

表 4.97: RSDFT の性能予測結果 (詳細設計 (3) 開発期間 I：2017 年度前半性能評価精緻化による予測)

機種	エコモード	課題実行 [日]	京倍率 [倍]	1SCF 時間 [秒] 全体	演算	通信	実行効率 [%]	電力効率 [GF/W]
京	-	126.1	-	4538.3	3783.0	755.3	69.6%	-
富岳	無効	4.1	31	1185.6	795.9	389.7	49.2%	9.0
(通常モード)	エコ&FLA only	6.0	21	1719.0	1329.3	389.7	33.9%	8.2
富岳	無効	3.9	33	1111.2	721.4	389.7	47.3%	7.5
(ブーストモード)	エコ&FLA only	5.6	23	1586.7	1197.0	389.7	33.1%	7.1

- 課題実行性能 (詳細設計 (3) 開発期間 II：2017 年度後半 EigenExa 逆変換の DGEMM 評価版 2 を用いた性能評価)

2017 年度後半の詳細設計 (3) 開発期間 II では、開発期間 I での 2017 年度前半評価精度精緻化による予測に対して、以下の評価を行った。

(a) 性能電力予測ツールの更新 (2018/1/4 版)

(b) EigenExa の逆変換 (TRDBAK) ブロックの DGEMM 評価版 2 を用いた再評価

各項目の詳細な評価内容を以降にて記述する。

(a) 性能電力予測ツールの更新 (2018/1/4)
性能電力予測ツールのアップデートに伴い、アプリ性能推定を更新した。アップデートの概要は、電力推定のパラメータを、シミュレータによる数値に置き換えたというものである。 ツールのアップデートに伴い、性能に変化はなく、システム全体の最大電力は-1〜0MW の変動であった。ノード電力は微増傾向であるが、全体電力は微減傾向である。内訳を調べると、DGEMM 区間が数 W 増えているが、その他の区間で下がったためと解釈できる。

(b) EigenExa の逆変換 (TRDBAK) ブロックの DGEMM(試作版 2) を用いた再評価
詳細設計 (3) 開発期間 II では、EigenExa の逆変換 (TRDBAK) ブロックの再評価を実施した。これまでの TRDBAK(逆変換) は主要な処理は DGEMM であり、FX100 の製品版での DGEMM を用いて評価を行っており、Intel と比較して性能が低く、効率は 35%程度であった。この性能は、エコモードで性能が劣化しない効率 (50%以下) であった。EigenExa の逆変換で用いられる主要な DGEMM サイズは、NM のサイズは 16x16、64x128 であり、K は約 32 種類変わり、最大で 10,549 の扁平な形状であった。このため富士通が開発した、細長い行列で性能向上が見込まれる DGEMM(試作版 2) を用いて再度性能の評価を行った。今回は、特徴的な 2 パターン (図 4.98) について、ライブラリでの実行時間から性能の評価を行った。なお、DGEMM の呼び出し回数は、thread 数の違いにより京と FX100 で異なる。評価結果を図 4.99 に示す。最新ライブラリで EigenExa の逆変換ブロックの富岳の推定性能は全体で、 32%→ 35%と 1 割程度向上した。FX100 での性能は京と比べて 60%→ 52%へと下がるが、扁平形状によるメモリ要求 BF 値が高いためと考えられる。富岳での推定性能が、FX100 に比べて、 更に 52%→ 28%へと低下した。この性能低下については、今後 DGEMM 専用モードを用いた評価精度の向上は実施せず、実機で DGEMM ブロックサイズなどのチューニングを行うことで性能改善を行うこととした。EigenExa の逆変換では、PARALLEL リージョン内で逐次版の DGEMM を呼び出している。現在、数値ライブラリ WG にて、EigenExa の逆変換において thread 並列版 DGEMM を使用する試作を行なうことになった。性能電力予測ツールの性能評価値に、12thread でプリフェッチが働くように調整された DGEMM(試作版 1) を用いて性能評価値を更新した。 この更新により、実行時間の変動は 1 秒以内であり、消費電力に変化はなかった。

表 4.98: EigenExa の逆変換 (TRDBAK) ブロックで使用する代表的な DGEMM パラメータ

DGEMM 区間	呼び出し回数	TRANSA	TRANSB	M	N	K	ALPHA	lda	ldb	BETA	ldc
DGEMM1	165	T	N	128	64	10549	1	10576	10576	0	128
DGEMM2	36	T	N	16	16	10549	-1	10576	10576	0	10576

以上の評価精度の向上により、通常モードエコモード無効での性能は 31 倍であり、消費電力は平均で 25MW, 最大で 29MW であった。京と富岳の予測性能を表 4.100 に示す。

表 4.99: EigenExa の逆変換 (TRDBAK) ブロックの試作版 DGEMM による性能

機種	京		FX100						富岳	
ｽﾚｯﾄﾞ数	8		12						12	
区間	製品版		製品版		試作版 1		試作版 2		試作版 2	
	時間 [ミリ秒]	効率	時間 [ミリ秒]	効率	時間 [ミリ秒]	効率	時間 [ミリ秒]	効率	時間 [ミリ秒]	効率
DGEMM1	384.1	59.5%	163.7	46.6%	156.0	49.5%	148.9	52.0%	111.7	37.3%
DGEMM2	5.3	33.1%	3.0	17.8%	2.9	18.6%	2.3	23.0%	2.1	13.6%

表 4.100: RSDFT の性能予測結果 (詳細設計 (3) 開発期間 II：2017 年度後半性能評価による予測)

機種	エコモード	課題実行 [日]	京倍率 [倍]	1SCF 時間 [秒] 全体	演算	通信	実行効率 [%]	電力効率 [GF/W]
京	-	126.1	-	4538.3	3783.0	755.3	69.6%	-
富岳	無効	4.1	31	1186.4	796.6	389.7	49.2%	8.9
(通常モード)	エコ&FLA only	6.0	21	1719.8	1330.1	389.7	33.9%	8.2
富岳	無効	3.9	33	1111.8	722.1	389.7	47.2%	7.5
(ブーストモード)	エコ&FLA only	5.6	23	1587.4	1197.6	389.7	33.1%	7.2

- 課題実行性能 (詳細設計 (4)：2018 年度 DGEMM での実機評価並びにシステム構成変更に伴う性能評価)

  2018 年度の詳細設計 (4) では、詳細設計 (3) 開発期間 II での評価に対して、以下の効果について影響の見積りを実施した。

  (a) システム構成変更による性能への影響
  (b) DGEMM の試作機 CPU を用いた評価結果の取り込み

  以下に、それぞれの見積もりの詳細を記す。

  (a) システム構成変更による性能への影響
  詳細設計 (4) 期間中、クロック数とノード数を変更するシステム変更案が提案され、性能評価を行った。システム構成案は、現行構成案、周波数を上げその分、y 軸方向にノード数を 5%削減、10%削減、15%削減する案である。WG では、これらの構成案に対して、ターゲット問題遂行に如何なる影響があるか、性能にどのようなインパクトがあるか技術駅な討議を行った。
  RSDFT の性能インパクトについては、演算の 9 割を占める gemm は効率はほぼ下らず、処理能力は周波数分高速化している。但し、全体の 1/3 を占める通信については、周波数による性能向上の効果が現段階では評価できていない。台数効果を含めて、全体的に通信による影響により、3%程度性能が低下すると推定された。
  以上の評価システム並びに評価手法の見直し伴い、通常モードエコモード無効での性能は 31 倍となり、消費電力は平均で 26MW, 最大で 30MW となった。京と富岳の予測性能を表 4.102 に示す。

  (b) DGEMM の試作機 CPU 測定を用いた評価結果の取り込み
  詳細設計 (4) では、DGEMM 性能について試作機 CPU を用いた実行性能の測定を行った。測定結果を表 4.101 に示す。FX100 の測定では、20150630_dgemm_12core のバージョンの DGEMM ライブラリを用いて、12 コアを使用してピーク性能比を算出した。富岳での実測値では、社内向け評価版 DGEMM ライブラリを使用した

測定値である。この測定ライブラリは、スレッド並列の再分割や配列コピーなどのキャッシュスラッシング回避などのチューニングが施されている。

表 4.101: DGEMM の試作機 CPU 測定性能

アプリケーション	区間名	演算効率(%)						経過時間比		パラメータ				
		京	FX100	富岳				(2.0GHz/1.8GHz)	(2.2GHz/2.0GHz)	TRANS		M	N	K
				予測ツール	実機					A	B			
				2.0GHz	1.8GHz	2.0GHz	2.2GHz							
RSDFT	id00000001	89.0%	88.3%	78.3%	84.5%	84.2%	83.7%	0.90	0.91	T	N	1536	1536	4913
	id00000002	84.4%	80.5%	68.1%	76.4%	75.8%	75.4%	0.91	0.91			768	768	4913
	id00000003	80.5%	75.6%	62.1%	69.7%	69.3%	68.7%	0.90	0.92			384	384	4913
	id00000004	73.2%	53.1%	39.0%	53.3%	53.2%	53.0%	0.90	0.91			192	192	4913
	id00000005	58.2%	26.5%	17.6%	35.7%	35.7%	35.7%	0.90	0.91			96	96	4913
	id00000006	22.9%	19.2%	11.9%	23.3%	23.2%	23.3%	0.90	0.91			48	48	4913
	id00000007	6.0%	14.2%	8.1%	13.7%	13.7%	13.6%	0.90	0.92			24	24	4913
	id00000008	1.7%	10.4%	5.6%	5.4%	5.4%	5.4%	0.90	0.90			12	12	4913
	id00000011	87.3%	89.9%	80.4%	88.9%	88.6%	88.5%	0.90	0.91	N	N	4913	1536	1536
	id00000012	86.2%	87.8%	77.2%	84.1%	83.7%	83.9%	0.90	0.91			4913	768	768
	id00000013	83.4%	83.0%	70.5%	80.6%	80.3%	80.3%	0.90	0.91			4913	384	384
	id00000014	75.6%	70.1%	55.5%	70.9%	70.4%	70.5%	0.91	0.91			4913	192	192
	id00000015	60.6%	69.6%	54.6%	51.4%	50.4%	50.1%	0.91	0.92			4913	96	96
	id00000016	26.4%	69.3%	54.4%	46.1%	46.8%	48.1%	0.90	0.88			4913	48	48
	id00000017	11.2%	51.1%	37.2%	28.2%	27.9%	27.9%	0.90	0.91			4913	24	24
	id00000018	7.2%	33.5%	22.0%	12.6%	12.6%	12.7%	0.90	0.91			4913	12	12

京の効率と比べて FX100 では、サイズが小さい DGEMM で効率が上がっているもののサイズが大きい DGEMM で効率が下がっている。予測ツールでは、この実測での結果をもとに性能を算出しており、富岳の予測ツールによる性能は FX100 での効率と比較して、10%程度低下すると評価された。今回の試作機 CPU を用いた測定では、その効率低下が数%程度に抑えられることがわかった。予測ツールに比べて実測値が概ね良いのは、予測ツールでの推定が少し安全方向に見積もられていたことに起因する。FX100 での性能に比べて、小さいサイズの DGEMM 性能が下がる理由は、SIMD 幅などが増えることにより、小さいサイズで有効に SIMD 幅を使えないためと推察される。

今まで RSDFT での性能推定では、予測ツールを用いた推定値を使用して算出していた。NN タイプのサイズの小さい DGEMM を除いて、実行時間の大部分を占めるサイズの大きな DGEMM については、試作機 CPU を用いた実測値は予測ツールを用いた見積もり値と比べて性能が向上しているため、RSDFT アプリの推定性能は向上する。試作機 CPU を用いた性能推定値は表 4.102 の通りである。

表 4.102: RSDFT の性能予測結果 (詳細設計 (4)：2018 年度性能評価による予測)

機種	エコモード	課題実行 [日]	京倍率 [倍]	1SCF 時間 [秒] 全体	演算	通信	実行効率 [%]	電力効率 [GF/W]
京	-	126.1	-	4538.3	3783.0	755.3	69.6%	-
富岳	無効	4.1	31	1111.7	722.1	389.6	47.2%	8.9
(通常モード)	エコ&FLA only	6.0	22	1587.3	1197.6	389.6	33.1%	8.3
富岳	無効	3.9	33	1050.8	661.1	389.6	45.5%	7.9
(ブーストモード)	エコ&FLA only	5.6	24	1478.9	1089.3	389.6	32.3%	7.6
試作機 CPU 測定	無効	-	33	-	-	-	-	-

- 課題実行性能 (ソフトウェア調整 (1)：2019 年度の性能評価)

  ソフトウェア調整 (1) での課題実行性能は、詳細設計 (4) から更新はない。

- 課題実行性能 (ソフトウェア調整 (2)：2020 年度の性能評価)

  ソフトウェア調整 (2) での課題実行性能は、以下の性能評価を実施した。

(a) 測定上の問題
(b) 性能チューニング
(c) 最終性能

以下に、それぞれの評価の詳細を記す。

(a) 測定上の問題

ターゲット問題の評価にあたっては、いくつかの問題があった。当初想定していた問題規模は、プロセスあたりのグリッド数が 17x17x17 となっており、素数となっているため動作しない。プロセスあたりのグリッド数 17 をターゲット問題として選んだ理由は、2011 年の GB で用いた計算に近い条件で、かつ DGEMM のキャッシュスラッシングを回避するために設定された問題であった。grid の分解能は 16〜18 で物理的な収束性は変わらないことは確認済みである。動作しない原因は FFT 処理に使用していた ffte ライブラリの制限によるものであり、fft のサイズが想定する基数で割り切れない問題と、さらに 3 次元 FFT における他軸分割でも割り切れる必要があるという制限によるものであった。プロセスあたりのグリッド数 16,18 を用い、混在情感下での評価やプロセスマッピングの回転をすることで制限緩和が可能であるが、前者はメモリ不足、後者は性能劣化が想定された。本問題に用いる FFT 処理に FFTW を用いることで、本問題は回避した。また、プロセスあたりのグリッド数を 17 にすることで、NN タイプの DGEMM 性能においてキャッシュスラッシングが発生しなくなり [表 4.103]、性能が約 4%向上することが確認できた。

表 4.103: プロセスあたりのグリッド数 16, 17 での DGEMM カーネルの実行時間の比較。

区間		DGEMMサイズ			Elapsed [sec]		区間		DGEMMサイズ			Elapsed [sec]	
		M	N	K	grid/np=16	grid/np=17			M	N	K	grid/np=16	grid/np=17
TN	1	1536	1536	4913	3.059	3.614	NN	11	1536	1536	4096	3.503	3.366
	2	768	768	4913	0.887	1.044		12	768	768	4096	1.020	0.929
	3	384	384	4913	0.273	0.309		13	384	384	4096	0.283	0.251
	4	192	192	4913	0.117	0.112		14	192	192	4096	0.101	0.085
	5	96	96	4913	0.048	0.052		15	96	96	4096	0.045	0.040
	6	48	48	4913	0.022	0.023		16	48	48	4096	0.023	0.023
	7	24	24	4913	0.016	0.016		17	24	24	4096	0.019	0.018
	8	12	12	4913	0.014	0.014		18	12	12	4096	0.014	0.014

(b) 性能チューニング

カーネルとして抽出していない「その他の区間」のうち calc_fermi 処理のコストが、ターゲット問題の評価で目立つようになったため対応を行った。ループ内で行われていた malloc 処理をループ外に出すことで、ターゲット問題で 110[s] の処理が 1[s] 以下になった。また、いくつかのパラメータについてサーベイを実施し、最速のパターンを見積もった。性能に寄与するコントロールパラメータはブロッキングサイズ NBLK、並びに CG のまとめ上げ処理を行う MB_d などが挙げられる。NBLK=1,536, MB_d=512 にて最速で、1,069[s] ほどとなった。NBLK パラメータは見積もり時に設定したパラメータと同じであり、また MB_d はメモリ量の制限から見積もり時には 8 と設定されていた。更にプロセスマッピングを band=4 に変更することで、band 方向のコミュニケータを CMG 内に割り当てることで、バ

ンド方向の MPI_Bcast 通信が高速化し、1,068[s/SCF] → 933.7[s/SCF] へと向上した [表 4.104]。

表 4.104: バンド分割を不連続な 3 分割から CMG 内の 4 分割に変更した時の性能の変化。

				実測		高速化率
分類	大区間	小区間	種別	nband=3	nband=4	b4効果
calc	sd	mate	gemm	96.34	96.89	0.99
	sd	rotv	gemm	228.37	185.54	1.23
	gs	gs	gemm	98.53	135.98	0.72
comm	sd	mate	bcast	10.82	0.38	28.77
	sd	mate	reduce	12.23	7.88	1.55
	sd	mate	exchange	0.09	0.13	0.70
	sd	rotv	bcast	39.16	21.18	1.85
	gs	gs	allreduce	156.23	67.74	2.31
	gs	gs	bcast	0.00	0.00	
	cg	cg	allreduce	62.25	47.78	1.30
other	cg	hamiltonian	other	7.50	5.41	1.39
	cg	other	other	13.52	10.11	1.34
	cg	bcset1	other	31.61	23.51	1.34
	eigen	all	all	157.00	275.01	0.57

(c) 最終性能

本 WG のターゲット問題である、「富岳の 10,368 ノードを用いて、Si 原子数 11 万・バンド数 22 万・SCF200 回をケース数 24 で同時実行 (Capacity 問題) での構造最適化のための SCF 計算」で、見積りの 33 倍に対し、37.5 倍の高速化を達成した。1 分分解能の電力計を用いて、フルラックに換算した電力は、実利用で想定する電力モード (ブーストモード、エコなし) で、最大消費電力 29.9MW、平均消費電力 25.6MW であった。表 4.105 の電力効率などの値は、1 分解能の電力計実測値から換算した全系での電力を用いて算出した値である。他 WG での重点アプリ電力やジョブ統計情報から算出した全アプリ一覧表中の電力値とは評価方法が異なるため、注意が必要である。見積りと実測の差は、主要演算の DGEMM は同等以上、通信は GS カーネル内の allreduce が遅くなったが、それ以外の集団通信は高速化していた。また 107.7[s] ほどの CG の実測時間に、見積もり時は演算時間が含まれておらず、80[s] ほど演算時間が増えている。見積もり時のプロセスマッピングでは、不連続なコミュニケータ形状であったバンド方向の分割を、測定では CMG に割り当てる事でバンド方向の通信が大幅に高速化した効果が大きい。

表 4.105: RSDFT の性能予測結果 (ソフトウェア調整 (2)：2020 年度実機による最終測定性能)

機種	エコモード	課題実行 [日]	京倍率 [倍]	1SCF 時間 [秒] 全体	実行効率 [%]	電力効率 [GF/W]
京	-	126.1	-	4538.3	69.6%	-
富岳	無効	3.6	35	1003.6	43.7%	8.5
(通常モード)	エコ&FLA only	5.1	25	1396.1	37.6%	9.6
富岳	無効	3.4	38	926.7	43.1%	8.1
(ブーストモード)	エコ&FLA only	4.6	28	1279.3	37.3%	9.4

4.8.6 富岳で達成された性能倍率

上述したコデザインの成果を反映して富岳で達成された京に対する性能倍率及び消費電力を表 4.106 に示す。

表 4.106: RSDFT の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
RSDFT	38 倍	30 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

表 4.107: RSDFT の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
RSDFT	38 倍		35 倍		28 倍		25 倍	
	25 MW	28 MW	22 MW	25 MW	20 MW	20 MW	17 MW	18 MW

4.9 重点課題 8 FrontFlow/blue のコデザイン

4.9.1 アプリケーションの特徴とコデザインの方針

本重点課題 8 は我が国の将来のものづくりに変革をもたらす計算科学技術の開発を目的としており、コンピュータシミュレーションによる製品そのものの性能の変革と、製品の開発プロセスの効率化などの変革、さらに産業界などの実際のものづくりの現場への普及を含んでいる。

ものづくりの分野ではいずれのアプリケーションも解析対象として様々な形状を取り扱わなければならず、非構造格子を用いるアプリケーションが多い。非構造格子を用いる場合、参照したいデータのメモリ空間上での位置を単純な規則から求め、その位置にあるデータをメモリから転送することが出来ない。参照したいデータの位置を求めるには、隣接関係を記述したリストの参照を介する必要があり、メモリ転送量が増えるという傾向があるため、要求 B/F が高く、アプリケーションのコアとなる部分の処理が類似しているという特徴がある。

FrontFlow/blue(または FFB) は非構造格子を用いるアプリケーション群を代表するアプリケーションとして位置づけられており、非構造格子を用いるアプリケーションで頻出する、リストを介したインダイレクトアクセスのために要求 B/F が高く、高いメモリスループットを要求する計算の性能向上の観点でコデザインに寄与する。富岳上で FrontFlow/blue が高速に動作できれば、その技術を他アプリケーションに水平展開することが容易となる。

これまでに京で 4 万ノード程度を用いて、最大 320 億要素規模の流れ解析の実行実績を有している。富岳を用いて、数千億要素規模の計算が実現できれば、レイノルズ数 10^7 規模の乱流の準直接数値計算が可能となり、水力機械の内部流れや、実車の空力の乱流計算においてループ試験や風洞実験と同等の精度を持つ流れ解析が実現する。また、この高精度な解析結果をリファレンスデータとして用いて、1 万個体 (100 個体 $\times$100 世代) の多目的設計最適化計算を富岳では数日で実行できるようになる。従来よりも高精度・広範囲な多目的設計最適化技術が実現でき、ターボ機械などの関連するものづくり設計の高度化に貢献できる。また、この問題規模は重点課題 8 内において最大の規模であるため、課題内の他のテーマにおけるターゲット問題の遂行の目途が立つ。

FrontFlow/blue は有限要素法に基づく流体シミュレーションコードである。行列は疎行列であり、非構造格子を用いるためメモリアクセスのパターンは入力データにより異なるランダムアクセスとなる。

FrontFlow/Blue における流れ場の計算は、移流・拡散項の計算により速度予測子を求め、速度予測子から圧力を求め、圧力から速度予測子を修正することで速度を決定するという手順で行われる。速度予測子と圧力の求解は陰的に行われる。陰解法ソルバーとして共役勾配 (CG) 法を用いており主要な演算は疎行列とベクトルとの積である。また、はっきりと処理内容が疎行列ベクトル積であると判るように実装されていないカーネル、例えば後述する勾配計算ルーチン (GRAD3X) および発散計算ルーチン (FLD3X2)、および運動方程式ルーチン (VEL3D1) についても、要素単位で定義される行列から全体行列を生成することこそ行っていないが、要素の隣接情報と各要素で定義される小行列を用いて全体行列に対応する値を計算する、いわゆる Element by Element の手法を用いているため、演算カーネルについては基本的に全て疎行列とベクトルの積であると言える。

疎行列とベクトルの積は主としてメモリスループットに依存する処理であり、有限要素法が前提としているデータ構造に起因して、メモリアクセスが不連続アクセスとなるため、行列値のみならず、コネクティビティ情報 (節点 n の隣接節点のリスト、節点 n の上位要素のリスト、要素 e の参照節点リストなど) もロードする必要があり、メモリバンド幅への要求が高くなる特徴がある。

そのため、性能改善はメモリスループットを向上させることが主眼となる。このことを念頭において以下の各事項についてコデザイン検討を行った。

- 勾配計算ルーチン (GRAD3X) 主要ループ
- 発散計算ルーチン (FLD3X2) 主要ループ
- 運動方程式ルーチン (VEL3D1) 主要ループ
- 疎行列ベクトル積ルーチン (CALAX) 主要ループ
- 配列のコピー
- 配列のゼロクリア
- 非スレッド並列ルーチン (CRSCVA, CLRCRS, NODLEX, DGNSCL)
- if を含むループのリストベクトル化
- 型変換ループの除去

上記各事項の具体的な内容については次節以降に示す。なお、以降に示すコデザインによる処理時間短縮の値は、特段の記載が無い場合は、現状は全て京でテスト問題を実行した際の実測値であり、一部富岳での測定値があるものについては明記する。

4.9.2 勾配計算ルーチン (GRAD3X) 主要ループの SIMD 化・ソフトウェアパイプライン化、ロードストア数削減を目的としたアルゴリズム変更

GRAD3X では要素の中央で定義されている圧力の値の勾配 (∇p) を節点上に求める。要素でループを回し、各要素の圧力値に係数を乗じた結果を、要素が参照している節点へと足し込む処理を行う。

図 4.88 は、本処理の模式図である。図中の四角い枠は要素を表し、黒丸は節点を示している。各要素は中心に定義された圧力 P_n と、頂点の個数に応じた数の係数 $C_{n,v}$ を持つ。例えば要素 1 は頂点として節点 1,2,12,11 の 4 つの節点を参照しており 4 つの係数 $C_{1,1}$,$C_{1,2}$,$C_{1,3}$,$C_{1,4}$ を持っている。矢印は圧力 P_n と係数 $C_{n,v}$ を乗じた値を対応する節点に足し込む処理を意味している。要素ごとに処理を行っているため、特定の節点への足し込みが不規則な間隔で複数回発生する。この図の例では図中赤丸で囲まれた節点 12 への足し込みは要素 1, 2, 11, 12 を処理する際の合計 4 回発生している。この足し込みの回数と発生間隔は入力データに依存したランダムなものである。複数の要素から同じ節点の値を更新するため、データの依存性がありこのままではスレッド並列化、SIMD 化、ソフトウェアパイプライン化が出来ない。

そこで本実装では計算領域はプロセス並列のための領域分散に加えてプロセス内でも領域分割を行いデータ依存性を回避している。図 4.89 は、本処理の模式図である。各プロセスが処理する領域は、各々がほぼ同程度の要素を持つ複数のブロックに分割され、互いに隣接しないブロックが同一のカラーに含まれるようカラーリングを行い、各カラー内でブロック番号が連続となるように番号の並び替えられている。図では示されていないが、要素番号についても各ブロック内で連続となるように並び替えられている。図下段が最終的なデータ構造となっており、各カラー内のブロックは互いに隣接していないのでデータ依存性がない一方で、各ブロック内の要素について

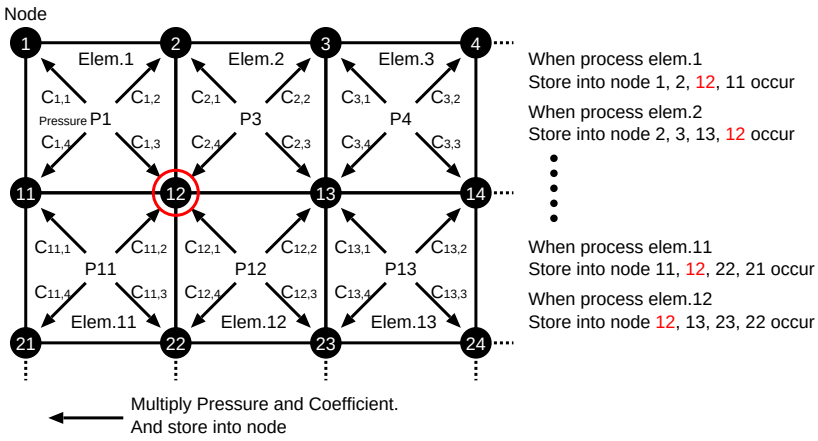


図 4.88: GRAD3X の処理内容とストアの様子

は互いに隣接しているためデータ依存性がある。ループ構造は三重ループとなっており、最外のループがカラーを、中間のループがカラー内のブロックを、最内のループがブロック内の要素で回る。スレッド並列は互いに隣接しておらずデータ依存性がないブロックで回る中間ループで施されている。最内ループは互いに隣接した要素群で回るループのため、データ依存性があり、SIMD 化及びソフトウェアパイプライン化を適用することはできない。このようなブロック分割を用いず、要素を直接カラーリングする実装ではループ構造がカラーを回すループと、カラー内の要素を回すループの二重ループ構造となり、最内ループでのデータ依存性がないため、最内ループでスレッド並列化、SIMD 化及びソフトウェアパイプライン化を適用できるのであるが、メモリアクセスの局所性に乏しく再利用性のあるデータを再利用する時間的密度が粗く、結果としてキャッシュを効率的に利用できず性能が出ていなかった。このループの処理はメモリスループットに強く依存しているため、SIMD 化とソフトウェアパイプライン化を適応するよりもメモリスループットを改善した方が得策であるという判断により現在の実装になったのであるが、それでもメモリスループットが十分に出ていなかったため今回再度のチューニングを行った。この処理には全体実行

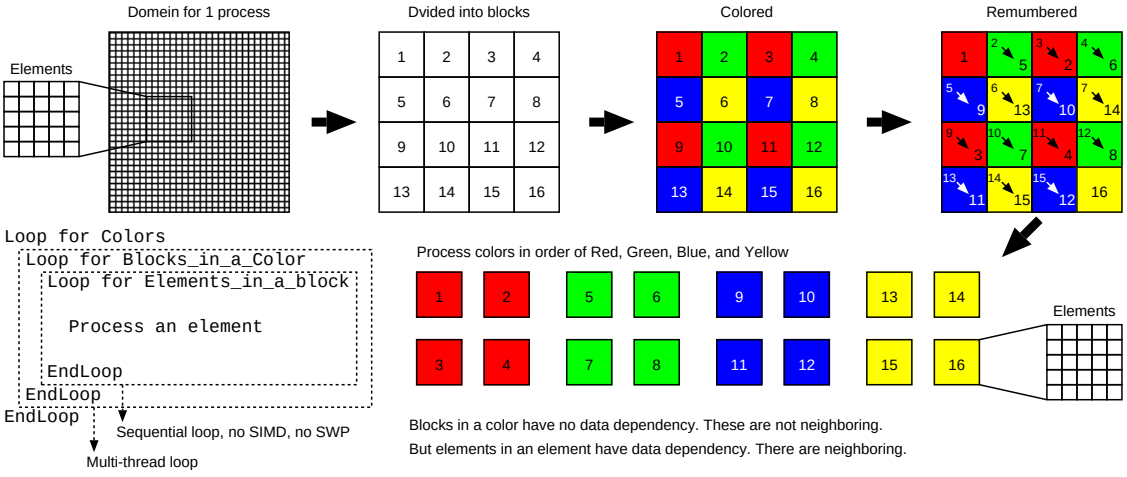


図 4.89: GRAD3X のループ構造

時間 394 秒の内、95 秒を占めており、その時間の多くの割合をメモリアクセス関連の待ちが占めていた。実行命令の内訳を調べるとストアの数が多かった。これは複数の要素から参照されている節点へのストアを要素で回るループで処理すると、同一節点へのストアが何度も必要になるという上述した特性に起因していた。そこでループ構造を節点で回るよう変更した。この実装では、節点ごとにその節点を参照している全要素からの寄与を一度に計算し集約することで各節点へのストアが 1 回だけとなるように変更した。図 4.90 はループ構造変更の前後でのストアの様子の模

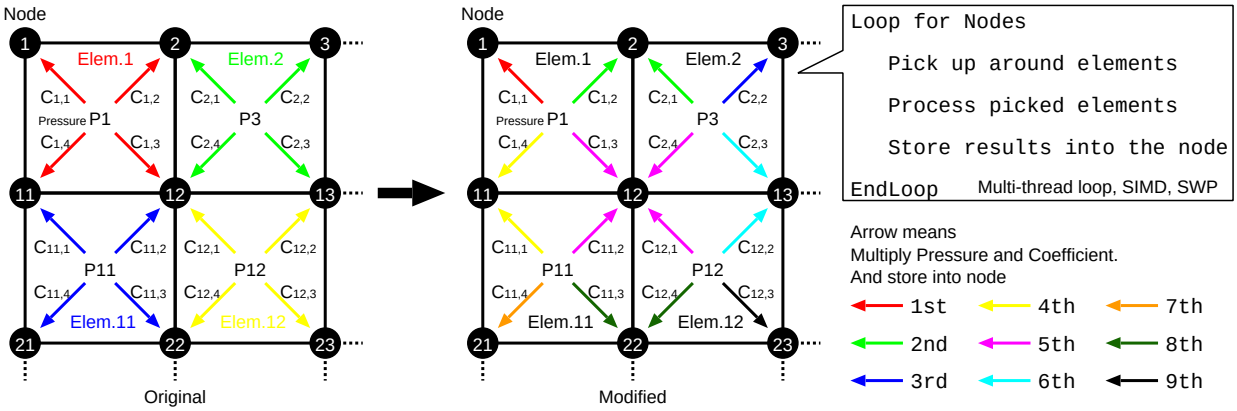


図 4.90: ループ構造変更前後のストアの様子

式図と、変更後の模擬ソースコードを示したものである。変更前では、処理は要素の順番で行われるので、要素 1 を処理する際には節点 1,2,11,12 へのストアが発生し、要素 2 の処理時には節点 2,3,13,12 へのストアが発生する。変更後では、処理は節点の順番で行われる。例えば節点 2 を処理する際には、図中緑の矢印で示すように、節点 2 を使っている二つの要素からの節点 2 への寄与をまとめて計算し節点 2 へとストアする。これ以降節点 2 への値の更新は発生しない。またストアの順序がシーケンシャルになったため、データ依存性がなくなるのでスレッド並列化を行うための複雑な構造も不要となり、同時にループには SIMD 化及びソフトウェアパイプライン化も適用できた。結果としてロード・ストア命令数は 42%削減、ストア命令に限れば 87%削減し、実行時間は 95 秒から 41 秒へと向上した。このときのメモリスループットは、23.2GB/s から 47.2GB/s まで向上した。これは京ではほぼ実効上限となる値である。

富岳実機を用いた測定では、同一の処理をおよそ 14 秒強程度の時間で実行出来と推定されている。(ただしメモリの不足から、半分の問題サイズで測定し、その 2 倍としている。メモリ不足については最新版では解決される)。このとき富岳でのモリスループットは 146.3GB/s であった。この値は富岳 1CMG の実効メモリスループット上限である 205GB/s と比してまだ向上する余地があることを示している。本カーネルの実装では、最内ループで、1 次元目のサイズが 3 である配列を A(1,I,J)、A(2,I,J)、A(3,I,J) のように 3 回アクセスしており、メモリ空間上は連続アクセスであるが、コンパイル時にコンパイラが連続であることを把握できず連続 SIMD ロード命令ではなく、不適切なロード命令 (Gather ロード命令) を発行していた。そこでここでは、配列の形状を A(1,I,J) から A(I,J,IP) のように変更することで、連続 SIMD ロード命令が発行されるように改変した。その結果、メモリスループットは 170GB/s にまで向上した。現在コンパイラは A(1,I,J)、A(2,I,J)、A(3,I,J) のようなアクセスパターンでも適切なロード命令 (Structure ロード命令) を発行できるように改良が行われており、コンパイラの適当ができ次第、A(1,I,J) を Structure ロード命令で処理する方が良いのか、A(I,1,J) として連続 SIMD ロード命令で処理する方が良いのか決

定する。

4.9.3 発散計算ルーチン (FLD3X2) 主要ループの SIMD 適用、ストリーム数調整、セクタキャッシュの利用

本アプリはかなり以前から使われているコードであり、ベクトル型スーパーコンピュータ上でバンクコンフリクトを避ける実装として、本来は 8 あれば良い多次元配列の一次元目のサイズを 9 として宣言していたため 8 となるように変更した。このような実装になっていると、キャッシュライン単位のデータ転送のため、実際には使わない 9 番目の配列成分も転送されるためメモリバンド幅の 1/9 が無駄になる。

また、FrontFlow/blue ではサブルーチン中で扱う配列は、全てメインルーチン中で確保したメモリ領域を使うようにしており、配列はそのサイズを示す変数とともにサブルーチンへ渡される。本サブルーチンは一次元目のサイズが 8 である 2 次元配列を受け取るため、コンパイル時にすでにサイズは 8 と確定しているが、配列サイズが変数で指定されているためコンパイラは一次元目のサイズが可変であることを前提とした命令列を生成するため無駄であった。そこで配列サイズが 8 固定である情報をコンパイラに与えるようにした。

本サブルーチンの主要ループはアルゴリズムを実現する最も基本的な実装をした場合では、8 回転する最内ループを持つ 2 重ループ構造となるが、実際には最内ループを 8 展開にアンロールした 1 重ループ構造で実装されていた。2 次元配列へのアクセスでは、ループ 1 回転で 1 次元目の 8 成分を順に参照する。前述したように配列の 1 次元目のサイズを適正なサイズである 8 としたため、ループの次の回転では連続した次の 8 成分を順に参照する。よってこのロード命令は SIMD 命令で実行できるのが望ましいが、アンロール実装にしたため配列インデックスはソースコード中で即値で指定されており、コンパイラが連続領域へのアクセスであると見切れず SIMD ロード命令が発行されなかった。そこで再度ループで実装しなおし SIMD ロード化を促進した。

その他にも、ループ中の配列数の調整を行った。キャッシュは一定サイズのライン単位で管理され、それぞれのラインには固有のライン番号が付与され管理されており、メモリ上のある領域を参照する際に、そのデータがキャッシュ上のどのラインに置かれるかはアドレスをもとに計算で決定される。メモリ容量よりキャッシュ容量の方が少ないため、当然、異なるメモリ領域のデータが同じキャッシュラインに置かれることもしばしば発生する。キャッシュはライン番号とともにウェイという単位で管理されており、例えば 2 ウェイのキャッシュであれば同じライン番号に 2 つの異なるアドレスからのデータを保持できる。しかしウェイ数は一般に小さな数であるため、ループ中で参照する配列数が増えると同一ラインに載る配列データが増加し、結果としてキャッシュからあふれやすくなる。本ループでは複数の配列を参照しているが、とりわけ三つの 2 次元配列を同一のアクセスパターンで参照しており、キャッシュラインの競合が起きやすい状態であった。そこで、アクセスパターンが同一である特徴を生かして、三つの 2 次元配列を一つの 3 次元配列として実装し直しキャッシュの利用効率を向上した。

図 4.91 に改変前後の疑似ソースイメージを示す。

さらに、セクタキャッシュのパラメータ調整を実施し 61 秒から 48 秒まで向上した。このときのメモリスループットは、40.3GB/s から 42.5GB/s まで向上した。これは京での実効上限にほぼ匹敵する値である。

このチューニング版コードを、富岳上で実行し CPU 性能解析レポートを採取したところ、スレッド間でのインバランスが発生していた。演算量の観点では、スレッド間のインバランスは発生しないが、原因はリストアクセスされる配列 (図 4.91) における配列 fgp) へのアクセスで、ス

```
subroutine FLD3X2(fDP, fDNX, fDNY, fDNZ,
                  fGP, iNODE, NE, N1, NP)

real*4    fDP(NE), fGP(3,NP)
real*4    fDNX(N1,NE), fDNY(N1,NE), fDNZ(N1,NE)
integer*4 iNODE(N1,NE)

DO iE=1, NE
   fDP(IE) = fDNX(1,iE)*fGP(1,iNODE(1,iE))+
             fDNX(2,iE)*fGP(1,iNODE(2,iE))+
             ...
             fDNX(8,iE)*fGP(1,iNODE(8,iE))+
             fDNY(1,iE)*fGP(2,iNODE(1,iE))+
             fDNY(2,iE)*fGP(2,iNODE(2,iE))+
             ...
             fDNY(8,iE)*fGP(2,iNODE(8,iE))+
             fDNZ(1,iE)*fGP(3,iNODE(1,iE))+
             fDNZ(2,iE)*fGP(3,iNODE(2,iE))+
             ...
             fDNZ(8,iE)*fGP(3,iNODE(8,iE))
ENDDO
```

```
subroutine FLD3X2(fDP, fDNXYZ,
                  fGP, iNODE, NE, NP)

real*4    fDP(NE), fGP(3,NP)
real*4    fDNXYZ(8,3,NE)
integer*4 iNODE(8,NE)

DO iE=1, NE
   fV=0.0
   DO J=1,8
      iP=iNODE(J,IE)
      fV += fDNXYZ(J,1,iE)*fGP(1,iP)+
            fDNXYZ(J,2,iE)*fGP(2,iP)+
            fDNXYZ(J,3,iE)*fGP(3,iP)
   ENDDO
   fDP(iE)=fV
ENDDO
```

図 4.91: FLD3X2 の改変

レッドごとにキャッシュミス率が変化していたためであった。リストの値は入力データ依存であるため、この現象の回避策として、配列 fgp のプリフェッチを促進するディレクティブを挿入し、パラメータスタディにより適切なプリフェッチ距離を決定した。これによりスレッド間のインバランスはなくなった。これにより富岳では、同一の処理をおよそ 18 秒程度の時間で実行できると推定されている。(ただしメモリの不足から、半分の問題サイズで測定し、その 2 倍としている。メモリ不足については最新版では解決される)。このとき富岳でのメモリスループットは 125GB/s であった。この値は富岳 1CMG の実効メモリスループット上限である 205GB/s と比してまだ向上する余地があることを示している。

4.9.4 運動方程式ルーチン (VEL3D1) 主要ループの高速化の検討

主要ルーチン中で最後まで改善されずに残っていたルーチンである運動方程式ルーチン (VEL3D1) 主要ループについては、メモリスループットの観点では計算時間として 10 秒弱が目標であるが現状 30 秒程度であるため改善検討を継続している。

本ループは他の主要ループと比べて、ループボディが大きく複雑で、参照される配列および一時変数が多いためレジスタスピルが発生している。富岳ではレジスタが京と比べ少なくなるためスピルの発生は重大な懸念であり、様々な実装パターンについて現在提供されている富岳向けコンパイラを用いてスピルの削減について調査している。

また、本ルーチンのループ構造は改善前の勾配計算ルーチンと同様に、図 4.88 並びに図 4.89 で示した、要素のループを回し節点に対して値をストアする構造をしているため、ストアが多く、またデータの依存性があるため SIMD 化ならびにソフトウェアパイプライン化が出来ないという問題があった。

そこで 4.9.2 に示したループ構造を節点ループに変える改善を検討した。その結果、実行に 32 秒を要していたが 12 秒まで改善した。メモリスループットは低いためさらなる改善の余地がある。また富岳のシミュレータでの実行では SIMD の効果が出て、8 秒弱にまで改善した。このときはコンパイラは、ある演算命令の結果が入るレジスタを、直後の命令で参照するというような命令列を生成していたため、浮動小数点演算待ちが多く計上されていた。メモリスループットの観点から考えても、さらなる性能向上が得られるものと期待している。ただしこの富岳シミュレータで

の実行に際しては、シミュレータでの実行の制限から、問題サイズと全体反復回数の大幅縮小を行っているため結果には留意が必要である。このときの京でのメモリスループットは、11.3GB/s から 34.3GB/s まで向上した。京の実効上限よりはまだ小さい値であるが、本カーネルについては他カーネルと異なって演算の占める割合が大きく、B/F 比が 1 程度とそれほど大きくなかったため、この程度の値にとどまっている。

富岳実機を用いた測定では、同一の処理をおよそ 7 秒弱程度の時間で実効できると推定されている。(ただしメモリの不足から、半分の問題サイズで測定し、その 2 倍としている。メモリ不足については最新版では解決される) このとき富岳でのメモリスループットは 74.6GB/s であった。

4.9.5 疎行列ベクトル積ルーチン (CALAX) 主要ループ

疎行列ベクトル積ルーチンについては、京ではメモリスループットの観点並びに B/F 比の観点から実効上の上限性能に達していたため、京上でのチューニング検討は行ってこなかった。一方で富岳上で実行した際には十分なメモリスループットが出せなかったため、チューニングを行った。

本カーネルは疎行列とベクトルとの積を行うため、外側ループが行で回るループ、内側ループが各行の非ゼロ要素について回るループという二重ループになっている。最内ループの回転数は高々30 である。京向けの実装では 30 回転する最内ループをアンロールした実装になっていおり、配列アクセスする際のインデックスがループ変数だけでは指定されず、ループ変数からのオフセット値がソース上で例えば I+1、I+2、I+3 などの即値で指定されていた。メモリ空間上は連続したアクセスになるのだが、コンパイル時にコンパイラが把握できず連続 SIMD ロード命令ではなく、不適切なロード命令 (Gather ロード命令) を発行しており、メモリスループットが低かった (およそ 94GB/s 程度)。

そこで 30 展開されたループの再度ループ化を行った。これにより最内は 30 回転するループとなり、配列アクセスのインデックスに即値が登場しなくなり、連続アクセスであることをコンパイラが把握的、連続 SIMD ロード命令が発行されるようになった。結果、これまで富岳上で 5.3 秒程度かかっていた処理が、2.8 秒程度で完了できるようになった。このときのメモリスループットは、94GB/s から 182GB/s 程度まで大きく改善した。これは富岳の実効上の上限 205GB/s 程度と比しても十分な値である。

4.9.6 配列のコピーの除去

本アプリはかなり以前から使われているコードである。過去のチューニングに際してデータ構造の大幅な変更が必要となった。その際アプリ全体を通してデータ構造を変えるのではなく、特定のルーチンに入る前に一時配列を使いデータ構造の変換を行い、ルーチンの終了後にアプリ内で汎用の形式にデータを戻すといったアドホックなチューニングを行った。そのような目的の多数の配列コピーが存在している。

とりわけ前述の勾配計算ルーチン前後でのデータ変換のためのコピー処理には時間を要していた。各節点に 3 次元ベクトルである勾配を保存するのに FrontFlow/blue では基本的には”Structure of Array” の形式を用いているが、勾配計算ルーチンでは”Array of Structure” の形式になっている方が性能が出るため、二つの追加ループを用いて”Structure of Array” と”Array of Structure” の相互変換を行っており、勾配計算ルーチンの結果を”Structure of Array” に変換するループで約 10 秒、隣接通信ルーチンを呼び隣接プロセスからの寄与分も節点に足しこんだ結果を再度”Array of Structure” に戻すためのループで約 11 秒を消費していた。

これは隣接通信ルーチンが”Structure of Array” を前提に実装されている為である。具体的には隣接通信ルーチンでは、隣接プロセスと共有している節点についての値 (速度) を隣接プロセスと交換する。その際に、自プロセスが持つ全節点についての速度を保持している配列から、送信する値だけを送信バッファにコピーし、逆に受信した値を受信バッファから全節点についての速度を保持している配列の当該位置へと書き込む処理を行っている。これをパック・アンパック処理と呼称している。この処理の実装は配列が”Structure of Array” であることを前提としているため、”Array of Structure” で実装されている勾配計算ルーチンとの間でデータ構造の変換が必要となっている。今回新たに”Array of Structure” を前提とした隣接通信ルーチンを追加し、勾配計算ルーチンにおける隣接通信に利用することで 21 秒かかっていた変換処理が不要となった。

4.9.7 配列のゼロクリアの除去

実行時間が全体の実行時間に占める割合の観点で主要な配列のゼロクリアは 2 カ所存在した。

一つは運動方程式ルーチン中での配列のゼロクリアであり、これは多次元配列を多重ループで初期化していた。多重ループの最内ループの回転数が著しく小さいためソフトウェアパイプラインは適用されず、小回転ループのオーバーヘッドにより実行効率は低かった。

配列の全成分を 0 にするという動作のためメモリの参照は連続であるため、多次元配列を 1 次元配列と見なし、内部で memset 関数を呼ぶ C で作成した初期化ルーチンを利用するよう変更した。単純かつ連続的な書き込みなので、ストア時にストア先アドレスを一旦キャッシュまでロードする動作のない XFILL が適用可能で 2.3 秒から 0.6 秒まで短縮した。これはメモリスループットの観点では十分な値であった。

もう一つは勾配計算ルーチン主要ループ直前での配列のゼロクリアである。これは元々のループ構造では、要素ごとに計算された値を、その要素が参照している節点へと足し込むというリダクション演算を行っており、節点は複数要素から共有されているため、特定の節点への足し込みが繰り返し何度も発生するため、予め節点の値をゼロクリアしておく処理である。4.9.2 で述べたように、勾配計算ルーチン主要ループのアルゴリズム変更により、特定の節点への書き込みは 1 回で済むようになるため、8.6 秒かかっていたゼロクリアが不要になる。

4.9.8 非スレッド並列ルーチン (CRSCVA, CLRCRS, NODLEX, DGNSCL) のスレッド並列化

行列の CRS 形式から ELL 形式への変換 (CRSCVA)、行列への境界条件の書き込み (CLRCRS)、要素定義値の節点定義値への補間 (NODLEX)、行列の対角スケーリングの処理 (DGNSCL) といった 4 ルーチンについては、アプリ開発当初は自動並列化によるスレッド並列に対応していたが、これまで行われてきたアプリへの機能追加のため行った改変により、自動並列化されなくなっていたことが判明した。

これらのルーチンの実行時間が全体の実行時間に占める割合は 4 ルーチン合計で、テスト問題におけるタイムステップループ実行時間の 15%に匹敵していた。

当初は、これらのルーチンの再スレッド並列化の効果は実測ではなく推定で行ってきた。具体的には 1 スレッドでの動作だったものが 8 スレッドで動作することになるため、7 倍程度には向上するであろうという評価を行ってきた。今回実際にこれら 4 ルーチンのスレッド並列化を実施した。メモリアクセスパターンの改善なども併用することでスレッド並列化の効果以上に改善し、テスト問題では 4 ルーチンで合計 62 秒かかっていたものが、3.8 秒にまで改善した。

4.9.9 if を含むループのリストベクトル化

これは上述の勾配計算ルーチン (GRAD3X) の第二ループのことである。ほとんどの入力データでは、ある節点の上位要素数は最大でも 8 程度であるが、一部複雑な形状をしている場合上位要素数が 8 以上であるような節点が発生する。実行効率を出すため GRAD3X では主要演算ループを、8 個目までの上位要素からの寄与を計算するループと、9 個目以降の上位要素からの寄与を計算するループに分けている。このループは外側が節点で回るループで、各節点に対して上位要素数が 8 以上かを判定する if 文があり、上位要素数が 8 以上である節点に対してのみ、この節点上で求める勾配の値への、9 番目以降の上位要素からの寄与を計算するループを実行する。この最内ループにはコンパイラにより SIMD 化及びソフトウェアパイプライン化が施されている。

テスト問題には上位要素数が 8 以上の節点は含まれていないため、このループは本来「何もしないループ」であるはずだが、実際には全体の実行時間 394 秒の内、その約 2%程度の 8 秒ほどの時間を占めていた。

調査の結果このループの処理時間の多くを分岐命令待ちが占めていると判明した。これはこのループの実際の操作は「何もしない」ではなく、全ての節点に対して上位要素数が 8 以上であるかを判定しているためであり、かつ節点で回るループは最内ループではないため SIMD とソフトウェアパイプラインが適用されず命令の実行効率が悪いためである。

このような、ループ中に if を含むようなコードで、くり返し実行される際に、if が真となる分布が毎回変化しないような場合は、予め、if が真となるインデックスだけを集めたリスト (リストベクトル) を作成し、リストベクトルに含まれる対象にだけ処理を行うリストベクトル化が有効である。

このループはテスト問題では 11880 回くり返し実行され、各節点の上位要素数はプログラム実行を通じて変化しないため、このループにはリストベクトル化が適用可能であった。

その結果、テスト問題では上リストベクトルは空になるため、実行時間は約 8 秒からほぼ 0 秒に改善した。なおプログラム実行中に 1 回だけ実行される if が真になるものだけを集めるリストベクトル作成処理の時間もほぼ 0 秒と見なせる時間である。

4.9.10 型変換ループの除去

FrontFlow/blue は圧力や速度などの浮動小数点の値を基本的には単精度で保持している。圧力などのスカラ量は要素中心に保持するが、速度や圧力勾配といったベクトル量は節点上に保持している。隣接通信において、自プロセスと隣接プロセスとで共有している節点上に定義される値は、両プロセスからの寄与の合計となる。FrontFlow/blue の元々の実装では、隣接通信ルーチン内において、自プロセスからの寄与と隣接プロセスからの寄与を合計する際に、一時的に倍精度に変換した後に合計し、結果を単精度に戻すという処理が入っていた。

これはアプリ開発の途上で様々な部分の実装を試行錯誤している段階において用いられた。計算結果の精度低下が発生した際に、原因の切り分けをしやすくするため、通信の影響による精度低下の懸念を払拭するために追加された処理である。この型変換ループは全体の実行時間の 7.64% を占めていた。

両プロセスからの寄与を合計する際の変数の型の影響を調査した結果、計算結果に影響がないことが判ったため、この型変換ループを除去した。これにより 30.3 秒かかっていたものが、0 秒となった。

4.9.11　その他改善検討

将来的なアプリ改善の事前検討として、行列は共通だが、ベクトルが複数ある場合に、行列・ベクトル積をベクトルの数だけ繰り返すのではなく、複数のベクトルをまとめ行列・行列積として処理することで行列のロードを削減して性能向上を狙う改善を検討している。

一度に処理するベクトルの数を増やすと、使う行列は共通であるため、演算量の増大に比してメモリ転送量の増大は抑えられる。よって B/F 値が小さくなり、疎行列・ベクトル積のようなメモリスループット律速の処理は大きく性能改善する。例えば 8 個のベクトルに対して 8 回疎行列ベクトル積を呼んだ合計時間よりも 8 個のベクトルをまとめて処理する時間の方が短くなる。

これまでの検討では、アプリケーション中の全てのカーネルについて複数のベクトルを扱う改善を検討していたが、例えば非線形な処理をしている部分については、行列の各成分が、反復計算の前ステップでのベクトルの値に依存するため、アルゴリズムを線形部分と非線形部分に分けるなどといった極めて大きな工数がかかることが判明したため、現段階においては一部分のカーネルのみ複数ベクトル化することとなった。

4.9.12　通信およびファイル I/O に関して

FrontFlow/blue の並列手法は計算対象となる形状を幾何的に分割した各ドメインをプロセスに割り当てる領域分割法を用いていおり、プロセス内においては演算ループのスレッド並列化を行っているハイブリッド並列である。プロセス間の通信は大域通信と隣接通信とに分けられる。大域通信は陰解法ソルバーとして用いている CG 法の反復中で発生するベクトルの内積計算のための大域通信 MPI_Allreduce である。隣接通信は、前述の通り隣接相互作用のみを扱うので、隣接するドメイン間で共有される節点での速度を交換するための MPI_Isend, MPI_Irecv,MPI_Waitall である。FrontFlow/blue は汎用ソフトであるため、通信時間、特に隣接通信時間は扱っている形状とその領域分割の様相に依存するが、コデザインでターゲットとしている問題では全実行時間における通信時間の割合は総じて低く、性能改善のための検討は不要とした。

FrontFlow/blue は汎用ソフトであるため、扱う問題に応じてどういったファイルをどのタイミングで I/O するかといった設定はパラメータファイルで設定でき、自由度が高い。ここでは最も標準的な I/O パターンとしてプログラム初期化時における入力、時間発展ループ中における入出力、終了時における出力に分けて考えた。初期化時における入力ではパラメータファイルとメッシュファイル、境界条件ファイルを入力する。メッシュファイルと境界条件ファイルは領域分割されたドメインごとのメッシュファイルと境界条件ファイルに分割されており、各プロセスは自身が担うドメインに関するファイルのみを読む。パラメータファイルは全プロセスで共通である。京ではランクディレクトリとステージングの機構により各入力ファイルは各プロセス毎に専用の場所にコピーされるので、パラメータファイルのようなプロセス間で共通するファイルの入力も動作としては異なるファイルを読む動作となるため特定のファイルへのアクセス集中は避けられるが、ポスト京ではステージングを用いないため、プロセス間で共通するファイルの入力については留意が必要である。時間発展ループ中における入出力では全プロセスがプログラム動作のログファイルのみを出力する。ログファイルはプログラム開始時に open し、プログラム終了時に close する。終了時における出力については、全プロセスが自身が担うドメインについての計算結果のファイルを出力する。

4.9.13 入力ファイル生成について

コデザインにおいて、様々なプロセス数でのウィークスケール測定を繰り返す必要があるため、このウィークスケール測定用入力データを用意するために、FrontFlow/blue では完全にウィークスケールしている入力データを自動的に作成できる、入力データ自動生成プログラムを用いている。

これまでは上記入力データ自動生成プログラムを用いて、性能測定したい問題サイズ・プロセス数での入力データを、あらかじめ生成していた。この入力データ自動生成プログラムは、ログインノード上で動作する、逐次で動作するプログラムであるため、富岳向けの非常に大きなプロセス数の入力データを作成するのには多大な時間がかかる。そして、大きなプロセス向けのデータはファイル数が膨大なため、ストレージ容量を圧迫するが、生成に多大な時間がかかるため、利用が終わったら削除することができず、次回の利用のために保存しておく必要があった。

入力データ生成を高速で行えるようになれば、必要になるたびにデータを生成すればよく、膨大なファイルを常時保存しておく必要がなくなる。そこで入力データ自動生成ツールの並列化を行った。このツールを富岳上で動作するようコンパイルし、富岳で FrontFlow/blue を実行するジョブスクリプト中で、①入力データ作成ツールを実行、②FrontFlow/blue を実行、③入力データを削除。というように、その都度入力データを生成・破棄することが可能となった。たとえば 10 万プロセスでの測定について考えると、①では各プロセスがそれぞれ自分が使う分の入力データだけを作成すればよいので、ほんの数秒で処理できる。

4.9.14 コデザインによるシステムへの実装要請項目

本コデザインを通してシステムへフィードバックした実装要請・改善項目を以下に示す。

1. インダイレクト SIMD ロード
   前述のように FrontFlow/blue はメモリアクセスのパターンが連続アクセスとならないインダイレクトアクセスである。ポスト京でのメモリスループットは SIMD ロードを前提としているので、インダイレクトアクセスで SIMD ロードができない場合メモリスループットが著しく低下してしまう。そのためインダイレクトなロード時にも SIMD ロード出来ることが重要である。

2. SIMD 幅に一致しないループの SIMD 化
   一度に処理するベクトルの数が SIMD 幅に一致している場合はループが一つの SIMD 命令で処理されるため効率が良い。しかしベクトル数が SIMD 幅に一致しない場合は SIMD が適用されず性能が出ない。このような場合でも、コンパイラが SIMD の適応可能性を判断し、SIMD が適応されることが必要である。

3. L1 アクセスまとめあげ
   FrontFlow/blue はメモリアクセスのパターンが連続アクセスとはならないインダイレクトアクセスである。このようなアクセスパターンのアプリケーションではキャッシュメモリを効率的に活用できるようブロック化、並べ替えなどを施すのが普通であり FrontFlow/blue でも施されている。しかしキャッシュによく乗るように改善を施し、例えば時間的に連続する複数個のロード命令についてそれぞれの要求アドレスが一つのキャッシュラインに含まれていたとしても、L1 へのアクセス要求が順次一つずつ出ているのでは L1 エンジンビジーという状態となりスループットが出せない。そこで連続するロード命令が一つのキャッシュ

ライン上に含まれている場合、L1 へのアクセス要求を 1 回ですませることで L1 エンジンビジーを避ける仕組み導入する。

表 4.108: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (FFB)

実装要請課題	対応
配列の間接参照に対する間接 SIMD ロード命令化	ループ中のインダイレクトメモリアクセス（ロード）に対して SIMD 命令を発行させ、メモリアクセスのパターンが不連続な場合でもメモリスループットの向上をはかる
ループ長が倍精度 8 要素あるいは単精度 16 要素の倍数に一致しない場合に効果的な SIMD 化を実現する機能	少回転ループの SIMD 化機能として開発する
人手でアンローリングを実施したループに対しリローリング（元のループに復元する処理）を行った上で最適化を実施する機能	コンパイラで自動的に実施することは非常に困難なため、アプリケーション開発者側でコードチューニングすることとする。
間接 SIMD ロード命令実行時の L1 キャッシュに対するアクセスリクエスト削減	間接 SIMD ロード命令実行時に、2 要素単位に同一 128Byte ブロックへのアクセスであるかをチェックし、同一の場合は 2 つのアクセスリクエストを 1 つに纏め上げてリクエスト数を削減する

4.9.15　富岳向け性能推定

性能電力予測ツールおよび性能シミュレータ等の環境を用いた富岳向け性能推定は第 4.1.5 節に説明される手法により実施した。

評価対象区間のソースプログラムに対する具体的な分析内容および性能向上を目的とするチューニングの内容については、「次世代超高速電子計算機システムに関するソフトウエア調整（1）報告書」第 4 章性能評価 4.7 節 FrontFlow/blue に詳述されている。

ここでは、富岳におけるターゲット問題の実行性能が、京からどの程度向上するのかについての性能測定とその結果を述べる。京でのベースラインの測定に際しては、京向けにチューニングされた本プロジェクト開始時点でのコードを用いた。一方富岳での性能測定に際しては、上記コードをベースとして、SIMD 幅の増大に合わせた改善を施したコードを用いた。

4.9.15.1　アプリ特有の性質

前述の通り、FrontFlow/blue は非構造格子を用いるアプリケーション群を代表するアプリケーションとして位置づけられている、ものづくりのための汎用アプリケーションであり、様々な入力データを扱える必要があるため、入力データの性質について制限がない。そのためプロセス形状は二次元や三次元で指定せず、一次元で指定する。通信は MPI_Isend, MPI_Irecv, MPI_Waitall からなる隣接通信および 1 要素の MPI_Allreduce による大域通信のみであり、かつ実行時間における通信時間の割合は小さい。

4.9.15.2　目標性能と実行条件

1. 問題規模
   ここではターボ機械、船体周りの流れ、実車空力などの大規模な解析を念頭に置き、計算機全体を一つの問題の計算に用いる大規模単一問題型計算を想定し、約 6748 億要素・10 万タイムステップの問題を扱う。また FrontFlow/blue は汎用のアプリケーションであるため、様々な入力データを扱う必要があるが、本評価に際しては要素数とプロセス数のみをパラメータとして入力データを自動生成するプログラムで作成された単純な形状を用いる。

2. 京でのベースライン測定
   FrontFlow/blue は大規模単一処理型計算を主眼としているので、ここでは想定する富岳システム全体を使って解く問題を京で解いた際の時間について推定した。本節では京での実行時間の推定方法について、まず問題の設定を述べた後、実行時間を初期化時間、演算時間、通信時間、バリア時間の 4 つに分け、それぞれの時間の推定方法について述べ、最後に推定結果を示す。

   4.9.15.2.0.1　問題の設定　富岳で想定している問題規模はシステム全体で約 6748 億要素を扱う 10 万タイムステップの計算である。この問題規模は、富岳上の各プロセスでアプリケーションが使えるメモリ量の上限により決定しており、富岳 1 プロセス当たりおよそ 106 万要素となっている。

   FrontFlow/blue は汎用のアプリケーションであり、問題規模が同じであっても、要素タイプ (ヘキサ要素なのか、テトラ要素なのか等)、また領域分割の様相などにより演算量や通信の負荷が変化する。本検討ではプロセス当たりの要素数と、全体のプロセス数をパラメータとする入力データ自動生成プログラムにより生成された、各プロセスが担う形状がヘキサ要素のみからなる立方体形状で、各プロセスの演算量は等しく、各プロセスは末端の 2 プロセスを除いて、ランク番号が隣接するプロセスとのみ隣接しているという形状を想定している。

   京で約 6748 億要素の問題を計算するには、1 プロセス当たりおよそ 814 万要素を扱う必要があるが、このような問題サイズはメモリの不足から京では実行不可能であるため、本検討では京で実行可能な問題サイズである 1 プロセス当たり 106 万要素の計算を 24 プロセスで 100 タイムステップ実行し、そこから 1 プロセス当たり 814 万要素での計算時間を推定するという手法を用いた。以下この 1 プロセス当たり 106 万要素、全体で 24 プロセス、100 タイムステップでの実行をテスト実行と呼ぶ。ここで 24 プロセスにしたのは、後述する富岳での性能予測のための測定と同じ入力データを用いるためである。

   4.9.15.2.0.2　初期化時間の推定　初期化時間とは入力ファイルの読み込みや、演算部分での処理を効率化するためのデータの並べ替えのような処理を行っている初期化部分の所要時間である。

   京でのテスト実行での測定では、初期化部分の所要時間はおよそ 66 秒程度であり、一方でメインループの処理時間は 100 タイムステップで 394 秒であった。初期化部分はアプリケーション実行時に 1 回だけ実行すれば良い部分である一方で、メインループの処理時間はタイムステップ数に比例するため、10 万タイムステップを想定すると初期化部分の時間は無視できるため、検討には含めないこととした。

4.9.15.2.0.3　演算時間の推定　演算時間とはメインループの実行時間のうち、隣接通信時間、大域通信時間、バリア時間を除いた純粋に演算だけをしている時間である。

ここではまず、テスト実行を行ってメインループに要した時間を計測し、ソースコード中に詳細に挿入したタイマーにより、所要時間がメインループの全時間の 1%を超える演算区間 14 個を選定した。さらに 1%を超えない演算区間の合計値、すなわちメインの時間から、14 演算区間の合計時間、隣接通信時間、大域通信時間、バリア時間を引いた余りの時間についても一つの区間と見なし合計 15 個の区間を選定した。扱っている問題では演算量は問題サイズに比例するため、テスト実行の要素数である 106 万と、想定している要素数である 814 万の比である 8 をテスト実行の 15 区間の演算時間の合計値に乗じて、1 プロセス当たり 814 万要素における 100 タイムステップの演算時間とした。

4.9.15.2.0.4　通信時間の推定　通信時間とは FrontFlow/blue で使われる隣接通信の時間と大域通信の時間との合計であり。隣接通信は節点が持つ xyz 各方向の流速成分 uvw をその節点を共有する隣接プロセスと交換する通信であり。大域通信は CG 法におけるベクトル内積値の 1 スカラの MPI_Allreduce である。

- 隣接通信の時間
  はじめに隣接通信である。問題のサイズによって隣接通信量は変化するが、想定する問題は京では 1 プロセス当たり 814 万要素となり実行出来ないため、ここでは問題サイズから理論的に導き出される通信量と、通信性能から推定した。まず通信量であるが、想定している問題では 1 プロセス当たりの形状が立方体であるため、1 プロセス当たりの要素数から隣接面での節点数を求めることが出来る。1 節点当たり uvw の 3 変数を持ち、各変数が 4 バイトであることから、隣接面一つ当たりの 1 回通信のメッセージ長を求めた。
  ここでさらに、1 タイムステップ当たりの隣接通信回数 156 回、タイムステップ数 100、隣接プロセスへの送信と隣接プロセスからの受信があることから、隣接面一つ当たりの総通信量を求めた。
  京ではスペック上は 5GB/s で送信でき 5GB/s で受信できるので隣接面一つ当たり送受信合計で 10GB/s の通信スループットを持つが、想定している問題では 1 回通信のメッセージ長がおよそ 450KB となり、想定している問題と同様の通信を行っている Intel MPI Benchmarks (IMB) の Exchange の京での結果では、このメッセージ長のスループットがおよそ 8GB であったことから、隣接面での送信受信の合計スループットは 8GB/s とし、総通信量を合計スループットで割ることで隣接通信時間を求めた。
  このように推定した隣接通信時間に、京で実際に実行できる問題サイズについて、上記のように通信量とスループットから推定した隣接通信時間と、実測した隣接通信時間との比 5.69 及び、実際に大並列の問題を計算した際の通信パターンが複雑になる可能性を考慮した安全係数 3 を乗じた値を隣接通信時間とした。
  ここで比 5.69 は、スループットで見積もった値は理想値であるのに対して、実際には輻輳や OS ジッタなど様々な影響で通信時間が劣化することを考慮した係数である。
  安全係数 3 は今のテスト問題では 1 プロセスが担う形状が 6 面体であるうち隣接面が 2 個だけであるが、隣接関係がより複雑になり隣接面が増えても隣接面数は最大で 6 面となるため、安全係数として 3 を乗じている。ちなみに隣接面数が 6 面以上に増えると、隣接面 1 面当たりの通信量は減り、合計の通信量は一定である。

- 大域通信の時間
  ついで大域通信時間である。前述のように FrontFlow/blue で用いられる大域通信は 1 スカラの MPI_Allreduce であり時間はノード数のみに依存する。
  ここでは京で実行出来る 8 万並列の問題で実測した 1 回当たりの MPI_Allreduce 時間 4.65×10^{-5} 秒と、1 タイムステップ当たりの大域通信回数 400 回、タイムステップ数 100 から大域通信時間を推定した。

4.9.15.2.0.5　バリア時間の推定　バリア時間については、FrontFlow/blue では隣接通信及び大域通信のラッパールーチン中で通信前と通信後のバリアを明に実行している。想定している問題ではプロセス間の処理量のインバランスが無いため原理的にはバリア時間は 0 であるはずだが実際にはバリア時間が生じている。この原因は各プロセス毎の OS ジッタや、動的に確保される配列のアドレスが各プロセスで微妙に異なることによるキャッシュミス発生の違いなどが原因と考えられモデル化が困難である。そこでここではバリア時間のモデルとして、バリア時間はその直前に行われている処理の時間に依存するという仮定を置いた。

- 通信前バリア時間
  通信前のバリアであるが、通信前のバリアの前に行われている処理は演算であり、演算時間は要素数に依存する。よって京でのテスト実行での通信前バリア時間に、テスト実行の要素数である 106 万と、想定している要素数である 814 万の比である約 8 を乗じた値を想定問題での通信前バリア時間とした。
- 通信後バリア時間
  通信後のバリア時間であるが、通信後バリアには隣接通信後バリアと、大域通信後バリアが存在する。
  はじめに隣接通信後バリア時間であるが、京でのテスト実行での実測した隣接通信後バリア時間に、テスト実行での実測した隣接通信時間と、想定問題での推定隣接通信時間との比を乗じた値を隣接通信後のバリア時間とした。
  次いで大域通信後バリア時間であるが、京でのテスト実行での実測した大域通信後バリア時間に、テスト実行での実測した大域通信時間と、想定問題での推定大域通信時間との比を乗じた値を大域通信後のバリア時間とした。

4.9.15.2.0.6　まとめ　以上の推定方法で富岳での想定問題である全体で約 6748 億要素の問題を京で実行した際の実行時間とその内訳を示す。なおここでの値は 100 タイムステップの時間を 10 万タイムステップ分へと線形外挿したものである。テスト問題では反復計算の収束判定条件を厳しく設定しているため、反復計算は常に予め指定している上限回数まで回ることになる。そのため、どのタイムステップにおいても演算量は一定であり、10 万タイムステップ分の計算時間は、100 タイムステップの 1000 倍となる。

表 4.109: FrontFlow/blue 京ベースライン実行時間 10 万タイムステップ

問題サイズ	時間 (h)		
	全体	演算	通信
約 6748 億	838	824	14.2

3. 富岳での性能測定と目標性能
   想定している問題を実際に富岳で実行した際の実行時間を測定した。ここで用いたソースコードは、前節 4.9.2 以降で述べたチューニングを統合したものである。

   本節では富岳での実行時間の測定方法について、京でのベースライン測定と同様に、まず問題の設定について述べ、実行時間を初期化時間、演算時間、通信時間、バリア時間の 4 つに分けそれぞれの時間の測定方法について述べ、最後に結果を示す。

4.9.15.2.0.7　問題の設定　先述したように、想定している問題は全体で約 6748 億要素を扱う 10 万タイムステップの計算であり、富岳では 1 プロセス当たりおよそ 106 万要素、各プロセスは末端の 2 プロセスを除いて、ランク番号が隣接するプロセスとのみ隣接しているという形状を想定してる。

4.9.15.2.0.8　初期化時間の測定　京ベースラインで述べたように、初期化時間は含めない。

4.9.15.2.0.9　演算時間の測定　富岳上で 1 ノード 4 プロセスで実行し実測した。ただし、現在のソースではメモリを過剰に用いており、想定している 1 プロセス当たり 106 万要素の問題は実行できないため、半分の 1 プロセス当たり 52 万要素の入力データで時間を測定し、得られた時間を 2 倍としている。なお、本稿執筆時点では、開発者よりメモリ削減版が提供されている

4.9.15.2.0.10　隣接通信の時間およびバリアの時間の推定　富士通が所蔵する 1 ノード当たり 64GB のメモリを持つ FX10 は、京と同じネットワーク性能をもち、かつメモリが潤沢にあるため、京や FX100 では再現できない 1 ノード当たり複数プロセスの測定が行えるため、この環境を用いて通信時間およびバリア時間を測定し、富岳の通信性能の京に対する向上比で割ることで推定した。

4.9.15.2.0.11　大域通信の時間　大域通信については富士通より提供された 1 要素 MPI_Allreduce の性能推定式と、1 タイムステップ当たりの大域通信の回数 400、およびタイムステップ数 100 から推定した。

4.9.15.2.0.12　まとめ　通常モード並びにブーストモード時の、富岳全ノード使用時の推定時間をエコモード無効時 (表 4.110) と、エコモード有効時 (表 4.111) について示す。また、ここから各構成での京に対する性能向上比は表 4.112 に示すものとなる。

表 4.110: FrontFlow/blue 富岳実行時間と電力性能 10 万タイムステップ (エコモード無効)

問題サイズ	タイプ	時間 (h)		
		全体	演算	通信
約 6748 億	通常モード	22.6	19.6	2.7
	ブーストモード	21.0	18.4	2.7

表 4.111: FrontFlow/blue 富岳実行時間と電力性能 10 万タイムステップ (エコモード有効)

問題サイズ	タイプ	時間 (h)		
		全体	演算	通信
約 6748 億	通常モード	23.8	21.1	2.7
	ブーストモード	22.2	19.6	2.7

表 4.112: FrontFlow/blue 富岳の対京性能向上比

エコモード	通常モード	ブーストモード
無し	37	39
有効	34	36

4.9.16　富岳での性能測定

4.9.16.1　富岳での勾配計算ルーチン (GRAD3X) のチューニング

リストアクセスの参照について ocl プリフェッチ指示行によるキャッシュアクセスレイテンシの隠蔽を実施した。チューニングの内容を図 4.92 に示す。またチューニング前後の CPU 性能解析レポートを図 4.93、図 4.94 に示す。該当区間の実行時間は、6.52 秒から 5.38 秒に減少した。また経過時間は、メモリビジー時間に迫っており、ほぼメモリバンド幅上限までの性能が得られている。

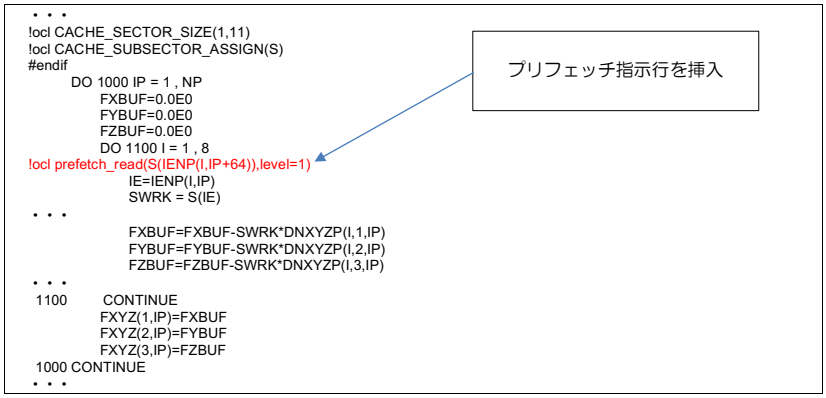


図 4.92: GRAD3X のチューニング (1)

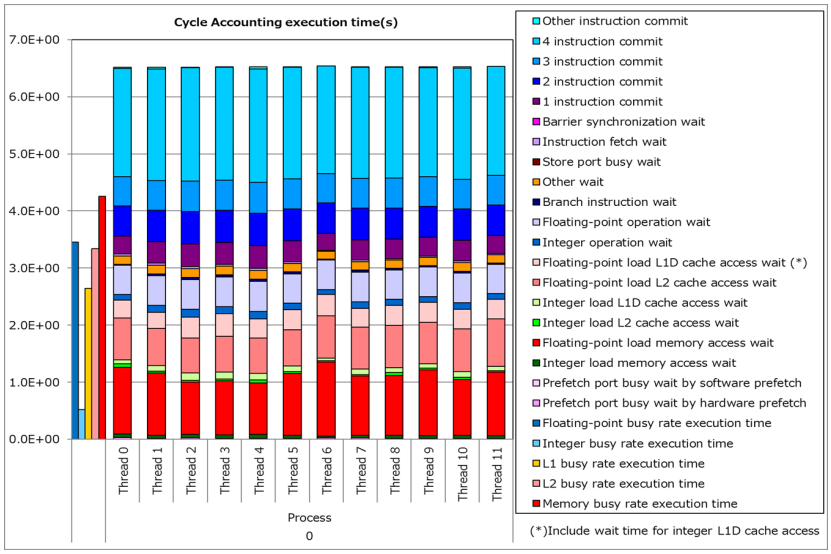


図 4.93: GRAD3X のチューニング前性能

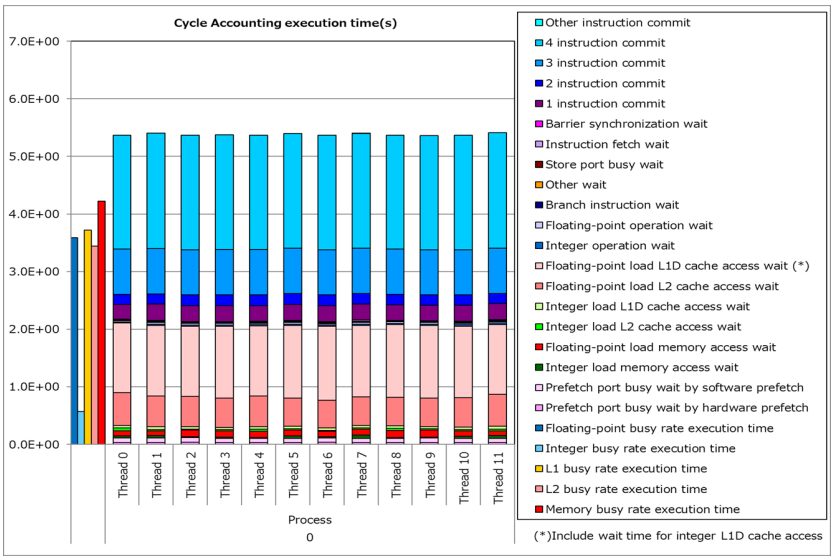


図 4.94: GRAD3X のチューニング後性能

GRAD3X の他の部分にさらなるチューニングを実施した。チューニングの前後のソースの概要を図 4.95、図 4.96 に示す。チューニング前のコードでは、multiple structures load、store 命令ではなく、gather load、scatter store 命令が発行されていた。調査した結果、定義と同一配列を参照していることが原因だった。そのため、FXYZ_tmp という別配列を参照させ、multiple structures load、store 命令の発行が出来る形に変更することで性能を改善する。

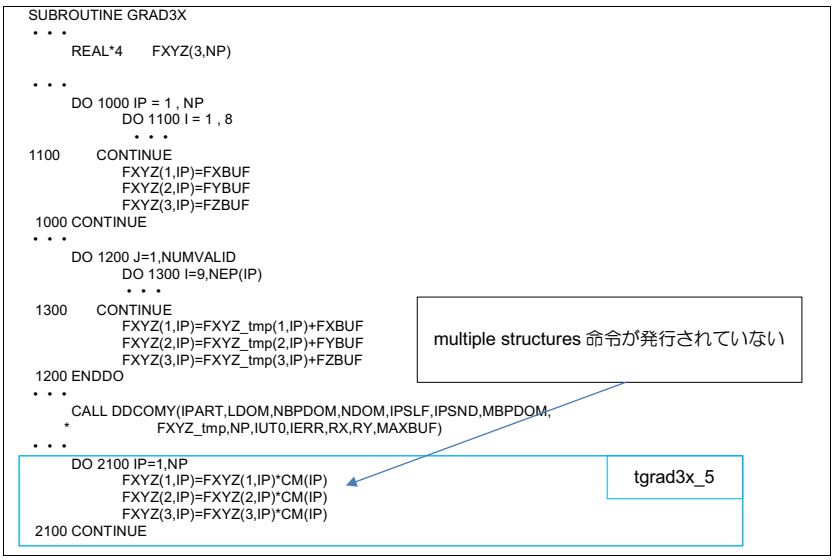


図 4.95: GRAD3X のチューニング (2) 前

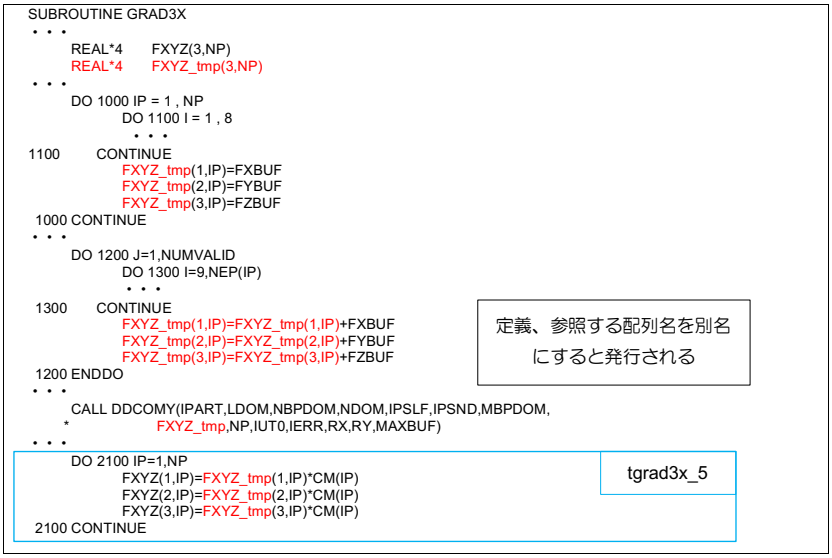


図 4.96: GRAD3X のチューニング (2) 後

またチューニング前後の CPU 性能解析レポートを図 4.97、図 4.98 に示す。multiple structure load、store 命令の発行により、実行時間は 1.75 秒から 1.24 秒に減少した。また経過時間は、メモリビジー時間に迫っており、ほぼメモリバンド幅上限までの性能が得られている。

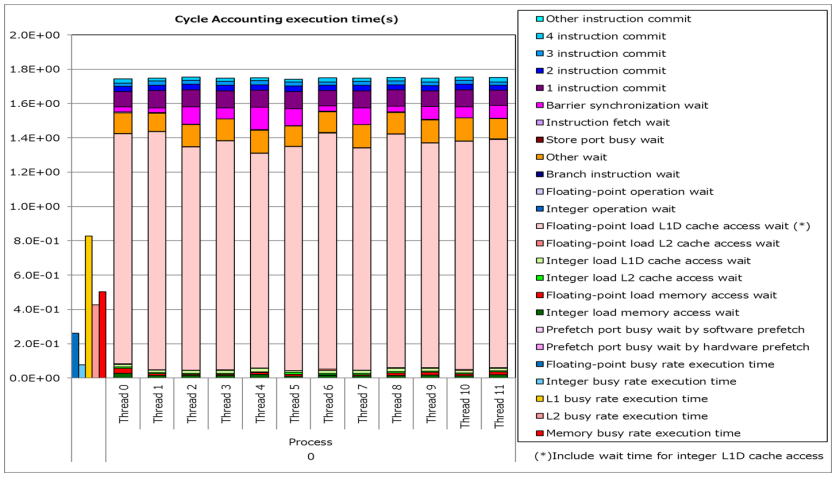


図 4.97: GRAD3X のチューニング (2) 前の性能

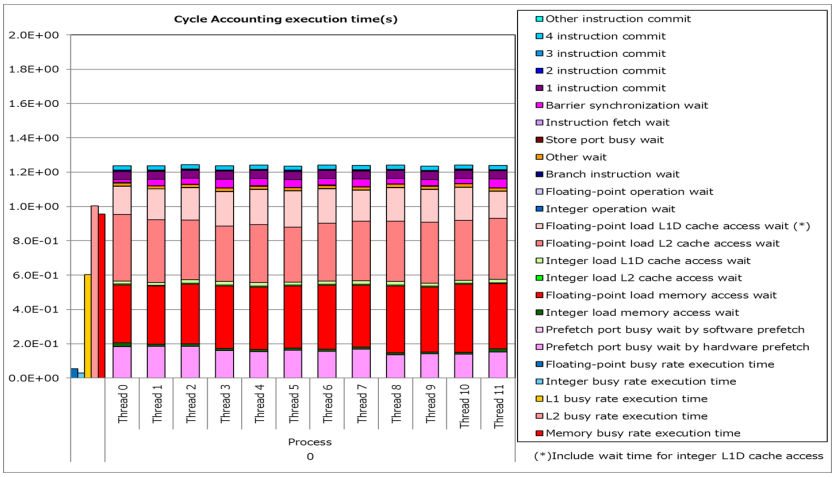


図 4.98: GRAD3X のチューニング (2) 後の性能

4.9.16.2　富岳での発散計算ルーチン (FLD3X2) のチューニング

ocl 指示行でプリフェッチを最内ループ内に適用させ、scaler プリフェッチ命令から gather プリフェッチ命令へ変更し性能向上を狙った。また、リストアクセス用配列:NODE を非 SIMD アクセスから SIMD アクセスへ変更する効果もある。チューニングの内容を図 4.99 に示す。またチューニング前後の CPU 性能解析レポートを図 4.100、図 4.101 に示す。プリフェッチ命令の gather プリフェッチ命令へ変更により、またリストアクセス用配列:NODE の SIMD 化により、該当区間の実行時間は、9.15 秒から 6.67 秒に減少した。Non-SIMD load 命令は、6.06E+10 から 1.02E+10 に減少し、Single vector contiguous load 命令は、2.55E10 から 3.18E10 に増加した。総命令数、4 命令コミット時間は減少し、L1 ビジー率は、74.5%に増加した。

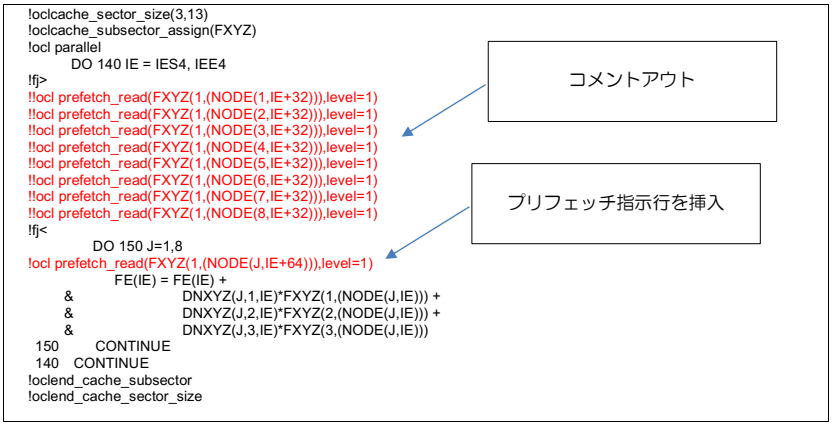


図 4.99: FIELD3X のチューニング

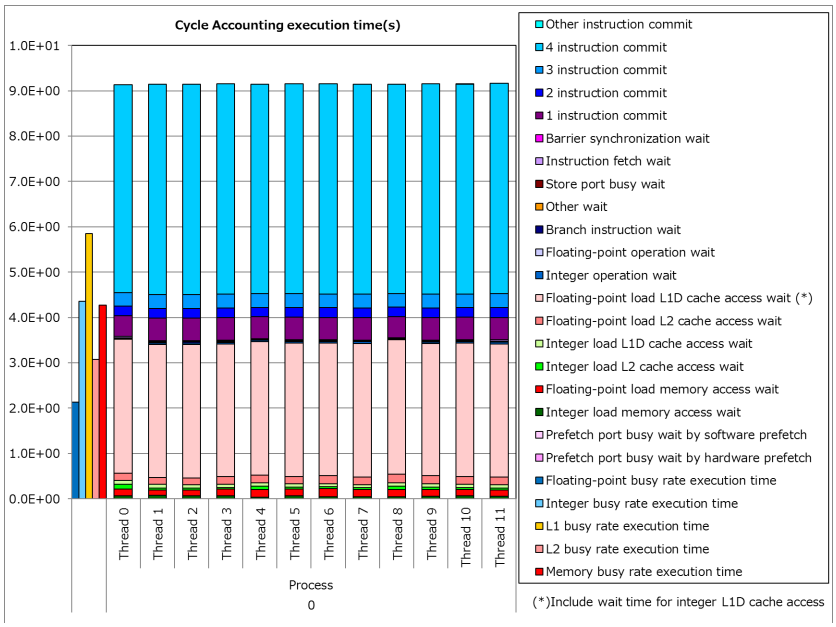


図 4.100: FIELD3X のチューニング前性能

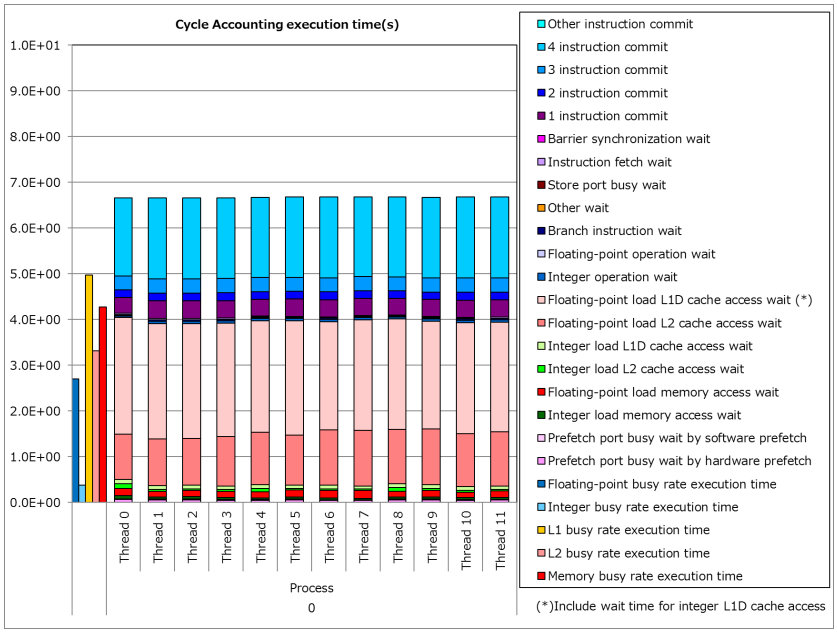


図 4.101: FIELD3X のチューニング後性能

4.9.16.3　富岳での性能測定結果

最終的な主要カーネルの単体性能の結果を、図 4.102、図 4.103 に示す。また図 4.104 に並列性能について示す。

メモリバンド幅ネックである 3 カーネルについては、実効メモリバンド幅のほぼ限界まで達しており、また演算リッチな 1 カーネルについては、最も高いピーク性能比が得られていることが分かる。さらに良好な並列性能が得られていることが分かる。

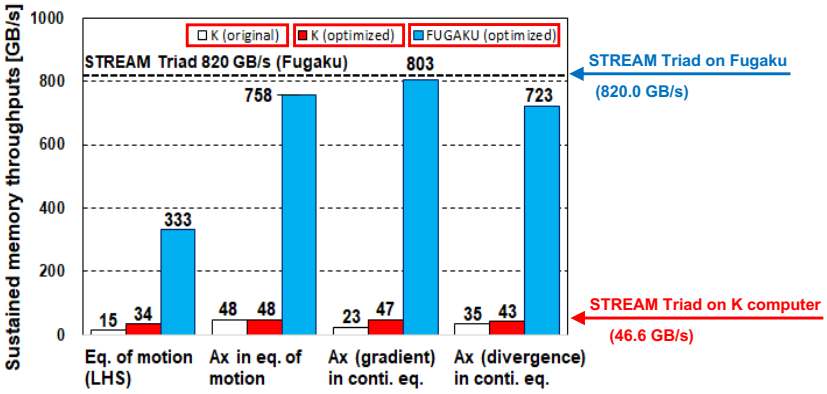


図 4.102: 富岳の FFB 主要カーネルのメモリバンド幅値 (ノード)

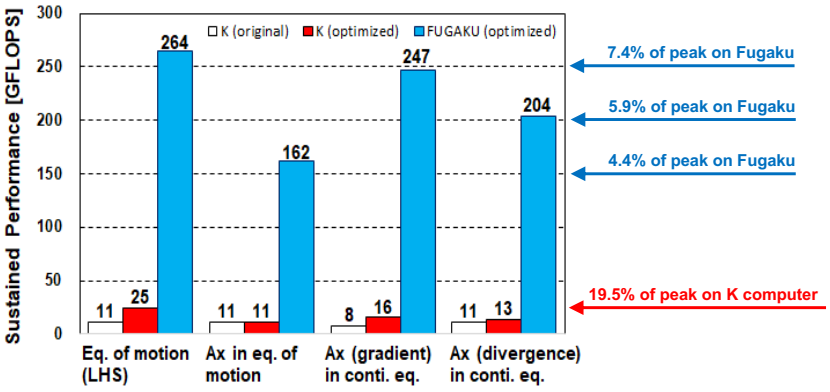


図 4.103: 富岳の FFB 主要カーネルの性能値 (ノード)

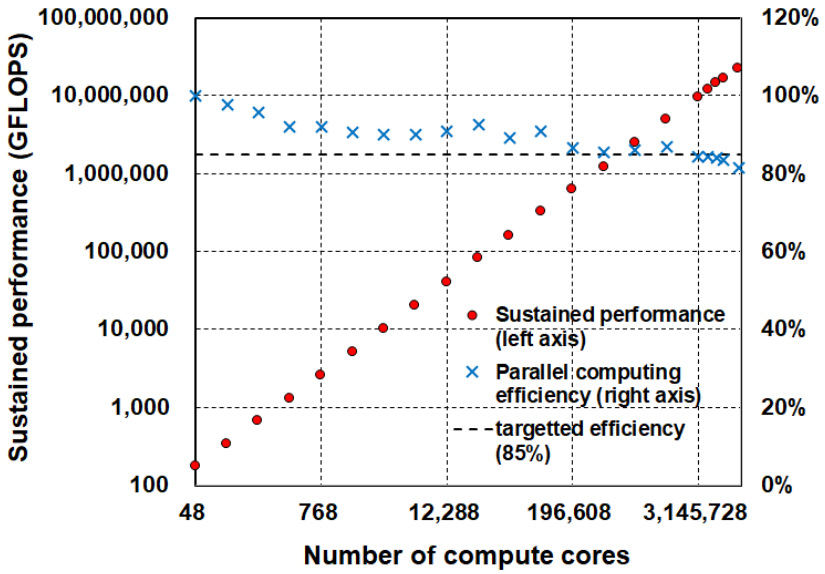


図 4.104: 富岳の FFB の並列性能

最終的な「京」に対する性能向上、および消費電力について、図 4.105 に示す。測定の概要は以下の通りである。

- 測定日時　2020/10/7-8
- コンパイラ・MPI のバージョン：lang/tcsds-1.2.27b
- ライブラリ METIS mpifccpx -Kvisimpact
- ライブラリ REVOCAP_Refiner mpiFCCpx -Kvisimpact -Qt -DNDEBUG
- FFB ライブラリと本体 mpifrtpx -Kvisimpact,ocl,optmsg=2,openmp

最終的にブースト・エコなしで 51 倍の性能向上が得られている。性能倍率の大幅な向上の背景は以下の通りである。

- 勾配を計算するカーネルで，有限要素法では頻出する，scatter store (要素を巡り，要素の周囲節点へ足しこむ) を，節点ベースループにする実装で sequential store 化．これによりキャッシュミスの低減と，ストア命令自体の削減.
- 他カーネルでも，要素をめぐるループで実装されている箇所をすべて節点をめぐるループに変更し sequential store 化.
- OCL 指示子を用い，リストアクセスされる配列へのプリフェッチを促進．さらにプリフェッチ自体も SIMD のプリフェッチ命令を用いることで，メモリアクセスのレイテンシの隠蔽と，プリフェッチ命令による命令数の増大の最小化.
- 不要な処理の除去

システム	京	富岳			
モード	ベースライン	ノーマル エコ無効	ブースト エコ無効	ノーマル エコ有効	ブースト エコ有効
ジョブID		1639449	1639316	1639451	1639351
ノード数	82944	158976	158976	158976	158976
プロセス数	82944	635904	635904	635904	635904
要素数/プロセス	8135928	1061208	1061208	1061208	1061208
10万タイムステップ実行時間［秒］	3016325.0	62282.1	59632.8	69196.7	64681.1
対京倍率		48	51	44	47
MAX MEMORY SIZE (USE) [GiB]		27	27	27	27
消費電力［MW］最大と(平均)		21 (17)	23 (19)	16 (12)	21 (13)

※ 京実行時間は推定
富岳は実行時間は，20タイムステップで実行し10万タイムステップ分に換算(1ステップの実行時間は固定)

図 4.105: 富岳の対「京」性能倍率

4.9.17　執筆者

本節の主な執筆者、および執筆協力者の氏名と所属を記す。（主な参加時期における所属）

4.9.17.0.1　執筆者（執筆責任者）

- 南 一生（理化学研究所）
- 熊畑 清（理化学研究所）

4.9.17.0.2　執筆協力者（５０音順）

- 加藤千幸（東京大学）
- 山出吉伸 (みずほリサーチ&テクノロジーズ株式会社)

4.10 重点課題 9 LQCD のコデザイン

4.10.1 アプリケーションの特徴とコデザインの方針

LQCD では 4 次元時空を規則格子に分割し、格子点と格子点を繋ぐ辺上に物理自由度をおいて量子色力学（QCD）の基礎方程式を評価する。LQCD で最も時間が費やされる部分はクォーク伝搬関数を解くところで、これはグルーオン背景場に依存するディラック方程式を 4 次元格子上で離散化した大規模連立方程式を解くことに相当する。これを QCD クォークソルバーと呼ぶ。離散化手法は幾つか存在するが最もよく使われるウィルソン型の離散化では大規模連立方程式の係数行列は構造格子に基づく大規模ステンシル疎行列となる。クォーク伝搬関数を解くための大規模連立方程式は反復法で解かれる。格子点上には未知数としてクォーク場に相当する 24 実数、格子辺上にはグルーオン場に相当する 18 実数が配置される。係数行列は 1 階差分型のステンシルの構造を持ちメモリバンド幅律速となりうる。必要なメモリ容量は比較的少ない。構造格子であるので並列化のための分割も規則的であるが通信バンド幅律速でもある。強スケーリングがしにくい少メモリ構造格子ステンシル計算を行うアプリケーション群を代表するアプリケーションの一つである。

LQCD は、高いメモリバンド幅及びネットワークバンド幅、通信を隠ぺいする機構、低キャッシュレイテンシ、低通信レイテンシ、OS ジッタがないこと、アウトオブオーダー実行により CPU 性能を最大限引き出すことが可能となる十分なレジスタとアウトオブオーダー資源等を必要とするアプリケーションとしてコデザインに寄与する。

以下に、QCD クォークソルバーの最適化とその計算科学的観点からのコデザインについて述べる。

4.10.2 QCD クォークソルバーの最適化

LQCD では、QCD クォークソルバーの一部である、単精度の領域分割前処理付き (領域内 Jacobi 法)BiCGStab 法（以降、QCDJDD 解法と呼ぶ）の最適化と性能評価を行う。表 4.113 にあるように、QCDJDD 解法をいくつかの区間に分けてそれぞれ最適化・性能評価を行う。演算部分は 5 区間、通信部分は疎行列ベクトル積ルーチン (4 次元ステンシル計算) 内の X、Y、Z、T 方向毎の隣接通信に関する区間と反復法に伴うリダクション通信が 3 区間ある。

表 4.113: LQCD 測定区間一覧

区間	区間名	説明
演算区間	jinv_ddd_in_s_	小領域内での固定反復
	ddd_in_s_	小領域内での疎行列ベクトル積
	ddd_out_pre_s_	小領域間での疎行列ベクトル積の送信開始前の前処理計算
	ddd_out_pos_s_	小領域間での疎行列ベクトル積の受信終了後の後処理計算
	other_calc	反復法のその他の計算
通信区間	comlib_irecv_all_c	小領域間での疎行列ベクトル積のための受信開始
	comlib_isend_all_c	小領域間での疎行列ベクトル積のための送信終了
	comlib_recv_wait_all_c	小領域間での疎行列ベクトル積のための受信待ち
	comlib_send_wait_all_c	小領域間での疎行列ベクトル積のための送信待ち
	s_drbicgstab_dd_hpc_iter_reduc1_	反復法のリダクション通信その 1 (実数、残差計算部、1ヶ所)
	s_drbicgstab_dd_hpc_iter_reduc2_	反復法のリダクション通信その 2 (複素数、内積計算部、1ヶ所)
	s_drbicgstab_dd_hpc_iter_reduc3_	反復法のリダクション通信その 3 (実ノルムと複素内積、2ヶ所)

性能電力予測ツールを用いた性能推定では、表 4.113 の演算区間ごとに経過時間が推定される。ポスト京試作機を用いた測定では、表 4.114 のように、反復法のすべての演算区間と反復法で通信とオーバーラップする区間の二つに分けて経過時間が測定される。

表 4.114: LQCD 測定区間一覧（ポスト京試作機用）

区間	区間名	説明
演算区間	all_calc	反復法のすべての演算区間
	overlapped	反復法で通信とオーバーラップする区間
通信区間	comlib_irecv_all_c	小領域間での疎行列ベクトル積のための受信開始
	comlib_isend_all_c	小領域間での疎行列ベクトル積のための送信終了
	comlib_recv_wait_all_c	小領域間での疎行列ベクトル積のための受信待ち
	comlib_send_wait_all_c	小領域間での疎行列ベクトル積のための送信待ち
	s_drbicgstab_dd_hpc_iter_reduc1_	反復法のリダクション通信その 1 (実数、残差計算部、1ヶ所)
	s_drbicgstab_dd_hpc_iter_reduc2_	反復法のリダクション通信その 2 (複素数、内積計算部、1ヶ所)
	s_drbicgstab_dd_hpc_iter_reduc3_	反復法のリダクション通信その 3 (実ノルムと複素内積、2ヶ所)

4.10.2.1 メモリバンド幅律速の回避

近年では、大規模連立方程式の反復解法に混合精度（倍精度と単精度を混ぜて用いる）を用いる方法が有効であることが分かっている。この混合精度反復法アルゴリズムでは主な計算を単精度で行い、反復改良により倍精度の解を得ることが出来る。本アプリケーションのように、メモリバンド幅律速となる計算では、このアルゴリズムを用いてほとんどの計算を単精度で行うことにより、必要なメモリバンド幅やキャッシュ容量への負担を半減することが出来る。本アプリケーションでも混合精度反復アルゴリズムを利用する。具体的には、単精度前処理付の倍精度 BiCGStab 法の前処理部分として QCDJDD 解法を組み込んだアルゴリズムを用いる。

富岳で想定される問題サイズは 192^4 であるため、計算の大部分を占める反復法に必要なデータ量は、QCDJDD 解法の場合、約 2 テラバイトである。システム全体の L2 キャッシュ容量は約 5 テラバイトであるため、反復法中のメモリアクセスを回避するために、全系規模の 147456 ノード (589824 プロセス) で実行することにした。

アルゴリズム実装を詳細に追跡し、使用しているメモリの削減とゼロ代入の削除を行い使用するメモリサイズの削減を行った。このことによりキャッシュ容量への負荷の削減とキャッシュアクセスやデータ移動の削減を行った。

4.10.2.2 広い SIMD 幅の利用

LQCD は複素数の計算が主体である。現在のコードは 2000 年代初期から発展してきた短い SIMD 幅の HPC アーキテクチャに合わせて構成されている。京では SIMD 幅が 2 であり、レジスタの上部と下部を入れ替えて演算を行う複素数演算に有用な命令があったため、複素数の実部・虚部の順にデータを配置して SIMD 演算を有効に活用していた。

近年の HPC アーキテクチャの発展では SIMD 幅の伸長によるベクトル演算の強化が見られる。この場合、ソフトウエア的には 90 年代ごろのベクトル計算機向けの最適化が再び必要となってくる。本アプリケーションもベクトル計算機向けな単純な連続アクセスが続くデータ構造に変更することで広い SIMD 幅を有効に利用することを検討した。具体的な検討状況は以下のとおりである。

富岳の SIMD 幅は、倍精度が 8、単精度が 16 である。複素数命令が使用可能であるが、複素数のベクトル版と、実部と虚部を分離したベクトル版での検証を行った結果、これまで検討してきた実部と虚部を分離したベクトル版のほうが性能が良いので、実部と虚部を分けた Structure of Array (SoA) の配列を用いて SIMD 化率向上を図る。フェルミオン場に対する京向け配列及び、富岳向け配列を C 言語風に書くと以下のようになる。n_x、n_y、n_z、n_t は、それぞれプロセスあたりの 4 次元の格子サイズである。

京　：$[n_t][n_z][n_y][n_x][3][4][2]$
富岳 (倍精度)　：$[n_t][n_z][n_y][n_x/8][3][4][2][8]$
富岳 (単精度)　：$[n_t][n_z][n_y][n_x/16][3][4][2][16]$
富岳 (半精度)　：$[n_t][n_z][n_y][n_x/32][3][4][2][32]$

富岳では 4 次元格子サイズのうち n_x を分割し、倍精度では長さ 8、単精度では長さ 16 をくくりだし、連続アクセス出来るように転置し SIMD 最適化している。

4.10.2.3　ループ分割およびスレッド並列化軸の変更によるスレッドインバランスの解消

589824 プロセスで 192^4 格子を分割すると MPI の 1 プロセスの担当する領域は $32 \times 3 \times 6 \times 4$ や $32 \times 6 \times 6 \times 2$ 等となる。詳細設計（1）では、プロセス当たりの格子の形状は隣接通信の時間がもっとも短くなる、$32 \times 3 \times 6 \times 4$ を採用していた。

演算区間の `ddd_in_s_`や `ddd_out_pos_s_`は x, y, z, t の 4 次元座標の 4 重ループであり、X 方向のサイズが SIMD 幅 16 とプロセス内領域分割数 2 で割られるため、最内ループ x の回転数が 1 となる。残りの $3 \times 6 \times 4$ の 3 重ループを 12 スレッドで分割する場合、プロセス外参照を行わないようにするための処理によりスレッドインバランスが発生する。

詳細設計（2）では、これを回避するために、プロセス当たりの格子の形状を $32 \times 6 \times 6 \times 2$ に変更することを検討した。この場合以下のように y, z, t の 3 重ループに対して、スレッド並列化対象のループと 4 次元差分計算の方向を同時に切り替えることでスレッドインバランスの解消が出来る。この場合、ロードインバランスの解消による性能向上に加えループボディの単純化によるコンパイラ最適化による性能向上も期待できる。一方で方向毎の差分計算の集積が 2 度に分割されるため、加算演算の増加とキャッシュ利用効率の低下による性能劣化が起こる。性能向上と性能劣化の長所短所のどちらが卓越するかを検証した結果、性能が向上することを確認した。また、第 4.10.2.17 で述べる「プロセス配置と TNI 割り当ての最適化」プログラムでよりよいプロセス配置が見つかり、プロセス当たりの格子の形状 $32 \times 3 \times 6 \times 4$ よりも実行時間が短くなったため、プロセス当たりの格子の形状 $32 \times 6 \times 6 \times 2$ を採用することにした。

以下に LQCD のスレッドインバランス解消のため擬似コードを示す。

for (z, t ループ (6x2=12) を OpenMP 並列化) **do**
　for (x, y ループ) **do**
　　X 方向の差分計算
　　Y 方向の差分計算
　end for
end for
for (y, t ループ (6x2=12) を OpenMP 並列化) **do**
　for (x, z ループ) **do**
　　Z 方向の差分計算
　　T 方向の差分計算
　end for
end for

詳細設計（3）において、`jinv_ddd_in_s_` 及び `ddd_in_s_`に対して、この負荷バランス調整を廃止し、単純な 4 重ループに戻し、関数のインライン展開、共通部分式の除去の最適化の抑止な

どを行ったところ、レジスタのスピル/フィルを行う命令が削減され、ポスト京 CPU 性能シミュレータにおいてより良い性能が得られた。そのため、本最適化は最新版のコードでは行われていない。ddd_out_pre_s_と ddd_out_pos_s_に関しては、まだ負荷バランス調整版のコードであるため、全体として性能がよいプロセス当たり格子形状が $32 \times 6 \times 6 \times 2$ で性能推定がおこなれた。

詳細設計（4）において、ddd_out_pre_s_と ddd_out_pos_s_に対しても、この負荷バランス調整を廃止し、単純な 4 重ループに戻し、関数のインライン展開、共通部分式の除去の最適化の抑止など最適化が施された。スレッドインバランスがあり演算区間における若干の性能低下が見込まれたが、通信時間をより短くできるプロセス当たり格子形状 $32 \times 6 \times 4 \times 3$ を採用してポスト京試作機で性能測定を行った。性能電力予測ツールを用いた性能推定では、ARM 向けの最適化により FX100 で性能が著しく低下したため、詳細設計（3）時のコードを用い、プロセス当たり格子形状 $32 \times 6 \times 6 \times 2$ で採取された PA データをもとに性能推定を行った。

4.10.2.4　L1I ミス数削減のための不均等分割

ddd_out_pre_s_は各スレッドのループ回転数が少ないため、L1I ミス数の割合が多い。もともとは、X、Y、Z、T 方向計算ループをそれぞれ 12 分割して全スレッドに担当を割り当てていたが、各スレッドが X、Y、Z、T 方向計算ループのどれか一つを担当するように変更した。各ループの割り当てスレッド数は x, t ループはそれぞれ 4、y, z ループはそれぞれ 2 である。

この変更により、各スレッドの担当する演算に偏りがでたが、各スレッドが担当する 1 ループあたりの回転数が増加し L1I ミスの割合が減少し、性能が向上した。

詳細設計（4）において、ddd_out_pre_s_を単純な 4 重ループに戻し、ddd_in_s_と同等の最適化を行ったため、本最適化は最新版のコードでは行われていない。

4.10.2.5　手動プリフェッチ挿入による浮動小数点ロード L2 アクセス待ち時間減少

ddd_in_s_等では最内ループ 1 回転先で使用されるデータに対してプリフェッチを行う。以下のように定義される共用体 scs_t の*in に対しては、VLENS = 8 の場合、256Byte 単位でプリフェッチを行う。また、ループ終端では次のループで使用する配列に対してプリフェッチを行う。

```
typedef union {
  float   c[3][4][2][VLENS];
  float   c_prefetch[24*VLENS];
  rvecs_t cv[3][4][2];
  rvecs_t cs[12][2];
  rvecs_t ccs[24];
} scs_t;
```

```
__builtin_prefetch(&((( in+i0+_PFI)->c_prefetch)[0]),0,1);
__builtin_prefetch(&((( in+i0+_PFI)->c_prefetch)[64]) ,0,1);
__builtin_prefetch(&((( in+i0+_PFI)->c_prefetch)[128]),0,1);
```

FX100 での翻訳時オプションと実行時環境変数設定は以下のようになっている。

実行時環境変数	XOS_MMM_L_ARENA_FREE=2 OMP_NUM_THREADS=12
最適化オプション	-Kfast,restp=all,ocl,preex,openmp,noswp,noprefetch,unroll=100

builtin_prefetch 関数によるプリフェッチ命令はウィークプリフェッチがデフォルトであるため、Makefile で以下の処理を行い、ストロングプリフェッチに変換している。

```
ddd_in_s_strp.s:ddd_in_s.s
        sed -e "s/\(prefetch\t.*,\)0$$/\120/g" \
            -e "s/\(prefetch\t.*,\)2$$/\122/g" $< > ddd_in_s_strp.s
```

手動プリフェッチ挿入により、L1 ミス dm 率が改善し浮動小数点ロード L2 アクセス待ち時間が減少し、全ての演算区間で性能向上を確認した。

詳細設計（3）で ARM64 および SVE 向けのチューニングとしてギャザープリフェッチを行ったが効果はなかった。

詳細設計（4）でループ分割をやめて単純ループに戻した。プリフェッチのタイミングが変更となり、ループボディ内で XYZT 方向の差分計算に間に合うように方向ごとにうまく手動プリフェッチを入れるようにした。

4.10.2.6 パラレルリージョンの拡大

omp parallel のコストを削減するため、以下の演算区間内で実行していた omp parallel を呼び出し元の prec_ddd_s_の isap ループの外に移動することでパラレルリージョン生成回数を削減した。

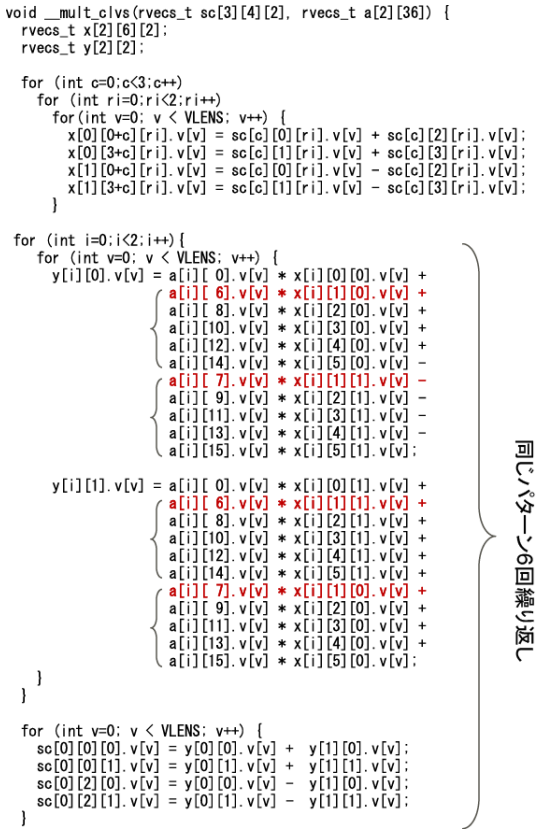
- ddd_in_s_
- ddd_out_pre_s_
- ddd_out_pos_s_
- other_calc
- jinv_ddd_in_s_

これにより、各演算区間で L1I ミス数削減による命令フェッチ待ちの減少、L1D ミス dm 率改善による浮動小数点ロード L2 アクセス待ちの減少、バリア同期待ちの減少、整数ロード命令数、整数ストア命令数減少による整数ロード L1D アクセス待ち、整数ロード L2 アクセス待ちの減少を確認した。

4.10.2.7 __mult_clvs の演算順序並び替えによる最適化

__mult_clvs は ddd_in_s_、jinv_ddd_in_s_、ddd_out_s_の内部でクローバー項の演算に使われている。当初の実装においては、シミュレータでの評価により、この区間でレジスタスピル/フィルによるロード/ストア命令数の増加がみられており、性能劣化の一因となっていた。図 4.106 が最適化前のコードの概略である。赤字で示した演算はその前の演算が完了するまで実行できず前の演算のレイテンシが隠ぺいできない。演算レイテンシを隠ぺいするには、依存関係のない演算を連続して処理するようにすればよい。

富岳では L1D キャッシュへの Byte/Flop 値は単精度で 2 であり、演算パイプラインは 2 本用意されている。1 ロードに対して 1FMA の演算（4 バイトに対して 2 演算）の演算密度があり、かつ、演算レイテンシが 9 サイクルの場合、演算パイプライン 2 本に対して 18 本の演算ストリーム




```
void __mult_clvs(rvecs_t sc[3][4][2], rvecs_t a[2][36]) {
  rvecs_t x[2][6][2];
  rvecs_t y[2][2];

  for (int c=0;c<3;c++)
    for (int ri=0;ri<2;ri++)
      for(int v=0; v < VLENS; v++) {
        x[0][0+c][ri].v[v] = sc[c][0][ri].v[v] + sc[c][2][ri].v[v];
        x[0][3+c][ri].v[v] = sc[c][1][ri].v[v] + sc[c][3][ri].v[v];
        x[1][0+c][ri].v[v] = sc[c][0][ri].v[v] - sc[c][2][ri].v[v];
        x[1][3+c][ri].v[v] = sc[c][1][ri].v[v] - sc[c][3][ri].v[v];
      }

  for (int i=0;i<2;i++){
    for (int v=0; v < VLENS; v++) {
      y[i][0].v[v] = a[i][ 0].v[v] * x[i][0][0].v[v] +
                     a[i][ 6].v[v] * x[i][1][0].v[v] +
                     a[i][ 8].v[v] * x[i][2][0].v[v] +
                     a[i][10].v[v] * x[i][3][0].v[v] +
                     a[i][12].v[v] * x[i][4][0].v[v] +
                     a[i][14].v[v] * x[i][5][0].v[v] -
                     a[i][ 7].v[v] * x[i][1][1].v[v] -
                     a[i][ 9].v[v] * x[i][2][1].v[v] -
                     a[i][11].v[v] * x[i][3][1].v[v] -
                     a[i][13].v[v] * x[i][4][1].v[v] -
                     a[i][15].v[v] * x[i][5][1].v[v];

      y[i][1].v[v] = a[i][ 0].v[v] * x[i][0][1].v[v] +
                     a[i][ 6].v[v] * x[i][1][1].v[v] +
                     a[i][ 8].v[v] * x[i][2][1].v[v] +
                     a[i][10].v[v] * x[i][3][1].v[v] +
                     a[i][12].v[v] * x[i][4][1].v[v] +
                     a[i][14].v[v] * x[i][5][1].v[v] +
                     a[i][ 7].v[v] * x[i][1][0].v[v] +
                     a[i][ 9].v[v] * x[i][2][0].v[v] +
                     a[i][11].v[v] * x[i][3][0].v[v] +
                     a[i][13].v[v] * x[i][4][0].v[v] +
                     a[i][15].v[v] * x[i][5][0].v[v];
    }
  }

  for (int v=0; v < VLENS; v++) {
    sc[0][0][0].v[v] = y[0][0].v[v] +  y[1][0].v[v];
    sc[0][0][1].v[v] = y[0][1].v[v] +  y[1][1].v[v];
    sc[0][2][0].v[v] = y[0][0].v[v] -  y[1][0].v[v];
    sc[0][2][1].v[v] = y[0][1].v[v] -  y[1][1].v[v];
  }
```

図 4.106: `__mult_clvs` の最適化前のコード

があればストリームをラウンドロビンでパイプラインに割り当てることで最大演算性能を引き出すことが出来る。

本区間の演算内容を検討した結果、32 個のレジスタを使用し、読み込みデータの再使用のためのロードを行ったとしても上記条件をほぼ満たすことが出来ることが分かった。具体的には演算ストリームは 16 本あり、4 ロード 4FMA の演算密度であった。演算の順番とストリームの整列後のコードを図 4.107 に示す。現在のコンパイラでは、図のように演算ストリームを整理したとしてもコンパイラの最適化機能によりブロック間の最適化が適用され余計なロード/ストアが発生する。これを防ぐために文脈上意味のない if 文を追加し、さらにコンパイラオプション（`-Knosch_post_ra -Knosch_pre_ra -Knoeval`）を用いて明示的に最適化の抑制をおこなった。これらの最適化によりスピル数が当初の 512 個から 14 個に減少していることを確認した。また、ARM アーキテクチャにおける利用可能なレジスタ数でもこのような最適化が可能であることが確認できた。

4.10.2.8　ARM C Language Extensions（ACLE）による最適化

詳細設計（3）では、SIMD ベクトル化方向である X 方向のステンシル差分計算で現れる、2 つのベクトルを連結して中間を 1 つのベクトルとして取り出す処理を SVE 向け ACLE 関数で実装した。その結果、差分計算に必要な配列要素を連結する処理が SIMD レジスタ上で完結するようになり、一時配列に対するロード/ストア命令を削減することができ、実行効率が大きく上がった。図 4.108 に最適化前の X 方向のステンシル差分計算の概念図、図 4.109 に最適化後の X 方向のステンシル差分計算の概念図を示した。

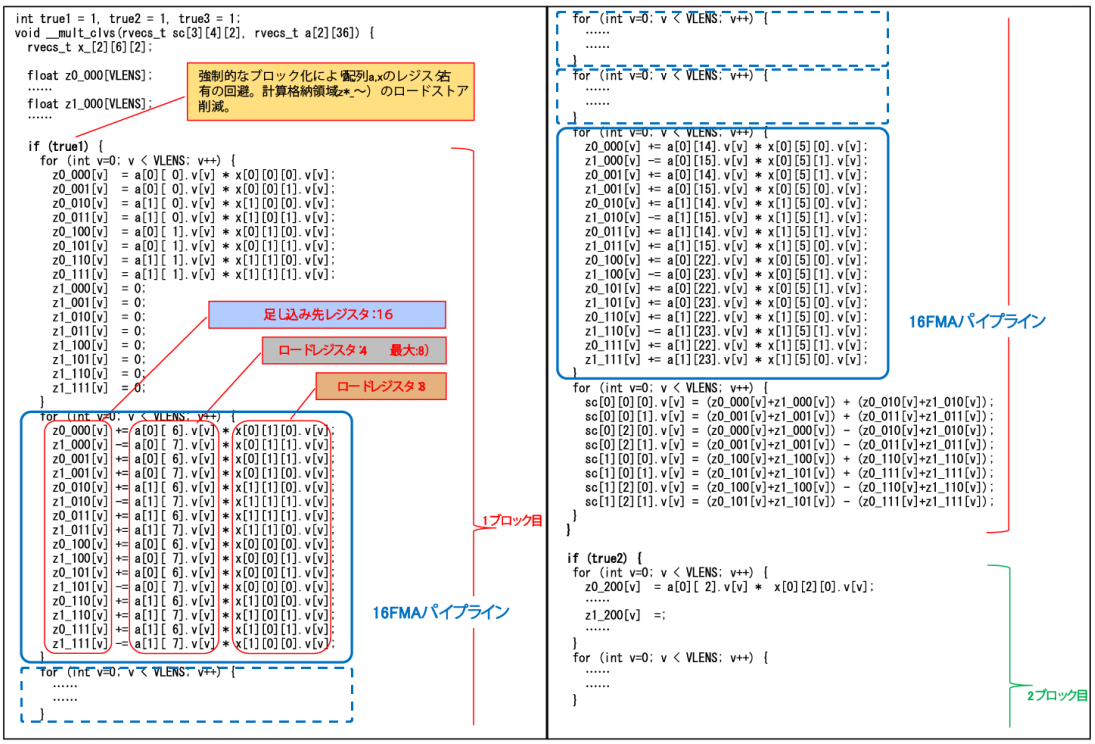


図 4.107: __mult_clvs の最適化後のコード

4.10.2.9　整数レジスタスピル/フィルの抑止

詳細設計（3）では、ループ分割およびスレッド並列化軸の変更によるスレッドインバランス解消の最適化を廃止して、ループを単純化した。さらに、__mult_clvs のインライン展開を行い、関数呼び出しのオーバーヘッドをなくした。コンパイラは共通式の結果を保持するレジスタを保持するために、その他の変数に対するレジスタスピル/フィルを生成する。そこで、ループ内の共通式に現れる変数のライブレンジが伸びるのを防ぐために、基の変数を (asm(""":"+r"(x))) のように書き換えることで共通式を無効化した。さらに、できるだけ共通ベースレジスタ+即値でアドレッシングできるよう、ソース上でベースと即値を明示的に分けて記述した。その結果、コンパイラにより整数レジスタのスピル/フィルが抑止され、実行効率が上がった。この最適化を採用するほうが、第 4.10.2.3 の「ループ分割およびスレッド並列化軸の変更によるスレッドインバランスの解消」よりも高速であったため、最新版のコードでは「ループ分割およびスレッド並列化軸の変更によるスレッドインバランスの解消」の最適化を採用していない。

4.10.2.10　命令レベルの並列性を高めるために演算をソース上でスケジュール

演算をソース上でスケジュールし、命令レベルの並列性を高め、効率を向上させることができる。図 4.110 に変更前と変更後のソースコードを示した。この変更は詳細設計（3）で行った。

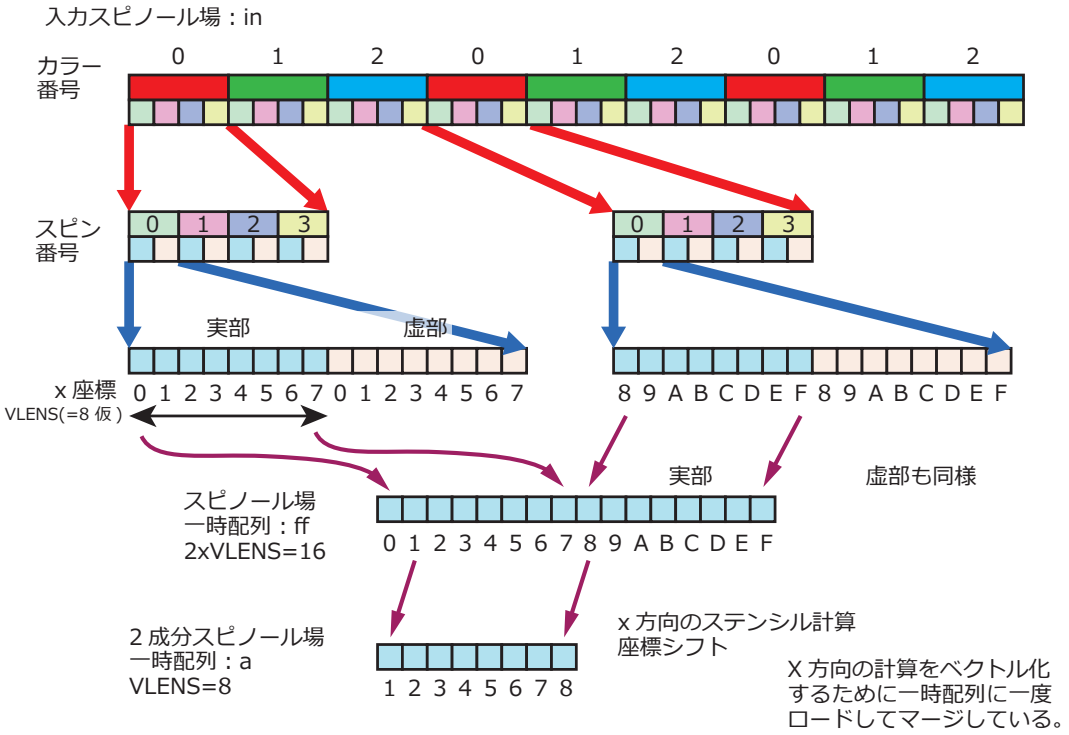


図 4.108: LQCD の X 方向のステンシル差分計算の概念図（最適化前）

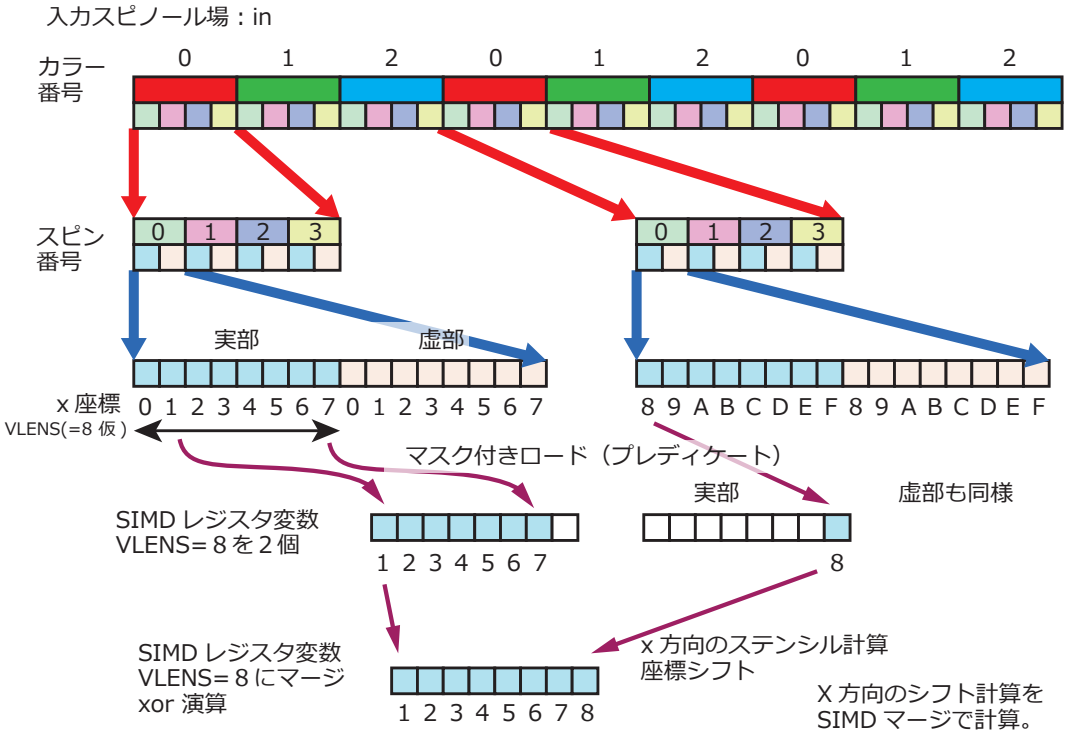


図 4.109: LQCD の X 方向のステンシル差分計算の概念図（最適化後）

4.10.2.11　問題サイズのコンパイル時定数化

コンパイル時に問題サイズを定数で与えることで、実行効率が少し上がった。この変更は詳細設計（3）で行った。

Before tuning

```
#define __mult_u_y_(upy,py,u) {\\
 for (int c = 0; c < 3; ++c) {\
 for (int s = 0; s < 2; ++s) {\
  for (int j = 0; j < VLENS; ++j) {\
   (upy).c[c][s][0][j]  = (u).c[0][c][0][j] * (py).c[0][s][0][j];\
   (upy).c[c][s][1][j]  = (u).c[0][c][0][j] * (py).c[0][s][1][j];\
   (upy).c[c][s][0][j] += (u).c[1][c][0][j] * (py).c[1][s][0][j];\
   (upy).c[c][s][1][j] += (u).c[1][c][0][j] * (py).c[1][s][1][j];\
   (upy).c[c][s][0][j] += (u).c[2][c][0][j] * (py).c[2][s][0][j];\
   (upy).c[c][s][1][j] += (u).c[2][c][0][j] * (py).c[2][s][1][j];\
   (upy).c[c][s][0][j] -= (u).c[0][c][1][j] * (py).c[0][s][1][j];\
   (upy).c[c][s][1][j] += (u).c[0][c][1][j] * (py).c[0][s][0][j];\
   (upy).c[c][s][0][j] -= (u).c[1][c][1][j] * (py).c[1][s][1][j];\
   (upy).c[c][s][1][j] += (u).c[1][c][1][j] * (py).c[1][s][0][j];\
   (upy).c[c][s][0][j] -= (u).c[2][c][1][j] * (py).c[2][s][1][j];\
   (upy).c[c][s][1][j] += (u).c[2][c][1][j] * (py).c[2][s][0][j];\
  }\
 }\
 }\
}
```

After tuning

```
#define __mult_u_y_vec_(upy,py,u) {                    \
   vecs_t tmp0, tmp1, tmp2;                            \
   tmp0 = fload1_s(pt, (u).c, dims_glus, 0, 0, 0);     \
   tmp1 = fload1_s(pt, (u).c, dims_glus, 0, 1, 0);     \
   tmp2 = fload1_s(pt, (u).c, dims_glus, 0, 2, 0);     \
   upy##_0_0_0 = fmul_s(pt, tmp0, py##_0_0_0);         \
   upy##_0_0_1 = fmul_s(pt, tmp0, py##_0_0_1);         \
   upy##_0_1_0 = fmul_s(pt, tmp0, py##_0_1_0);         \
   upy##_0_1_1 = fmul_s(pt, tmp0, py##_0_1_1);         \
(snip)
   upy##_0_0_0 = fnmadd_s(pt, tmp0, py##_2_0_1, upy##_0_0_0);  \
   upy##_0_0_1 = fmadd_s(pt, tmp0, py##_2_0_0, upy##_0_0_1);   \
   upy##_0_1_0 = fnmadd_s(pt, tmp0, py##_2_1_1, upy##_0_1_0);  \
   upy##_0_1_1 = fmadd_s(pt, tmp0, py##_2_1_0, upy##_0_1_1);   \
   upy##_1_0_0 = fnmadd_s(pt, tmp1, py##_2_0_1, upy##_1_0_0);  \
   upy##_1_0_1 = fmadd_s(pt, tmp1, py##_2_0_0, upy##_1_0_1);   \
   upy##_1_1_0 = fnmadd_s(pt, tmp1, py##_2_1_1, upy##_1_1_0);  \
   upy##_1_1_1 = fmadd_s(pt, tmp1, py##_2_1_0, upy##_1_1_1);   \
   upy##_2_0_0 = fnmadd_s(pt, tmp2, py##_2_0_1, upy##_2_0_0);  \
   upy##_2_0_1 = fmadd_s(pt, tmp2, py##_2_0_0, upy##_2_0_1);   \
   upy##_2_1_0 = fnmadd_s(pt, tmp2, py##_2_1_1, upy##_2_1_0);  \
   upy##_2_1_1 = fmadd_s(pt, tmp2, py##_2_1_0, upy##_2_1_1);   \
 }
```

図 4.110: LQCD の命令レベルの並列性を高めるための最適化前後のソースコード

4.10.2.12　リダクションの効率化

反復解法にあらわれるリダクションを含む

リダクションを含む多重ループ

```
for(i=0; i<n; i++)
   for(j=0; j<m; j++)
      s += a[i][j];
```

のようなループに対して、そのままでは

リダクション最適化前のリダクションを含む多重ループ

```
for(i=0; i<n; i++) {
   vector = {0,0, ...};
   for(j=0; j<m/VLEN; j+=VLEN) {
      vector += {a[i][j], a[i][j+1], ...};
   }s += sum(vector);
}
```

のように外側ループの中でベクトル要素間リダクションが行われてしまうため、ベクトル組み込み関数を用いて以下の構造のループを直接記述した。

リダクション最適化後のリダクションを含む多重ループ

```
vector = {0,0, ...};
for(i=0; i<n; i++) {
   for(j=0; j<m/VLEN; j+=VLEN) {
    vector += {a[i][j], a[i][j+1], ...};
   }
}
s = sum(vector);
```

この変更は詳細設計（4）で行った。

4.10.2.13　演算区間のチューニングのまとめ

以下に各演算区間のチューニングをまとめる。

- ddd_in_s_
  - ARM C Language Extensions による最適化
  - 整数レジスタスピル/フィルの抑止
  - 命令レベルの並列性を高めるために演算をソース上でスケジュール
  - ループ分割およびスレッド並列化軸の変更（詳細設計（3）でより良い最適化方法が見つかったため最新版のコードでは行われていない）
  - 手動プリフェッチ挿入
  - 作業配列 tmp の削除
- jinv_ddd_in_s_
  - ddd_in_s_と同様のチューニング
  - ddd_in_s_以外のループ処理への手動プリフェッチ挿入
- ddd_out_pre_s_
  - ddd_in_s_と同様のチューニング
- ddd_out_pos_s_
  - ddd_in_s_と同様のチューニング
- other_calc
  - 手動プリフェッチ挿入
  - リダクションの効率化
- __mult_clvs

  - ddd_in_s_、jinv_ddd_in_s_、ddd_out_s_の内部でクローバー項の演算に使われている
  - 互いに独立な演算の順序を並べ替えることによる演算レイテンシ隠蔽
  - 演算順序の並べ替えによるレジスタスピル/フィルの削減
- 全演算区間共通
  - 更新配列のアラインメント調整
  - パラレルリージョン拡大
  - 問題サイズのコンパイル時定数化

4.10.2.14　FP16 の利用

QCD クォークソルバーでは反復法を用いてクォークの運動方程式に該当する大規模連立方程式を解いている。倍精度の結果を得るために単精度を利用した混合精度の解法が組み込まれている。単精度を利用することでキャッシュ効率やメモリバンド幅、倍精度（以下 FP64）より 2 倍速い単精度計算を利用して計算効率を上げている。富岳では、半精度（以下 FP16）浮動小数点実数を単精度（以下 FP32）の 2 倍の SIMD 幅で実行できる機能が利用でき、さらなる高速化が期待できる。

詳細設計（4）において、QCD クォークソルバーに FP16 も利用した混合精度ソルバーを用いることを検討し、FP16、FP32、FP64 の 3 種混合精度反復法のアルゴリズム検証を行った。一般に混合精度を用いると倍精度のみを用いた場合に比べて収束までの反復回数が伸びる傾向がある。FP16 を利用した混合精度ソルバーの収束までの反復回数の振る舞いを調べておくことが重要である。FP16 では内積などのノルムの計算での有効数字が確保しにくいことから定常反復ソルバーにおいて利用することとした。QCD クォークソルバーで用いている FP64-FP32 混合精度ソルバーの中では FP64 反復法ソルバーに対する前処理行列の計算に FP32 反復法ソルバーを用いている。さらに、FP32 反復法ソルバーでも前処理を行なっており、前処理行列として固定反復法による一定回数反復し得られた近似解行列を用いている。この FP32 固定反復法の部分を全て FP16 に置き換えたものを FP64-FP32-PF16 混合精度ソルバーとして QCD クォークソルバー全体の収束までの振る舞いを調べた。C/C++言語での FP16 による実装には IEEE-754 に準拠したクラスライブラリ `half-1.12.0` (`http://half.sourceforge.net/`) を利用した。格子サイズ 32^4 のウィルソンゲージ作用 $\beta = 6.00$ のクエンチ近似のゲージ配位 1 つの上で 5 つのクォーク質量パラメータ $\kappa = 0.1324, 0.1333, 0.1342, 0.1348, 0.1350$ での QCD クォークソルバーの収束までの反復回数を調べた。これらのパラメータでは擬スカラー中間子質量 $aM_{\rm PS}$ が κ のそれぞれの値に対して $aM_{\rm PS} \simeq 0.100, 0.090, 0.065, 0.035, 0.010$ となる。図 4.111 は、それぞれのクォーク質量パラメータについて、QCD クォークソルバーの残差を反復回数の関数としてプロットした図である。図の左側が FP64-FP32-PF16 混合精度ソルバーのもので、図の右側が FP64-FP32 混合精度ソルバーのものである。収束までの反復回数は、クォーク質量（擬スカラー中間子質量）が軽くなると FP64-32-16 混合精度での回数は FP64-32 混合精度での回数に比べて徐々に伸びる様子が見られ、$\kappa = 0.1348$ の時は約 17%、$\kappa = 0.1350$ の時は約 26%程度伸びた。FP16 の演算性能が FP32 に比べてこの反復回数の伸びを上回るほど速くなっていれば収束までの時間は速くなる可能性がある。

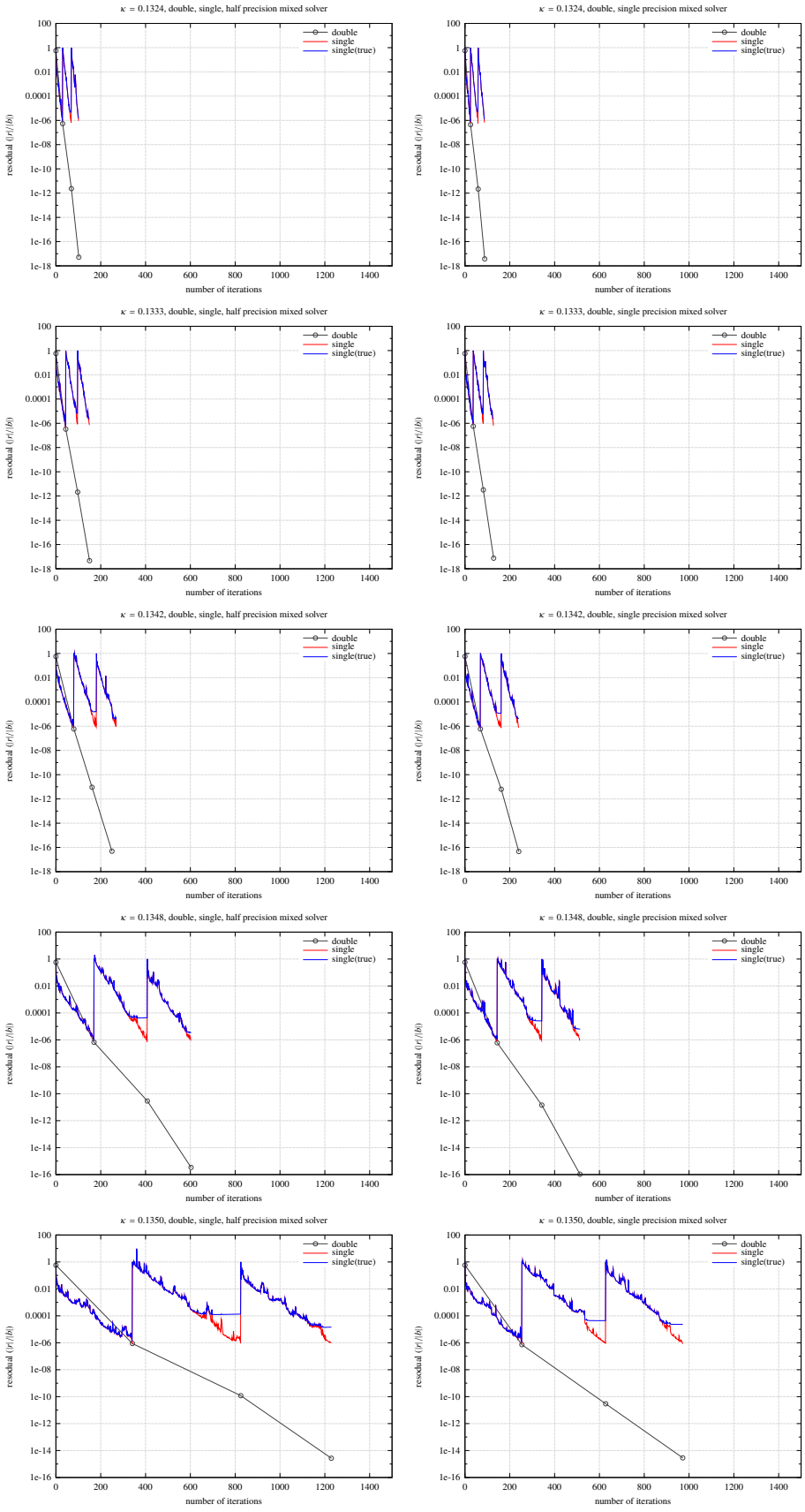


図 4.111: LQCD の混合精度ソルバーの残差を反復回数の関数としてプロットした図。

ソフトウエア調整（１）において、試作機を用いて FP16 の試験を行った。FP16 を用いたコードは trad モードではコンパイルできないため、clang モードでコンパイルし、計算結果に問題がないことを確かめた。FP16 を用いた単純なコードでは SVE のベクトル命令が生成されることが確認できたが、QCD コードではスカラ命令が生成された。また、現在 clang モードは trad モードと比較して最適化が不十分であるため、実際の利用において有効かどうかは今後しらべる必要がある。

なお、FP16 の SIMD ベクトル長が FP32 の倍であることはまだ利用できていない。ターゲット問題の格子サイズ 192^4 ではプロセスあたりの格子サイズで用いるベクトル長 (X 方向の格子サイズの半分) がすでに FP32 SIMD ベクトル長と同じ 16 となっており、FP16 の SIMD ベクトル長 32 をこの格子サイズで埋めることが出来ない。この事についての改善方法は検討中である。一方、ターゲット問題とは異なる小ノード数での実行時にはプロセスあたりの格子サイズが大きくなるので FP16 SIMD ベクトル長の 32 を埋めることができる場合がある。この場合に動作する高速版の FP16 固定反復ソルバーを作成する予定である。

4.10.2.15　ダブルバッファリング

システム全系で 192^4 の問題を解くために、MPI 並列化を行う。本アプリケーションは構造格子ステンシル計算に属する問題であるので、隣接プロセスとのステンシル計算の袖の計算のデータの交換が必要となる。

本分割では MPI の 1 プロセスあたりの格子点数が比較的少ないため隣接通信の効率が全体性能に大きな影響を与える。本アプリケーションではこの影響を最大限抑えるため、通信と計算の重ね合わせによる通信時間の隠蔽を実装している。具体的にはプロセスあたりの形状 $n_x \times n_y \times n_z \times n_t$ を更に 2 分割し $(n_x/2) \times n_y \times n_z \times n_t$ にして、これを小領域の単位として (A) 小領域の内部のステンシル計算と、(B) 小領域間を繋ぐステンシル計算に計算を分割する。小領域間を繋ぐステンシル計算部分は更に、(B-1) データ送信部、(B-2) データ受信しステンシル計算する、の 2 つの部分に分割される。(C) データの送受信部、は MPI のノンブロッキング手続き、もしくは RDMA アクセスにより実現することで (B-1)→(A)+(C)→(B-2) のように (A) と (C) を同時に行うことで通信時間の隠蔽が実現される。

本アプリケーションでは既に通信の隠蔽は実装されているが、富岳の場合の細粒度 (大規模並列) 並列の場合に、データの送受信に関わる確認手続き (送信終了と受信終了の確認) の時間の影響が大きいことが分かった。この部分は (B-1) ならびに (B-2) の部分で行われる手続きであり計算による隠蔽は出来ない。そこで、送信終了と受信終了の確認を遅らせることの出来るアルゴリズムであるダブルバッファリング法を検討した（図 4.112)。

QCD クォークソルバーの解くべき連立方程式の係数行列（ステンシル計算）は周期境界条件であり、4 次元の各方向に対して隣接領域の交換が発生する。隣接領域の交換のための送受信バッファを用意し、MPI によりデータを交換するのが通常である。本コデザインではこの交換に専用の RDMA 通信とダブルバッファリング法を組み合わせることでレイテンシーを最小にすることを考える。

まず各 MPI ランクは通信リンク毎に送信バッファを一つ受信バッファを二つもつ。4 次元の周期境界条件ではリンクは 8 方向である。二つの受信バッファにはそれぞれパリティが付加される。各 MPI ランクはバッファ指定のためのパリティ状態変数 P を持つ。隣接領域の交換は全 MPI ランクで同時に行われるのでこのパリティ状態変数 P は全 MPI ランクで共通の値を持つと仮定する。

1. 送信データを送信バッファに書き込む。後述する **uTofu** 版ではバッファ末尾の状態フラグを送信毎に更新する。
2. RDMA Put 通信を用いて送信バッファのデータを送信先の受信バッファに転送する。Put 通信は非同期／ノンブロッキングに行われる。データの最後尾が到達すると全てのデータが受信完了となるモードで Put 通信を行う。
3. 通信と関連しない計算を行う。
4. ローカルの受信バッファにデータの最後尾が到着するのを待つ。これによりすべてのデータの受信が完了している。データ最後尾の到着は、受信バッファ末尾に用意した状態フラグの更新で検出する。
5. 受信バッファのデータを利用した計算を行う。
6. 受信バッファの末尾に用意している状態フラグの状態を「空き」にする。**uTofu** 版ではこのステップを省く。
7. RDMA Put 通信の完了を確認する。確認できたらパリティ状態変数を反転する。この後、送信バッファが変更可能となる。

以前の版では、冒頭に送信先の受信バッファの空きを待つステップ（空き状況の問い合わせには RDMA Get 通信を利用）が記載されていたが、詳細に検討した結果、不要であることがわかった。またアルゴリズムの検証のために、LQCD と同様なステンシル計算である 2 次元のラプラス方程式をヤコビ反復法で解くコードを実装し、以下の検証を行った:

- 上記の受信バッファの空きを待つステップがなくても正しく機能すること。
- パリティ状態変数の反転のタイミングも適切であること。

検証コードは複数の実装を用意して FX100 上で比較し、オーバーヘッドが十分に小さく通信バンド幅律速になるものを探した。用意した実装の詳細は以下の通り（比較の便宜のため、**uTofu** を用いた実装も記述している）:

1. MPI-1 の持続通信を用いたシングルバッファリング（以下 SB）およびダブルバッファリング（DB）。`FJMPI` 版の関数があるものはそちらを利用した。なお受信バッファのデータポーリングに `MPI_Waitall` 関数を用いるため、受信バッファの空き状況を示す状態フラグは不要である。

   利用関数: `FJMPI_Prequest_recv_init()`, `FJMPI_Prequest_send_init()`, `FJMPI_Prequest_startall()`, `MPI_Waitall()`。

2. MPI の片方向通信を受動モードで用いた SB、DB。バッファの同期には、MPI-3 の関数群を用いた。受信側では `MPI_Win_allocate` 関数で割り付けたメモリ領域をあらかじめ `MPI_Win_lock` 関数で送信元に公開し、受信内容とメモリとの同期には `MPI_Win_sync` 関数を用いた。送信側での送信完了には `MPI_Win_flush_local` 関数を用いた。`MPI_Win_sync` 関数を `MPI_Win_flush_local` より後ろに記述する必要があり、図 4.112 では “Check local recive buffer” の前に “Put completed” が入ることになる（Parity 変更のタイミングは図と同じ）。

表 4.115: LQCD のダブルバッファリングにおける、各ステップでの使用関数

実装	STEP A	STEP B	STEP C
1	FJMPI_Request_startall	MPI_Waitall	MPI_Waitall
2	MPI_Put	MPI_Win_sync および recieve buffer 内のデータの確認	MPI_Win_flush_local
3	FJMPI_Rdma_put	recieve buffer 内のデータの確認	FJMPI_Rdma_poll_cq
4	utofu_put	recieve buffer 内のデータの確認	utofu_poll_tcq

利用関数: MPI_Win_allocate(), MPI_Win_lock()（MPI_LOCK_EXCLUSIVE モードで利用）, MPI_Put(), MPI_Win_flush_local(), MPI_Win_sync()

3. 拡張 RDMA インターフェースによる DB。京での LQCD で使われていた rdma_comlib を DB 化して用いている。

   利用関数: FJMPI_Rdma_reg_mem(), FJMPI_Rdma_put(), FJMPI_Rdma_poll_cq()

4. **uTofu** を用いた実装。上記拡張 RDMA 版を基にしている。受信確認フラグの仕様に変更有り（後述）。

   利用関数: utofu_create_vcq(), utofu_query_vcq_id(), utofu_reg_mem(), utofu_dereg_mem(), utofu_poll_tcq(), utofu_poll_mrq(), utofu_free_vcq(), utofu_put()

ダブルバッファリングに関して、上記の使用関数と使用箇所（図 4.112 参照）を、表 4.115 にまとめた。

検証の結果、アルゴリズムの正しさは確認できた。性能では MPI 片方向通信のもの（実装 2.）は通信バンド幅を使い切ることはできなかったため[9]、今後の実装手段としては不適と判断した。その他の実装は通信バンド幅律速になり、SB と DB 間で速度に有意な差は見られなかった。FX100 上では僅差で富士通 RDMA 版が最速となった。**uTofu** については上記アルゴリズムの検証のために十分な機能が実装されていなかったため、FX100 上での検証には用いていない。

アルゴリズムの検証後、qws-0.1.8（MPI による SB を利用）に上記実装 1.（MPI-1 持続通信）と 3.（富士通 RDMA）を用いたダブルバッファリングを組み込んだものを用意した。qws-0.2.2 以降からは、富士通 RDMA 版を 富岳向けの **uTofu** に置き換えている。

[9]MPI 片方向通信の仕様から来る必然的な結果なのか、MPI の実装の問題なのかの切り分けはしていない。

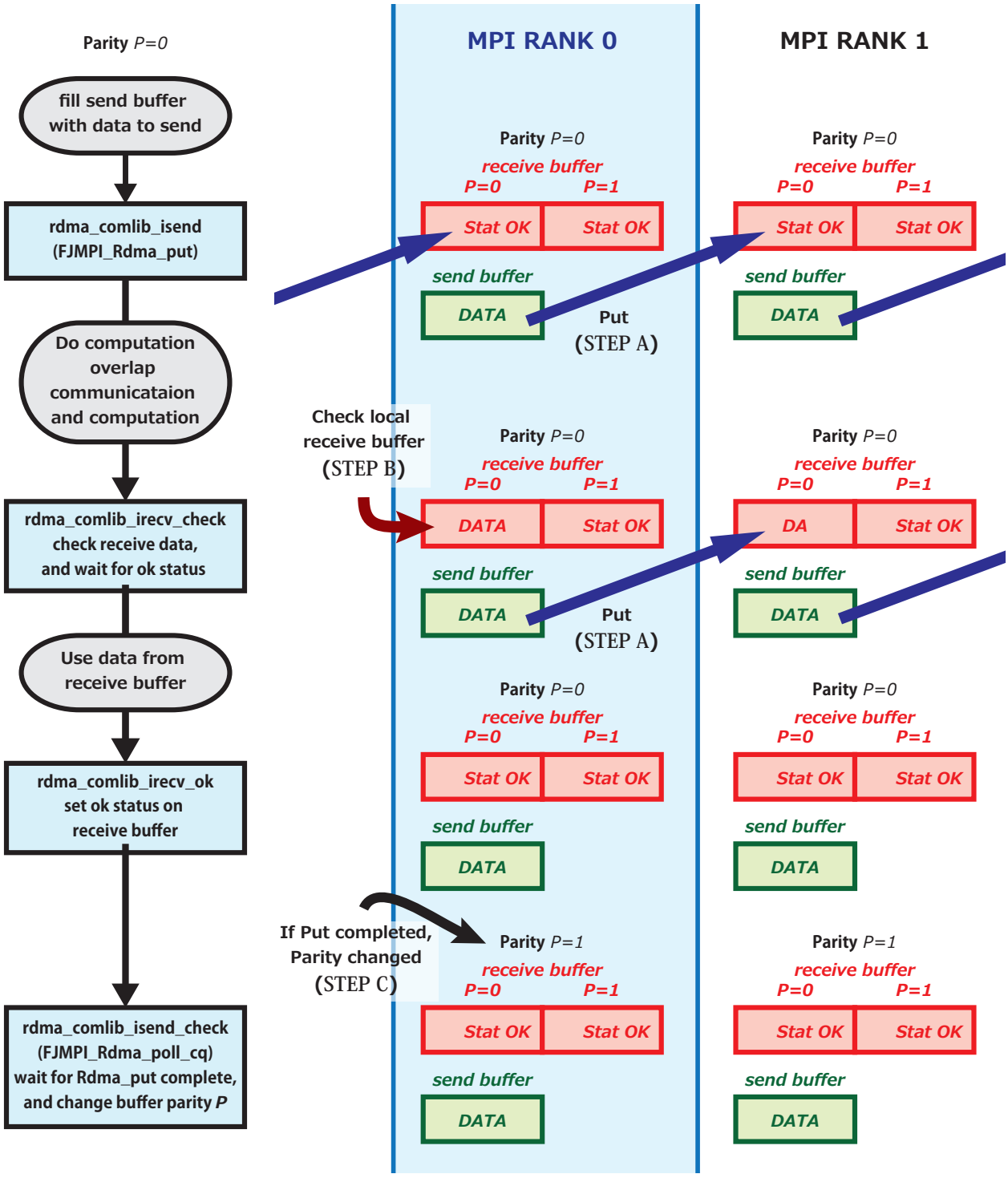


図 4.112: ダブルバッファリングの状態遷移を表す図。左端は、富士通 RDMA を用いた時の利用関数。

4.10.2.16　uTofu

LQCD で用いる 4 次元格子上では最大 8 方向への通信が必要であり、領域分割法の下では同時に 7 方向への通信が発生する。**uTofu** はそのような同時通信を可能にする通信インターフェース

を持つため富岳の性能を引き出すのにもっとも適していると期待しており、実機での実装方法の比較・検証は重要である。なお **uTofu** の利用は次節のプロセス配置と TNI の割り当てや 4.10.2.18 のプロセスマッピングにも関連している。

ここでは、qws-0.2.4 + 評価環境 II 版での実装と、得られた性能の概略を述べる。

vcq の使用法 **uTofu** を用いた通信指示は、仮想化したコントロールキュー (vcq) 経由で行う必要がある。各方向に一つずつ計 8 個の vcq を用意した。ある方向への送信と、その方向からの受信は同一の vcq が担う。アプリケーション上の隣接ランクへの vcq id の通知は、MPI 通信で行う。vcq は `UTOFU_VCQ_FLAG_THREAD_SAFE` フラグを用いてスレッド安全に構築し、put 通信時は最大 8 個の put をスレッド並列に発行している。この並列化で、メモリから TNI へのデータ転送（および領域分割時のランク内隣接領域へのデータコピー）を高速化できる。

キューのポーリング vcq に溜まった送信に関する通知（TCQ）・受信に関する通知（MRQ）は適宜ポーリングして取り除く必要がある。送信に関する通知は、put 通信の完了確認時の TCQ に対するポーリングで自動的に取り除かれる。MRQ 内の通知は、受信データ待ちの冒頭にポーリングを追加して取り除く。

受信確認フラグの仕様変更 **uTofu** 版では、受信確認フラグの仕様を変更した。受信データは、デフォルトでは一旦メモリに書き込まれるが、これを直接キャッシュに書き込むキャッシュインジェクションに対応するためである[10]。キャッシュインジェクションのためには、受信バッファの適切なアラインメントに加えて受信バッファに受信以外の書き込みがあってはならない。そのため、受信バッファ内の状態フラグを受信側が操作して「未受信」状態に更新するのではなく、送信側が送信毎にフラグの値を変える仕様に変更した。受信側は前回の受信完了時のフラグの値を覚えておき、バッファ内フラグ値の変化で受信完了を判定する。

プロセスマッピング プロセスマッピング（ランクマップ）の通知にも qws-0.2.4 では **uTofu** を用いた。後には MPI 通信を用いるように変更したが、ここでは当初の **uTofu** 版について述べる。

各ランクが必要なのは、4 次元 LQCD 座標の意味での隣接ランクのランク番号である。隣接ランクの 6 次元 Tofu 座標は自ランクの Tofu 座標を元に計算できるが、ランク番号は直ちには分からない。qws-0.2.4 では **uTofu** の Atomic Read Modify Write (ARMW) 通信を用いて隣接ランクとランク番号を交換している。ARMW 通信に必要な隣接ランクの vcq id は、`utofu_create_vcq_with_cmp_id()` 関数で計算している。この計算で得た vcq id を持つ vcq を生成するには別途コントロール・キュー（cq）の値も必要だが、この値を得る方法は **uTofu** の仕様では定めがなく、`utofu_create_vcq_with_cmp_id()` 関数の振る舞いから推測した[11]。

uTofu を用いたダブルバッファリングの動作検証には、前節でアルゴリズムの検証に用いた 2 次元ラプラス方程式をヤコビ法で解くプログラムを用いた。通信バンド幅をほぼ理論通りに使い切れること、実機 3 ラックまで極めて良好に弱スケールすること、を確認できた。環境は実機評

[10] フラグ仕様変更の検証目的のために、qws-0.2.4 では put 時の指定でキャッシュインジェクションを無効にしている。

[11] 言語環境 lang/tcsds-1.2.27b で推測値では vcq の作成に失敗したため、ARMW 通信をやめて MPI 通信に切り替えた。切り替え後は、`MPI_Allreduce` を用いて全ランクに LQCD で必要な 4 次元のランク座標と 6 次元 Tofu 座標のテーブルを通知している。

価環境 II で、2020 年 2 月版コンパイラを用いた。なお同環境での MPI 版では概ね弱スケールするものの、速度は半分程度であった。

uTofu を用いた qws-0.2.4 + 評価環境 II 版（コンパイラは 2019 年 12 月版）は、36 ラックまで概ね弱スケールすることは確認できた。しかし、実行速度は 3 ラックまでは 1 ノード時とほぼ同じでスケールしたが、それ以上になると約半分に低下した[12]。なお、同環境下でランクマップを用いない MPI 版は全般的に弱スケールしなかった。ランクマップを用いた MPI 版は、16 ラックまでは概ね弱スケールするものの **uTofu** 版より 4 割程度遅く、36 ラックでは弱スケールしなかった。

4.10.2.17　プロセス配置と TNI 割り当ての最適化

隣接通信の実行時間に関わるプロセス配置と TNI 割り当ての探索方法について説明する。この方法では、ボトルネックとなる Tofu リンクもしくは TNI の転送メッセージ長が最小となるものを 1 つ探索する。以降の説明ではプロセス割り当ての空間を $\{T_1, T_2, \ldots, T_n\}$ と表現する。T_i はネットワークのそれぞれの軸であり、

$$
\begin{aligned}
&T_i = (t_i, s_i, o_i, h_i)\,, \\
&t_i \in \{\mathrm{TX}, \mathrm{TY}, \mathrm{TZ}, \mathrm{TA}, \mathrm{TB}, \mathrm{TC}, \mathrm{IN\_NODE}\}\,, \\
&s_i \in \mathbb{N}\,, \quad o_i \in \{\mathrm{torus}, \mathrm{mesh}\}\,, \quad h_i \in \mathbb{N}
\end{aligned}
\tag{4.13}
$$

と表現する。t_i は Tofu の物理軸を示す。ただし、1 ノードに複数のプロセスを割り当てる場合はその軸を IN_NODE と表現する。s_i は T_i の長さである。o_i は T_i の端の座標同士が隣接であれば torus、そうでなければ mesh である。h_i は T_i の軸で見た隣の座標について、t_i の軸で見たときの距離である。例えば、長さ 24 の物理 X 軸 (24 でシステム全体、すなわちトーラスとする) は (TX, 24, torus, 1) と表現される。長さ 24 の物理 X 軸について、仮想的に長さ 2 の TXc、長さ 12 の TXd の軸に分けて扱い、TX の座標 x が TXc の座標 x mod 2、TXd の座標 $[x/2]$ に対応させる場合、それぞれ (TX, 2,mesh, 1)、(TX, 12, torus, 2) となる。これはソースコードにおける struct TofuAxis に対応する。

手順 1 入力された Tofu 物理形状に対して、それぞれの物理軸について最大 2 分割した組み合わせを列挙する。つまり、長さ s、トーラス性 o の物理軸 t について $s = s_1 \times s_2$ を満たす全ての s_1、s_2 に対して、$\{(t, s_1, \mathrm{mesh}, 1), (t, s_2, o, s_1)\}$ を列挙する。ただし、s_1、s_2 のいずれか 1 のとき $\{(t, s, o, 1)\}$ とする。列挙の結果は、ソースコードにおける変数 `tofu_split_list` に対応する。

手順 2 それぞれの物理軸について手順 1 で列挙した中から 1 つずつ選択し、それらを組み合わせる。そして、全ての組み合わせを P_2 として列挙する。例えば、物理 X、Y、Z 軸の長さがぞれぞれ 24、22、24 のとき、 $p_2 \in P_2$ の例は、{(TX, 24, torus, 1), (TY, 11, mesh, 2), (TY, 2, mesh,1), (TZ, 8, torus, 3), (TZ, 3, mesh, 1), (TA, 2, mesh, 1), (TB, 3, torus, 1,), (TC, 2, mesh ,1), (IN_NODE, 4, mesh,1)} である。P_2 はソースコードにおける変数 `splitted_tofu_axis` に対応する。

手順 3 P_2 の要素を全て処理済みの場合アルゴリズムを終了する。まだ選んでいない $p_2 \in P_2$ について、p_2 からトーラスになるように軸を選んで組み合わせ、プロセス分割の各軸に割り当て

[12] 後に OS ジッターによるものだと判明している。

る、可能な組み合わせを列挙する。つまり、プロセス分割の 4 つの軸の長さを (n_1, n_2, n_3, n_4) とすると、i 番目に対して、

$$C_i = \{(t_{i,1}, s_{i,1}, o_{i,1}, h_{i,1}), \ldots, (t_{i,n}, s_{i,n}, o_{i,n}, h_{i,n})\} \subset p_2 \tag{4.14}$$

の軸を割り当てるとき、(C_1, C_2, C_3, C_4) の全ての組み合わせを列挙する。ただし、C_i は以下を満たす。

$$\begin{aligned} &n_i \leq \prod_j s_{i,j}\,, \\ &t_{i,1}, \ldots, t_{i,n}\text{に IN\_NODE を除き 2 種類以上が存在、もしくは } o_{i,j} = \text{torus が存在} \end{aligned} \tag{4.15}$$

ここでは、n_i が偶数であることと、$\prod_j s_{i,j}$ ($t_{i,j}$=IN_NODE である j を除く) が奇数であっても、そのうち n_i 個の頂点を通るトーラスは構成できるものと仮定する。また、$t_{i,j} = t_{i,k}$ となる j, k があるような組み合わせは、その物理軸を分割しない組み合わせと等価のため除外してよい。たとえば、プロセス数 6 の軸に対して、{(TY, 2,mesh, 1), (TZ, 3,mesh, 1)} を割り当てることができる。列挙の結果を P_3 とする。P_3 は、ソースコードにおける変数 `tofu_axis_alloc_list` に対応する。

手順 4 P_3 の要素を全て処理済みの場合手順 3 に戻る。まだ選んでいない $p_3 \in P_3$ について、Tofu 軸とシミュレーションのプロセス分割の軸の組み合わせと転送メッセージ長の対応 F を計算する。F の要素は

$$\begin{aligned} &(t, q, d, m) \in F\,, \\ &t \in \{\mathrm{TX}, \mathrm{TY}, \mathrm{TZ}, \mathrm{TA}, \mathrm{TB}, \mathrm{TC}, \mathrm{IN\_NODE}\}\,, \\ &q \in \{\mathrm{QX}, \mathrm{QY}, \mathrm{QZ}, \mathrm{QT}\}\,, \\ &d \in \{\mathrm{plus}, \mathrm{minus}, \mathrm{both}\}\,, \\ &m \in \mathbb{N} \end{aligned} \tag{4.16}$$

と表現する。t は物理軸、q は LQCD のプロセス分割の軸、d はプロセス分割の軸で見たときの送信方向、m はメッセージ長を表す。これは、ソースコードにおける `struct LinkFlow` に対応する。プロセス分割の軸 q に対して、

$$\{(t_1, s_1, o_1, h_1), \ldots, (t_n, s_n, o_n, h_n)\} \tag{4.17}$$

を割り当てるとき、以下の手順で F を計算する。

手順 4.1 q の軸において、ノードあたりの、ノード間に通信するプロセス数 n_{inter} とノード内に通信するプロセス数 n_{intra} を求める。全体のノードあたりプロセス数を n と置く。$t_i = \mathrm{IN\_NODE}$ がある場合 $n_q = s_i$、ない場合 $n_q = 1$ と置く。このとき、

$$\begin{aligned} n_{\mathrm{inter}} &= \frac{n}{n_q}\,, \\ n_{\mathrm{intra}} &= \frac{(n_q - 1) \times n}{n_q} \end{aligned} \tag{4.18}$$

となる。

手順 4.2 q の軸におけるプロセスあたりの送信メッセージ長を m_q と置く。$t_i \neq$ IN_NODE のとき、
$(t_i, q, \mathrm{both}, n_{\mathrm{inter}} \times m_q \times h_i) \in F$ とする。ホップによる他ノードへの転送分を含めるため、h_i を掛ける。$t_i =$ IN_NODE のとき、$(\mathrm{IN\_NODE}, q, \mathrm{both}, n_{\mathrm{intra}} \times m_q)$ とする。F はソースコードにおける変数 `link_flow_list` に対応する。

手順 5 手順 4 で得た F に対して、Tofu 物理軸毎の合計転送メッセージ長を計算する。ただし、IN_NODE を除く。物理軸 t の合計転送メッセージ長は、全ての $(t, q, d, m) \in F$ について m を合計した値である。全ての物理軸について計算した結果は、ソースコードにおける変数 `link_flow_size` に対応する。この中の最良値が、今まで求めた他の割り当てによる最良値を上回る場合、手順 4 に戻る。

手順 6 手順 4 で得た F に対して、6 つの TNI を割り当てる可能な組み合わせを列挙する。物理軸 t について、$(t, q_i, \mathrm{both}, m_i) \in F$ のとき、TNI a, b ($a = b$ でもよい) に対して手順 6a か手順 6b のいずれかの方法で割り当てることができる。これらの方法は、ある Tofu リンクから見て、そのリンクを使う TNI が唯一となる制限を満たす。a, b に対して割り当てられた通信の集合をそれぞれ I_a, I_b とする。6 つの TNI はそれぞれ区別する必要がないため、$(I_1, \ldots, I_6)$ の代わりに、$K = \{(I'_1, n_1), \ldots, (I'_m, n_m)\}$ と表現することができる。ただし、n_i は、$I'_i = I_j$ となる j の数である。こうすることで、割り当てる TNI を選ぶとき、$|K|$ 通りに減らすことができる。K の要素の表現は、ソースコードにおける `typedef TNI_Alloc` に対応する。

手順 6a t が IN_NODE のとき、もしくは q_i のうち 1 回の隣接通信で両方向に通信する軸が丁度 1 つのとき ($q_i \in$ {QY,QZ,QT} となる i が 1 つのとき)、異なる q_i 毎に、$(t, q_i$,plus, $m_i) \in I_a$、$(t, q_i$, minus, $m_i) \in I_b$ もしくは $(t, q_i$, minus, $m_i) \in I_a$、$(t, q_i$, plus, $m_i) \in I_b$ と割り当てる。これは、前者の場合、q_i の正方向に対する通信は、t の方向によらず I_a を、q_i の負方向に対する通信は、t の方向によらず I_b を使うことを意味する。ただし、$q_i =$ QX かつ $a = b$ のとき、q_i の通信が同時に両方向発生することがないため、plus, minus のどちらか片方のみ割り当てるものとする。例えば、$\{(t, \mathrm{QX}, \mathrm{both}, m_x), (t, \mathrm{QY}, \mathrm{both}, m_y)\} \subset F$ のとき、以下の割り当てがそれぞれできる。

A) $\{(t, \mathrm{QX}, \mathrm{plus}, m_x), (t, \mathrm{QY}, \mathrm{plus}, m_y)\} \subset I_a, \{(t, \mathrm{QX}, \mathrm{minus}, m_x), (t, \mathrm{QY}, \mathrm{minus}, m_y)\} \subset I_b$

B) $\{(t, \mathrm{QX}, \mathrm{plus}, m_x), (t, \mathrm{QY}, \mathrm{minus}, m_y)\} \subset I_a, \{(t, \mathrm{QX}, \mathrm{minus}, m_x), (t, \mathrm{QY}, \mathrm{plus}, m_y)\} \subset I_b$

C) $\{(t, \mathrm{QX}, \mathrm{minus}, m_x), (t, \mathrm{QY}, \mathrm{plus}, m_y)\} \subset I_a, \{(t, \mathrm{QX}, \mathrm{plus}, m_x), (t, \mathrm{QY}, \mathrm{minus}, m_y)\} \subset I_b$

D) $\{(t, \mathrm{QX}, \mathrm{minus}, m_x), (t, \mathrm{QY}, \mathrm{minus}, m_y)\} \subset I_a, \{(t, \mathrm{QX}, \mathrm{plus}, m_x), (t, \mathrm{QY}, \mathrm{plus}, m_y)\} \subset I_b$

このとき、A と D、B と C は a, b を入れ替えると同じなので、片方は省略してよい。この割り当て方法は、ソースコードにおける関数 `enum_tni_allocation_body_one_sim_axis_dir` に対応する。

手順 6b t が IN_NODE でないとき、全ての i について、$(t, q_i$, both, $m_i) \in I_a$ と割り当てる。これは、t が片方向のリンクしか持たない場合 (TA,TC のいずれかの場合)、t を通る通信は q_i の方向によらず a を使うことを意味する。両方向のリンクを持つ場合、さらに $(t, q_i$, both, $m_i) \in I_b$ と割り当てる。これは、t の正負のうちある方向を通る通信は q_i の方向によらず a を、もう片方の方向を通る通信は q_i の方向によらず b を使うことを意味する。ここでは t の正負を最後まで区別しなくてよいため、同じ値を割り当てている。ただし、$a = b$ のとき、両方向分存在することを表現するため、さらに $(\{t\}, q_i$, both, $m_i) \in I_a$ とする。この割り当て方法は、ソースコードにおける関数 `enum_tni_allocation_body_both_sim_axis_dir` に

対応する。

手順 7 手順 6 で列挙した割り当てについて、それぞれボトルネックとなる TNI の転送メッセージ長を計算する。割り当て $I = \{(t_1, q_1, d_1, m_1), \ldots, (t_n, q_n, d_n, m_n)\}$ の TNI の転送メッセージ長を手順 7.1、手順 7.2 で計算する。ボトルネックとなる TNI の転送メッセージ長が、今までの最良値より小さい場合は最良値を更新する。手順 7 を終えると、手順 4 に戻る。手順 7 による TNI の転送メッセージ長の計算を手順 6 による割り当ての途中で行い、最良値を超える場合にその時点でその割り当てについて打ち切るようにしてもよい。

手順 7.1 t_i = IN_NODE となる i について、m_i を加算する。

手順 7.2 手順 7.1 で処理したものを除いた I' において、それぞれの $q \in$ {QX,QY,QZ,QT} について、$I_q = \{(t_{q,1}, q, d_{q,1}, m_{q,1}), \ldots, (t_{q,n_q}, q, d_{q,n_q}, m_{q,n_q})\}$、$I' = \cup_q I_q$ に分割する。I_q それぞれについて、$d_{q,i} \in$ {plus, both} となる i のうち、最大の $m_{q,i}$ を加算する。さらに、$d_{q,i} \in$ {minus, both} となる i のうち、最大の $m_{q,i}$ を加算する。最大値のみ加算するのは、同軸同方向の通信は、どれか 1 つのリンクしか使わないためである。

4.10.2.18　効率的なプロセスマッピングと隣接通信で通信の完了を待つ区間の時間推定と検証

詳細設計（4）では、性能電力予測ツールによる性能推定と試作機による性能測定でプロセスあたり格子サイズが異なる。性能電力予測ツールによる性能推定では、プロセスあたり格子サイズが $32 \times 6 \times 6 \times 2$ で、試作機による性能測定では、プロセスあたり格子サイズが $32 \times 6 \times 4 \times 3$ である。それぞれ、隣接通信受信待ち時間は以下のようになる。

プロセスあたり格子サイズ	隣接通信受信待ち時間 [ミリ秒]	
	1 回あたり	全体
$32 \times 6 \times 6 \times 2$	0.018	0.36
$32 \times 6 \times 4 \times 3$	0.012	0.24

以下に、その推定方法を述べる。

4.10.2.18.1　プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$ の隣接通信時間

格子サイズ 192^4 の時、プロセスあたり格子サイズを $32 \times 6 \times 6 \times 2$ とするとプロセス分割は $6 \times 32 \times 32 \times 96$ となる。この時、各方向の転送量は

$$\begin{aligned} \mathrm{QX}_{\pm} \text{ 方向転送量} &= \quad 3456 \text{ バイト} \\ \mathrm{QY}_{\pm} \text{ 方向転送量} &= \quad 9216 \text{ バイト} \\ \mathrm{QZ}_{\pm} \text{ 方向転送量} &= \quad 9216 \text{ バイト} \\ \mathrm{QT}_{\pm} \text{ 方向転送量} &= 27648 \text{ バイト} \end{aligned}$$

となる。下付き添え字の $\pm$ はそれぞれ前方、後方への転送を意味する。X 方向に関しては、隣接通信 1 回あたり前方か後方のどちらか片方のみしか送信しない。その他の方向に関して、前方、後方の両方への送信が必要である。

上述の「プロセス配置と TNI 割り当ての最適化」プログラムを用いると、物理 6 次元形状で $24 \times 22 \times 24 \times 2 \times 3 \times 2$ ノードを使用して、表 4.116 のように TY 軸と TZ 軸をそれぞれ 2 つの軸に仮想的に分割して使用するプロセスマッピングが最適だとわかる。表 4.116 の TYc/TYd、TZc/TZd は Y、Z 軸をそれぞれ分割したときの仮想軸の名前である。

表 4.116: LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TYc,TZc (2×3)	1	0	4
QY	32	TYd,TB (11×3)	1	0	4
QZ	32	TZd (8)	4	3	1
QT	96	TA,TC,TX ($2 \times 2 \times 24$)	1	0	4

TY 軸は長さ 2 の TYc 軸と長さ 11 の TYd 軸に分割し、TY の座標 y が、それぞれ、$y \bmod 2$ と $\left[\frac{y}{2}\right]$ に対応するものとする。したがって、TYc の座標上で隣接のものは TY 上でも隣接、TYd の座標上で隣接のものは TY 上では 2 ホップ離れることとなる。TZ 軸は長さ 3 の TZc 軸と長さ 8 の TZd 軸に分割し、TZ の座標 z が、それぞれ、$z \bmod 3$ と $\left[\frac{z}{3}\right]$ に対応するものとする。したがって、TZc の座標上で隣接のものは TZ 上でも隣接、TZd の座標上で隣接のものは TZ 上では 3 ホップ離れることとなる。また、TZ 軸はトーラスであるため、TZd 軸もトーラスとなる。そのため、TZd 軸全体で QZ 軸に割り当てることができる。このプロセス配置をした際のメッセージ送信先の QCD 座標上の方向と Tofu 座標上の方向の組み合わせに対する、TNI の割り当ておよび通信量を表 4.117 に示す。ノードあたりの TNI の個数は 6、物理 6 次元 Tofu 座標上の通信方向（以後こ

表 4.117: LQCD の TNI 割り当てと通信量 (プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$)

QCD 座標上の方向	Tofu 座標上の方向	TNI#	プロセスあたりメッセージ長 (Byte)	通信プロセス数	リンク多重度 (ホップ数)	リンク/ノード内への通信量 (Byte)
QX	TY±	0/1	3456	4	1	13824
	TZ±	2		4	1	13824
QY-	TB±	0	9216	4	1	36864
	TY±	0		4	2	73728
QY+	TB±	1	9216	4	1	36864
	TY±	1		4	2	73728
QZ-	TZ±	2	9216	1	3	27648
	LB	3		3	-	27648
QZ+	TZ±	2	9216	1	3	27648
	LB	3		3	-	27648
QT-	TA	4	27648	4	1	110592
	TC	4		4	1	110592
	TX±	4		4	1	110592
QT+	TA	5	27648	4	1	110592
	TC	5		4	1	110592
	TX±	5		4	1	110592

の通信経路をリンクと呼ぶ）は 10 方向あり、TNI の割り当ては、任意のリンクについて対応する TNI が丁度 1 つとなるように行う。この制約は、複数の TNI が 1 つのリンクを同時に使用することによる TNI-リンク間の競合を避けるために設けるものである。なお、リンクと TNI の割り当ては送信と受信で同じとする。こうすることで送受信が対称的となるため、また、リンクおよび TNI は送信方向と受信方向のバンド幅が同等であるため、以降の議論は送受信双方に適用される。Tofu 座標上の方向の LB は自ノードへのループバックであることを示す。

リンク/ノード内への通信量はノード内の全プロセス分を合計し、また複数ホップによる他ノードの通信を転送する分も含めたものである。これは、プロセスあたりメッセージ長、通信プロセス数、およびリンク多重度を掛けて求めることができる。通信プロセス数は表 4.117 の通りである。リンク多重度は、TYd を通る通信は 2、TZd を通る通信は 3 となる。

TNI の割り当てが 0/1 となっている行について、QX 方向の通信は TY-か TY+の両方を同時に使うことがないため、QY-と QY+のうち同じリンクを使う方に割り当てた TNI を選択することを意味する。例えば、QX 方向の通信と QY-方向の通信の両方が TY-を使用する場合、TNI#0

を使用する。

表 4.117 を基に、リンクあたり、および TNI あたりの通信量の合計を計算する。全ての QCD 座標の軸はトーラス状にノードを割り当てているため、QCD 座標上の同じ軸の正方向と負方向の通信が同じリンクを使うことはない。そのため、リンクあたりの通信量を合計する際はどちらか片方の値のみ加算すればよい。また、QCD 座標上のある方向の通信が同時に複数のリンクを使用することもない。そのため、TNI あたりの通信量を合計する際は、それらの中でリンクへの通信量が最大となるもののみを加算すればよい。以上のように計算したリンクあたり、および TNI あたりの通信量の合計をそれぞれ表 4.118、表 4.119 に示す。

表 4.118: LQCD の隣接通信のリンクあたり通信量 (プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$)

Tofu 座標上の方向	QCD 座標上の方向	TNI#	リンクへの送出量 (Byte)	合計対象	合計 (Byte)
TA	QT-	4	110592	○	110592
	QT+	5	110592		
TB-	QY-	0	36864	○	36864
	QY+	1	36864		
TB+	QY-	0	36864	○	36864
	QY+	1	36864		
TC	QT-	4	110592	○	110592
	QT+	5	110592		
TX-	QT-	4	110592	○	110592
	QT+	5	110592		
TX+	QT-	4	110592	○	110592
	QT+	5	110592		
TY-	QX	0/1	13824	○	87552
	QY-	0	73728	○	
	QY+	1	73728		
TY+	QX	0/1	13824	○	87552
	QY-	0	73728	○	
	QY+	1	73728		
TZ-	QX	2	13824	○	41472
	QZ-	2	27648	○	
	QZ+	2	27684		
TZ+	QX	2	13824	○	41472
	QZ-	2	27648	○	
	QZ+	2	27684		
LB	QZ-	3	27648	-	-
	QZ+	3	27648	-	-

表 4.118、表 4.119 より、ボトルネックは TA、TC、TX+、TX-、TNI#4、5 であり、その通信量は 110592 バイトとなり、110592B/(6.12GB/s)=0.01807 ミリ秒が隣接通信受信待ち 1 回あたりの時間となる。QCDJDD 解法 1 反復には 20 回の隣接通信があるため、評価区間全体の隣接通信受信待ち時間は 0.36[ミリ秒] と推定される。

4.10.2.18.2　隣接通信性能推定の検証

隣接通信の通信完了を待つ `comlib_recv_wait_all_c` 区間について、第 4.10.2.18 で行った机上推定手法の実測による検証を行った。実測は、富岳で実行する際の通信の状況を再現するように、京の一部のノードに収まる規模に縮小して行った。実測の対象とする富岳のプロセス配置と TNI 割り当ては第 4.10.2.18 で推定に用いたものとは異なるが、割り当てやメッセージ長計算の手順は同じなので、その手順の正しさに対する検証となる。また、推定に用いた割り当てと異なり、本項で使用するものはノード間通信に 4 つの TNI しか用いないため、ノード内通信を省略することで TNI 数が 4 である京でもボトルネックを再現できるため実行時間の検証も可能である。

実測の対象とする富岳の実行条件は第 4.10.2.18 と同じとする。プロセス配置を表 4.120、TNI 割り当てと通信量を表 4.121 に示す。このときのリンクあたり、および TNI あたりの通信量の合

表 4.119: LQCD の隣接通信の TNI あたり通信量 (プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$)

TNI#	QCD 座標上の方向	Tofu 座標上の方向	リンクへの送出量 (Byte)	合計対象	合計 (Byte)
0	QX	TY-	13824	○	87552
		TY+	13824		
	QY-	TB-	36864		
		TB+	36864		
		TY-	73728	○	
		TY+	73728		
1	QX	TY-	13824	○	87552
		TY+	13824		
	QY+	TB-	36864		
		TB+	36864		
		TY-	73728	○	
		TY+	73728		
2	QX	TZ-	13824	○	69120
		TZ+	13824		
	QZ-	TZ-	27648	○	
		TZ+	27648		
	QZ+	TZ-	27648	○	
		TZ+	27648		
3	QZ-	LB	27648	○	55296
	QZ+	LB	27648	○	
4	QT-	TA	110592	○	110592
		TC	110592		
		TX-	110592		
		TX+	110592		
5	QT+	TA	110592	○	110592
		TC	110592		
		TX-	110592		
		TX+	110592		

計はそれぞれ表 4.122、表 4.123 の通りである。以上の表より、ボトルネックとなる箇所の通信量は 124416 バイトとなる。表記の意味や計算方法は第 4.10.2.18 と同じである。

表 4.120: LQCD の軸と Tofu の軸の対応 (検証用、プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TYc,TZc (2×3)	1	0	4
QY	32	TYd,TB (11×3)	1	0	4
QZ	32	TZd,TA,TC ($8 \times 2 \times 2$)	1	0	4
QT	96	TX (24)	4	3	1

以上の通信の状況を再現して規模を縮小した京の実行条件を述べる。プロセス分割数は $6 \times 6 \times 12 \times 3$、使用ノード数は $2 \times 6 \times 9 \times 2 \times 3 \times 2 = 1296$ ノードである。プロセス配置は表 4.124 の通りである。ノード内プロセス数を 4 にする代わりに 1 にして、元のプロセス配置でノード間通信数が 4 のものはメッセージ長を 4 倍している。ノード内プロセス数を 1 にするのは、プロセスあたりに割り当てることができる TNI の資源数を十分大きくするためである。また、ノード内通信を省略することで、表 4.123 の TNI#4 および 5 の割り当てを不要とし、TNI の負荷も再現することができる。

通信の状況を再現するためには、分割せずに利用する Tofu の軸は経路重複がないため長さ 2 以上あれば十分である。分割して利用する場合は、連続の側 (TYc、TZc) は同じ長さ、複数ホップする側 (TYd、TZd) は 3 以上にする必要がある。そうすることで複数ホップする通信による重複数が最大、かつ連続の側の通信も重複する経路を再現できる。表 4.124 は上記を満たす中で最小の規模である。通信の状況を再現しているため、リンクと TNI あたりの通信量は表 4.122、表 4.123 と同じとなる。ただし、上記の通り TNI#4 と 5 は使用しない。また、TX の一部を使うように縮

表 4.121: LQCD の TNI 割り当てと通信量 (検証用、プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$)

QCD 座標上の方向	Tofu 座標上の方向	TNI#	プロセスあたりメッセージ長 (Byte)	通信プロセス数	リンク多重度 (ホップ数)	リンク/ノード内への通信量 (Byte)
QX	TY±	0/1	3456	4	1	13824
	TZ±	2/3		4	1	13824
QY-	TB±	0	9216	4	1	36864
	TY±	0		4	2	73728
QY+	TB±	1	9216	4	1	36864
	TY±	1		4	2	73728
QZ-	TA	2	9216	4	1	36864
	TC	2		4	1	36864
	TZ±	2		4	3	110592
QZ+	TA	3	9216	4	1	36864
	TC	3		4	1	36864
	TZ±	3		4	3	110592
QZ-	TX±	0	27648	1	1	27648
	LB	4		3	-	82944
QZ+	TX±	1	27648	1	1	27648
	LB	5		3	-	82944

表 4.122: LQCD の隣接通信のリンクあたり通信量 (検証用、プロセスあたり格子サイズ 32×6×6×2)

Tofu 座標上の方向	QCD 座標上の方向	TNI#	リンクへの送出量 (Byte)	合計対象	合計 (Byte)
TA	QZ-	2	36864	○	36864
	QZ+	3	36864		
TB-	QY-	0	36864	○	36864
	QY+	1	36864		
TB+	QZ-	2	36864	○	36864
	QZ+	3	36864		
TC	QY-	0	36864	○	36864
	QY+	1	36864		
TX-	QT-	0	27648	○	27648
	QT+	1	27648		
TX+	QT-	0	27648	○	27648
	QT+	1	27648		
TY-	QX	0/1	13824	○	87552
	QY-	0	73728	○	
	QY+	1	73728		
TY+	QX	0/1	13824	○	87552
	QY-	0	73728	○	
	QY+	1	73728		
TZ-	QX	2/3	13824	○	124416
	QZ-	2	110592	○	
	QZ+	3	110592		
TZ+	QX	2/3	13824	○	124416
	QZ-	2	110592	○	
	QZ+	3	110592		
LB	QT-	4	82944	-	-
	QT+	5	82944	-	-

小したことでトーラスと見なせなくなるため、QT への割り当ては TX から TB に入れ替えている。そのため、TX と TB は読み替える必要がある。

通信量や時間の実測には、隣接通信のみを実行するプログラムを用いた。

実測により得られたリンク毎の通信量を表 4.125 に示す。机上値に対する割合は+3%前後であり、これはパケットのヘッダと受信完了を通知するパケットによる増加分である。したがって、机上計算の結果は実測と一致しているといえる。

実測により得られた隣接通信の実行時間を表 4.126 に示す。比較する区間は受信完了待ちである。これは隣接通信全体から、送信の開始処理の実行時間を引いて求めている。受信完了待ち時間の机上値は、ボトルネックになる個所の転送量 124416 バイト、京の Tofu の実効通信バンド幅

表 4.123: LQCD の隣接通信の TNI あたり通信量 (検証用、プロセスあたり格子サイズ 32×6×6×2)

TNI#	QCD 座標上の方向	Tofu 座標上の方向	リンクへの送出量 (Byte)	合計対象	合計 (Byte)
0	QX	TY-	13824	○	115200
		TY+	13824		
	QY-	TB-	36864		
		TB+	36864		
		TY-	73728	○	
		TY+	73728		
	QT-	TX-	27648	○	
		TX+	27648		
1	QX	TY-	13824	○	115200
		TY+	13824		
	QY+	TB-	36864		
		TB+	36864		
		TY-	73728	○	
		TY+	73728		
	QT+	TX-	27648	○	
		TX+	27648		
2	QX	TZ-	13824	○	124416
		TZ+	13824		
	QZ-	TA	36864		
		TC	36864		
		TZ-	110592	○	
		TZ+	110592		
3	QX	TZ-	13824	○	124416
		TZ+	13824		
	QZ+	TA	36864		
		TC	36864		
		TZ-	110592	○	
		TZ+	110592		
4	QT-	LB	82944	○	82944
5	QT+	LB	82944	○	82944

表 4.124: LQCD の軸と Tofu の軸の対応 (検証用規模縮小版、プロセスあたり格子サイズ 32 × 6 × 6 × 2)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	プロセスあたりメッセージ長
QX	6	TYc,TZc (2x3)	1	4 倍
QY	6	TYd,TX (2x3)	1	4 倍
QZ	12	TZd,TA,TC (3x2x2)	1	4 倍
QT	3	TB (3)	1	1 倍

表 4.125: LQCD の隣接通信の通信量の実測値（プロセスあたり格子サイズ 32 × 6 × 6 × 2）

リンク	机上値 (Byte)	実測値 (Byte)	机上値に対する割合
A	36864	37832	+2.6%
B+	27648	28512	+3.1%
B-	27648	28512	+3.1%
C	36864	37864	+2.7%
X+	36864	37832	+2.6%
X-	36864	37832	+2.6%
Y+	87552	89896	+2.7%
Y-	87552	89896	+2.7%
Z+	124416	127792	+2.7%
Z-	124416	127792	+2.7%

の想定値 4.5[GB/s] より、27.65[マイクロ秒] と計算した。

京での実行時間の実測値は机上値に対して 11.5%大きくなっている。この差の原因として考えられるのは、複数ホップで発生する経路競合による効率の低下、および複数方向から同時に同じ TNI で受信したときの受信 TNI での律速である。これらの影響はタイミングにもよるため、詳細

表 4.126: LQCD の隣接通信の実行時間の実測値（プロセスあたり格子サイズ $32 \times 6 \times 6 \times 2$）

区間	机上値 (マイクロ秒)	実測値 (マイクロ秒)	机上値に対する割合
受信完了待ち	27.65	30.83	+11.5%
送信開始処理	-	3.71	-
全体	-	34.54	-

に解析したり推定に含めることは難しい。受信完了待ち時間が現在の推定より 11.5%大きくなったとしても、アプリ全体の推定時間は 3.6%の増加であり影響は小さいため、推定方法は従来通りの机上計算とし、富岳実機が利用できるようになりしだい再度確認することとする。

4.10.2.18.3　プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$ の隣接通信時間

プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$ の場合も、4.10.2.18.1 と同様に、表 4.127 のプロセスマッピングが最適だとわかる。このとき、TNI の割り当ておよび通信量は表 4.128 となる。表 4.129 及び表 4.130 からボトルネックは TB-、TB+、TY-、TY+、TNI#2、3、4、5 であり、その通信量は 73,728 バイトとなり、73728B/(6.12GB/s)=0.01205 ミリ秒が隣接通信受信待ち 1 回あたりの時間となることがわかる。QCDJDD 解法 1 反復には 20 回の隣接通信があるため、評価区間全体の隣接通信受信待ち時間は 0.24[ミリ秒] と推定される。

表 4.127: LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TA,TZc (2×3)	1	0	4
QY	32	TZd,TC (8×2)	2	2	2
QZ	48	TX (24)	2	2	2
QT	64	TY,TB (22×3)	1	0	4

表 4.128: LQCD の TNI 割り当てと通信量 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$)

QCD 座標上の方向	Tofu 座標上の方向	TNI#	プロセスあたりメッセージ長 (Byte)	通信プロセス数	リンク多重度 (ホップ数)	リンク/ノード内への通信量 (Byte)
QX	TA	0	3456	4	1	13824
	TZ±	0/1		4	1	13824
QY−	TC	0	9216	2	1	18432
	TZ+	0		2	3	55296
	LB	3		2	-	18432
QY+	TC	1	9216	2	1	18432
	TZ−	1		2	3	55296
	LB	2		2	-	18432
QZ−	TX	3	13824	2	1	27648
	LB	3		2	-	27648
QZ+	TX	2	13824	2	1	27648
	LB	2		2	-	27648
QT−	TB±	5	18432	4	1	73728
	TY±	5		4	1	73728
QT+	TB±	4	18432	4	1	73728
	TY±	4		4	1	73728

4.10.2.18.4　プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$ の隣接通信時間検証方法

ソフトウエア調整（1）において、プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$ の隣接通信の通信完了を待つ `comlib_recv_wait_all_c` 区間について経過時間の検証方法を検討した。富岳少ノードでも、4.10.2.18.2 と同様に、実行する際の通信の状況を再現することができる。Tofu の軸

表 4.129: LQCD の隣接通信のリンクあたり通信量 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$)

Tofu 座標上の方向	QCD 座標上の方向	TNI#	リンクへの送出量 (Byte)	合計対象	合計 (Byte)
TA	QX	0	13842	○	13842
TB−	QT∓	5/4	73728	○	73728
TB+	QT±	4/5	73728	○	73728
TC	QY∓	0/1	18432	○	18432
TX−	QZ−	3	27648	○	27648
TX+	QZ+	2	27648	○	27648
TY−	QT∓	5/4	73728	○	73728
TY+	QT±	4/5	73728	○	73728
TZ−	QX	1	13824	○	69120
	QY+	1	55296	○	
TZ+	QX	0	13824	○	69120
	QY−	0	55296	○	
LB	QY−	3	18432	-	-
	QZ−	3	27648	-	-
	QY+	2	18432	-	-
	QZ+	2	27648	-	-

表 4.130: LQCD の隣接通信の TNI あたり通信量 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$)

TNI#	QCD 座標上の方向	Tofu 座標上の方向	リンクへの送出量 (Byte)	合計対象	合計 (Byte)
0	QX	TA	13824		69120
	QX+	TZ±	13824	○	
	QY−	TC	18432		
		TZ+	55296	○	
1	QX−	TZ∓	13824	○	69120
	QY+	TC	18432		
		TZ−	55296	○	
2	QZ+	TX+	27648	○	73728
		LB	27648	○	
	QY+	LB	18432	○	
3	QZ−	TX−	27648	○	73728
		LB	27648	○	
	QY−	LB	18432	○	
4	QT+	TB±	73728	○	73728
		TY±	73728		
5	QT−	TB∓	73728	○	73728
		TY∓	73728		

を変更しない場合は、表 4.131 の様に、TY を縮小して $(\mathrm{TX} \times \mathrm{TA}) \times (\mathrm{TY} \times \mathrm{TB}) \times (\mathrm{TZ} \times \mathrm{TC}) = (24 \times 2) \times (2 \times 3) \times (24 \times 2)$ の 36 ラックで、実際の全くおなじリンクと TNI の割付で通信状況が再現可能である。36 ラック未満の場合、Tofu の X 軸、Z 軸のトーラス性が失われる。その場合は、Tofu の X 軸と B 軸を交換し、Tofu の X 軸と Y 軸でトーラスを構成する。例えば $(\mathrm{TX} \times \mathrm{TA}) \times (\mathrm{TY} \times \mathrm{TB}) \times (\mathrm{TZ} \times \mathrm{TC}) = (2 \times 2) \times (2 \times 3) \times (24 \times 2)$ の 3 ラックの場合は、LQCD の軸を表 4.132 にあるように Tofu の軸に対応させることで、通信状況が再現可能である。この時、Tofu の X 軸と B 軸に関するリンクと TNI 割付が入れ替わる点に注意する必要がある。QY 軸は TZd 軸と TC 軸の二つで構成されるため、TZd 軸はトーラスでなくても良い。そのため、Tofu の Z 軸がトーラスになっていなくても通信状況が再現可能であり、その最小構成は表 4.133 にあるように $(\mathrm{TX} \times \mathrm{TA}) \times (\mathrm{TY} \times \mathrm{TB}) \times (\mathrm{TZ} \times \mathrm{TC}) = (2 \times 2) \times (2 \times 3) \times (9 \times 2)$ の 432 ノードである。

4.10.2.19　1MPI プロセスで 4CMG を使う実行方法

TNI での律速をさけるために 1MPI プロセスで 4CMG を使う実行方法について検討した。192^4 を 147456 プロセスで並列化する場合、通信に係る表面積が最も小さくなるのは $n_x \times n_y \times n_z \times n_t =$

表 4.131: LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$、隣接通信 36 ラック検証用)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TA,TZc (2×3)	1	0	4
QY	32	TZd,TC (8×2)	2	2	2
QZ	48	TX (24)	2	2	2
QT	6	TY,TB (2×3)	1	0	4

表 4.132: LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$、隣接通信 3 ラック検証用)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TA,TZc (2×3)	1	0	4
QY	32	TZd,TC (8×2)	2	2	2
QZ	6	TB (3)	2	2	2
QT	4	TY,TX (2×2)	1	0	4

表 4.133: LQCD の軸と Tofu の軸の対応 (プロセスあたり格子サイズ $32 \times 6 \times 4 \times 3$、隣接通信 432 ノード検証用)

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TA,TZc (2×3)	1	0	4
QY	12	TZd,TC (3×2)	2	2	2
QZ	6	TB (3)	2	2	2
QT	4	TY,TX (2×2)	1	0	4

$32 \times 6 \times 6 \times 8$ でプロセス分割を $6 \times 32 \times 32 \times 24$ とした場合である。この時、各方向の転送量は

$$\begin{aligned} \mathrm{QX}_{\pm} \text{ 方向転送量} &= \quad 13824 \text{ バイト} \\ \mathrm{QY}_{\pm} \text{ 方向転送量} &= \quad 36864 \text{ バイト} \\ \mathrm{QZ}_{\pm} \text{ 方向転送量} &= \quad 36864 \text{ バイト} \\ \mathrm{QT}_{\pm} \text{ 方向転送量} &= \quad 27648 \text{ バイト} \end{aligned}$$

となる。

効率的なプロセスマッピングでの議論と同様に TX 軸と TY 軸をそれぞれ 2 つの軸に仮想的に分割して使用する。TX 軸は長さ 4 の TXc 軸と長さ 6 の TXd 軸に分割し、TX の座標 x が、それぞれ、$x \bmod 4$ と $\left[\frac{x}{4}\right]$ に対応するものとする。TY 軸は長さ 2 の TYc 軸と長さ 12 の TYd 軸に分割し、TY の座標 z が、それぞれ、$z \bmod 2$ と $\left[\frac{z}{2}\right]$ に対応するものとする。QCD の軸と Tofu の軸を表 4.134 に示す。

表 4.134: LQCD の軸と Tofu の軸の対応（1MPI プロセスで 4CMG）

QCD の軸	プロセス数	Tofu の軸	ノード内プロセス数	ノード内への通信プロセス数	他ノードへの通信プロセス数
QX	6	TXd	1	0	1
QY	32	TB×TYd	1	0	1
QZ	32	TA×TC×TXc×TYc	1	0	1
QT	24	TZ	1	0	1

この時の各リンクの転送量を表 4.135 に示す。仮に、表 4.136 のようにリンクと TNI を割り当てると、TNI#2,3,5 がボトルネックとなり、隣接通信受信待ち 1 回あたり 110592B/(6.12GB/s)=0.01807 ミリ秒かかる。これは 4.10.2.18.3）で議論した経過時間より長い。

表 4.135: LQCD の隣接通信区間における各リンクの転送量（1MPI プロセスで 4CMG）

リンク	転送量	転送量の計算方法
$TX_{\pm}$	92160 バイト	$QX_{\pm}$ 方向転送量 × 4 ホップ + $QZ_{\pm}$ 方向転送量
$TY_{\pm}$	110592 バイト	$QZ_{\pm}$ 方向転送量 + $QY_{\pm}$ 方向転送量 × 2 ホップ
$TZ_{\pm}$	27648 バイト	$QT_{\pm}$ 方向転送量
TA	36864 バイト	$QZ_{\pm}$ 方向転送量
$TB_{\pm}$	36864 バイト	$QY_{\pm}$ 方向転送量
TC	36864 バイト	$QZ_{\pm}$ 方向転送量

表 4.136: LQCD の隣接通信区間における各 TNI の転送量

TNI 番号	使用リンク	転送量	転送量の計算方法
0	TX_{+}	92160 バイト	TX
1	TX_{-}	92160 バイト	TX
2	TY_{+}	110592 バイト	TY
3	TY_{-}	110592 バイト	TY
4	TZ_{+}、TZ_{-}、TA	92160 バイト	TZ_{+} + TZ_{-} + TA
5	TB_{+}、TB_{-}、TC	110592 バイト	TB_{+} + TB_{-} + TC

4.10.3　5 次元型フェルミオン行列の逆行列反復解法コード

LQCD では上記のウィルソン型の他にも様々なフェルミオン行列が用いられる。一例として、4 次元格子に更に 1 次元を加えることにより軽いクォークが持つ対称性を改良した、5 次元型フェルミオン行列を検討する。時空に対応する 4 次元部分の構造はウィルソン型行列と同じであるので、最適化中の LQCD カーネルライブラリを利用することにより、かなりの高速化が期待できる。一方でキャッシュ利用の効率化の点からは、5 次元目方向の格子の自由度 (10 から 100 程度) を 4 次元格子点上の自由度として扱う方法が有効である。これら 2 種類の実装を比較し、LQCD カーネルライブラリでの最適化手法の一般化を図る。

第 1 のアプローチでは、公開されている 5 次元型フェルミオン行列を含むコードを変更して、その中の 4 次元カーネル部分として LQCD ライブラリを呼ぶコードを実装した。第 2 のアプローチでは、LQCD ライブラリの実装を参考に、5 次元型フェルミオン行列を直接実装した。予備的な性能比較では第 2 の方法のコードの方が高い実行性能を示したが、これは第 1 の方法では線形代数演算などが十分最適化されていないことや、LQCD ライブラリが高性能を発揮するパラメターから外れていたことが原因と考えられる。どちらの実装においても、更なる最適化と高い実行性能が得られる条件を引き続き検討している。

4.10.4　核力計算コードの改善検討

192^4 の格子では、クォークのプロパゲータ一つあたりのメモリ容量は、$16(\mathrm{Byte}) \times 3 \times 4 \times 3 \times 4 \times 192^3 = 16.3\mathrm{GByte}$ となる。フレーバ 2 + 1 セクタで、負パリティ核力・ハイペロン力を含めて、スピン軌道力（LS 力）もしくは反対称 LS 力まで結合チャンネルで求める場合、このクォークプロパゲータをアップとダウン クォークについて（クォークの運動量の方向に対応して）7 本、ストレンジクォークについて 7 本持つ必要がある。これらを合計すると、228 GByte となる。これは富岳の 1 ノードあたりのメモリ容量を超えるため、クォークのプロパゲータは、空間的に分割して持つ事が必須となる。

一方、バリオンの 4 点関数を効率よく計算するには、3 次元 FFT を利用する。現在のコードでは、二つのバリオンの運動する方向（7 つ）と二つのハイペロンの組み合わせの数 115 チャンネルのバリオン 4 点関数の Wick contraction を (半) 自動生成し、計算を行うことになるが、8280 万回の 3 次元 FFT が必要になってしまう。クォークのプロパゲータが空間的に分割されている状況か

ら、3 次元 FFT も MPI 通信を伴ったものとなり、これを実行するのは困難である。そのため、アルゴリズムやコーディングが (負パリティ核力・ハイペロン力計算のために使用する momentum wall source 用には) 非常に複雑となるが、バリオン 4 点関数をパーツに分け、これを 3 次元 FFT まで終わらせた物を作り置きする方法を開発して用いる事を検討した。現在はより良い方法を検討している。

4.10.5 コデザインによるシステムへの実装要請項目

本コデザインを通してシステムへフィードバックした実装要請・改善項目を以下に示す。

1. 1〜3 要素 MPI_Allreduce

   LQCD でもっとも時間がかかる部分は、反復法によるクォーク伝搬関数の求解である。ターゲット問題では 1 反復あたりの実行時間が 1 ミリ秒程度と非常に短いため、MPI_Allreduce の実行時間が無視できないものとなっているため、富岳に 3 要素までの MPI_Allreduce の高速処理機構を持たせることになった。

   富岳における 1〜3 要素の MPI_Allreduce は、京の 1 要素 MPI_Allreduce と同様に、TBI(Tofu Barrier Interface) と呼ばれる専用のハードウェア資源を使用しツリー構造でリダクション処理およびブロードキャスト処理を行うことで実現された。

2. OS ジッタ 0

   ターゲット問題では 1 反復あたりの実行時間が 1 ミリ秒程度と非常に短いため、OS ジッタの影響も無視できない。そのため、富岳では、アシスタントコアを備え、計算コアで OS ジッタが発生しないようにすることになった。

3. RDMA 通信機構

   クォーク伝搬関数の求解においては、疎行列ベクトル積が演算の主要部分である。ターゲット問題では反復法で利用されるすべてのデータが L2 キャッシュに載るぐらいプロセスあたり問題規模が小さいため、疎行列ベクトル積に必要なプロセス外データの転送に必要な時間とプロセス内で閉じた疎行列ベクトル積演算の実行時間が同程度になることを確認した。隣接通信と演算を重ねるための DMA 通信機構が LQCD でも必要であると認識され、これを京に引き続き富岳でも備えることになった。

4. 複数の RDMA 通信の発行時間短縮

   LQCD は 4 次元ステンシル計算で、一度の袖領域の通信で 8 個の RDMA 通信を発行する必要がある。京では RDMA 通信の発行にかかるオーバーヘッドが大きいため、複数の RDMA 通信の発行するための API とその時間短縮効果を検討した。**uTofu** を使うことにより複数の RDMA 通信の発行が可能となることとなった。

5. 通信時間が他のジョブの影響を受けて大幅に増大する問題の対策

   京では通信時間が他のジョブの影響を受けて大幅に増大することがある。この現象の原因は通信経路を共有する「他のジョブの I/O 通信」による所が大きい。LQCD は隣接通信時間が比較的大きいため、富岳で対策するよう求めた。富岳では、SIO と GIO 共に対称性を持つ (X, Y, Z)=(2, 2, 8) を意識し、ラックの起点座標にジョブの割り当てを行い、ノード形状内の I/O ノード (SIO,GIO) の配置を固定させることでこの問題を解決できる。具体的には、

以下のようにジョブ投入コマンドの ”node ”引数に「4lx6mx16n:strict-io」を指定する。

pjsub -L node=4lx6mx16n:strict-io job.sh ($l, m, n = 1, 2, 3\cdots$)

6. その他コンパイラの最適化の改善に関する事項、ライブラリに関する事項を含み、システムへの実装要請の一覧を表 4.137 にまとめる

表 4.137: アプリケーション検討会で議論されたシステムへの実装要請一覧表 (LQCD)

実装要請課題	対応
SIMD 幅に等しい大きさの配列を返す関数をインライン展開する場合、レジスタ経由で値が返却されること	関数インライン展開時の値返却関連命令列の削減機能として開発する
Tree Height Reduction 最適化の強化	Tree Height Reduction 最適化のアルゴリズム選択機能として開発する
複数のループに渡って使用される一時配列の除去	Tree Height Reduction 最適化を適用後、既存の最適化機能で実現
SIMD 化時のアドレッシングモード活用による整数レジスタ使用数の削減	富岳向けアドレッシングモード選択機能として開発する
OS ジッタによりアプリケーションの性能低下が起きない OS の提供	ホスト Linux では OS ノイズの推定目標値を平均ノイズ率 1x10-5 以下、最大ノイズ長 10 μ秒以下に設定済み。McKernel では OS ノイズ 0 となる予定
RDMA 通信機構の提供	隣接通信と演算を重ねるための RDMA 通信機構が LQCD でも必要であると認識され、これを京に引き続き富岳でも備えることになった
複数の RDMA 通信の発行時間短縮	uTofu インターフェースで対応する。スレッドセーフを保証する
通信時間が他のジョブのファイル I/O の影響を受けて大幅に増大する問題の対策	I/O ノードを占有利用するように各ジョブにノード群を割り当て、通信時に他ジョブの I/O による影響を受けないようにする
1〜3 要素 MPI_Allreduce の高速な実装	3 要素までの MPI_Allreduce を Tofu ハードウエアで処理する高速処理機構を実装する。
複素数の実部、虚部がそれぞれ SIMD 要素数分連続して交互に格納された配列を引数とする GEMM ライブラリ	SIMD 要素数単位で実部と虚部が分離された配列に対応する行列行列積機能を開発する
行列転置機能	BLAS 拡張の行列転置機能を開発する
FFT のバッチモード対応	FFTW での対応を検討する

4.10.6 京でのベースライン測定

ターゲットの格子点数と同じ 192^4 格子を京 82944 ノードで OpenMP 8 スレッドを使用し実行した。京の 6 次元物理ノード形状は以下に示す通りであり、プロセス間の通信遅延やスループットが良くなるよう、以下のように QCDJDD 解法のプロセス分割を行う。

$$\mathrm{TX} \times \mathrm{TY} \times \mathrm{TZ} \times \mathrm{TA} \times \mathrm{TB} \times \mathrm{TC} = 24 \times 18 \times 16 \times 2 \times 3 \times 2$$
$$\mathrm{QX} \times \mathrm{QY} \times \mathrm{QZ} \times \mathrm{QT} = 24 \times 24 \times 24 \times 6$$

このとき、ノードあたりの格子サイズは $8 \times 8 \times 8 \times 32$ である。TY 軸と TZ 軸を (TY = 18 = TYd×TYc、TYd=6、TYc=3)、(TZ = 16 = TZd×TZc、TZd=8、TZc=2) と分割し、

$$\begin{aligned}
&\mathrm{QX} = \mathrm{TX} = 24 \\
&\mathrm{QY} = \mathrm{TYc} \times \mathrm{TZd} = 3 \times 8 = 24 \\
&\mathrm{QZ} = \mathrm{TZc} \times \mathrm{TA} \times \mathrm{TB} \times \mathrm{TC} = 2 \times 2 \times 3 \times 2 = 24 \\
&\mathrm{QT} = \mathrm{TYd} = 6
\end{aligned}$$

とした。QX 方向は Tofu 上で TX 方向に隣接になり、QY 方向は、TY 方向隣接と TZ 方向に 2 ノード離れるものの混在となり、QZ 方向は、TZ 方向で隣接し、QT 方向は TY 方向に 3 ノード離れる様に配置した。

各区間の QCDJDD 解法 1 反復あたりの実測性能を表 4.138 に示す。表における演算効率は倍精度演算の理論性能を 1 として算出している。演算と通信のオーバーラップする演算区間 `ddd_in_s_` と通信区間 `comlib_recv_wait_all_c` については、富岳の推定では個々に推定した後に実行時間が短い方を合計から除くようにしているが、京の実測ではオーバラップさせたまま測定している。

表 4.138: LQCD の京でのベースライン測定結果

		実行時間（ミリ秒）	演算効率（%）
演算区間合計		27.99	34.8
内訳	`jinv_ddd_in_s_`	14.09	41.9
	`ddd_in_s_`	6.52	44.3
	`ddd_out_pre_s_`	0.95	12.6
	`ddd_out_pos_s_`	3.84	16.9
	`other_calc`	2.58	7.1
通信区間合計		2.66	
内訳	`comlib_irecv_all_c`	0.45	
	`comlib_isend_all_c`	0.17	
	`comlib_recv_wait_all_c`	0.16	
	`comlib_send_wait_all_c`	0.17	
	`s_drbicgstab_dd_hpc_iter_reduc1_`	0.18	
	`s_drbicgstab_dd_hpc_iter_reduc2_`	0.85	
	`s_drbicgstab_dd_hpc_iter_reduc3_`	0.67	
オーバーラップのため差し引く時間		-	
合計		30.65	31.8

4.10.7 富岳向け性能推定

富岳向け性能推定は第 4.1.5 節に説明される手法により実施した。ソフトウエア調整（1）までに実施された富岳向け性能推定を以下に示す。

4.10.7.1 富岳向け性能推定における通信性能の推定

通信性能については以下のとおり推定する。表 4.139 に LQCD の富岳における通信時間を示す。

- s_drbicgstab_dd_hpc_iter_reduc[123]_
  3 要素までのリダクションが 1 回実行できる高速版 MPI_Allreduce の全系利用時経過時間の推定値を用いる。

- comlib_irecv_all_c
  ダブルバッファリングにより処理が不要となるため 0 秒と推定する。

- comlib_isend_all_c
  通信のみを切り出したコードを用いて推定する。
  - 性能電力予測ツールによる推定では、京の実測値をそのまま富岳の推定値とする。
  - 試作機による推定では、**uTofu** を使い試作機での実測値を富岳の推定値とする。
- comlib_send_wait_all_c
  京の実測値をそのまま富岳の推定値とする。（通信のみを切り出したコードを用いて測定）
- comlib_recv_wait_all_c
  隣接通信で通信の完了待ち時間を最適化して通信時間を推定する。（参照：4.10.2.18）

表 4.139: LQCD の富岳における通信区間の推定実行時間。単位はミリ秒。

区間名	プロセスあたり格子サイズ							
	$32 \times 6 \times 6 \times 2$ (性能電力予測ツール推定用)				$32 \times 6 \times 4 \times 3$ (試作機推定用)			
	エコモード無効		エコモード有効		エコモード無効		エコモード有効	
	ノーマル	ブースト	ノーマル	ブースト	ノーマル	ブースト	ノーマル	ブースト
comlib_irecv_all_c	0	0	0	0	0	0	0	0
comlib_isend_all_c	0.13	0.13	0.13	0.13	0.081	0.073	0.082	0.074
comlib_recv_wait_all_c	0.36	0.36	0.36	0.36	0.241	0.241	0.241	0.241
comlib_send_wait_all_c	0.11	0.11	0.11	0.11	0.106	0.106	0.106	0.106
s_drbicgstab_dd_hpc_iter_reduc1_	0.02	0.02	0.02	0.02	0.020	0.020	0.020	0.020
s_drbicgstab_dd_hpc_iter_reduc2_	0.02	0.02	0.02	0.02	0.021	0.021	0.021	0.021
s_drbicgstab_dd_hpc_iter_reduc3_	0.04	0.04	0.04	0.04	0.043	0.043	0.043	0.043
合計	0.68	0.68	0.68	0.68	0.512	0.504	0.513	0.505

4.10.7.2　試作機による評価結果

表 4.140 に試作機 2 ノードを用いた性能測定結果を示す。測定に用いたソースコードは qws-0.2.2+patch_200107 である。プロセスあたり格子サイズは $32 \times 6 \times 4 \times 3$ である。演算効率は未計測で、消費電力は参考値である。

表 4.140: LQCD の富岳試作機を用いた性能測定結果。

モード	エコモード無効						エコモード有効					
	通常モード			ブーストモード			通常モード			ブーストモード		
区間	実行時間 [ms]	演算効率 [%]	消費電力 [W]	実行時間 [ms]	演算効率 [%]	消費電力 [W]	実行時間 [ms]	演算効率 [%]	消費電力 [W]	実行時間 [ms]	演算効率 [%]	消費電力 [W]
all_calc	0.452	-	-	0.412	-	-	0.509	-	-	0.465	-	-
overlapped	0.138	-	-	0.126	-	-	0.159	-	-	0.145	-	-
演算区間	0.590	-	-	0.538	-	-	0.668	-	-	0.610	-	-
通信区間	0.512	-	-	0.504	-	-	0.513	-	-	0.505	-	-
オーバーラップ区間	0.138	-	-	0.126	-	-	0.159	-	-	0.145	-	-
全体	0.964	-	117	0.916	-	138	1.022	-	85	0.970	-	99

4.10.8　富岳を用いた実測

富岳で実測された性能を以下に示す。利用したコードは QWS のバージョン 0.2.6（Fugaku benchmark version (Aug/03/2020)、https://github.com/RIKEN-LQCD/qws/tree/v0.2.6）である。ベンチマークテストはブーストモードかつエコモード無効で実施された。

2MPI プロセス（1 ノード 2CMG）で測定された LQCD の単体性能を表 4.141 と表 4.142 に示す。それぞれ、QCDJDD 解法 500 反復の性能と、QCDJDD 解法 500 反復中の ddd_in_s_ 区間の性能をプロセス当たり格子サイズを変えて測定した結果である。演算数は理論的に算出された値で、効率は単精度ピーク比に対するものである。この表から、通信時間を考慮しない場合は、ターゲット問題よりもプロセス当たりの格子サイズが大きい $32 \times 6 \times 4 \times 6$ が最も良い性能であることがわかる。

表 4.141: LQCD の単体性能（QCDJDD 解法 500 反復）。

サイズ	経過時間 [秒]	性能 [TFLOPS]	効率 [%]	演算数
$32 \times 6 \times 4 \times 3$	0.334867	0.8272	12.24%	69254421000
$32 \times 6 \times 4 \times 6$	0.515010	1.0839	16.04%	139560981000
$32 \times 6 \times 8 \times 6$	1.145754	0.9786	14.48%	280304661000
$32 \times 6 \times 8 \times 12$	2.606202	0.8616	12.75%	561369621000
$32 \times 12 \times 8 \times 12$	5.778703	0.7773	11.50%	1122981141000

表 4.142: LQCD の単体性能（QCDJDD 解法 500 反復中の ddd_in_s_ 区間）。

サイズ	経過時間 [秒]	性能 [TFLOPS]	効率 [%]	演算数
$32 \times 6 \times 4 \times 3$	0.068043	1.1208	16.58%	19065600000
$32 \times 6 \times 4 \times 6$	0.119455	1.3170	19.49%	39329280000
$32 \times 6 \times 8 \times 6$	0.219403	1.4693	21.74%	80593920000
$32 \times 6 \times 8 \times 12$	0.559297	1.1699	17.31%	163584000000
$32 \times 12 \times 8 \times 12$	1.192644	1.1146	16.49%	332328960000

LQCD では、Tofu ハードウェアによる 3 要素までの MPI_Allreduce の高速処理機構の有無が重要である。この効果を検証するため、“Intel(R) MPI Benchmarks 2019 Update 6, MPI 1 part (IMB–MPI1)” を用いた 72 ラック、27648 ノード（三次元形状指定：$48 \times 12 \times 48$）での Allreduce のベンチマークテスト結果を表 4.143 に示す。Tofu ハードウェアによる 3 要素までの MPI_Allreduce の高速処理機構の有無で、3 要素までの MPI_Allreduce の性能が約 6 倍ことなることがわかる。147456 ノードを利用するターゲット問題では、MPI_Allreduce の経過時間が全体経過時間の約 8% を占める。そのため、LQCD においては、Tofu ハードウェアによる 3 要素までの MPI_Allreduce の高速処理機構の恩恵が大きい。

147456 ノードまでの QCDJDD 解法 1 反復のウィークスケーリングを図 4.113 に示す。432 ノードからターゲットノードである 147456 ノードまで、一部 OS ジッタが原因と思われる例外を除き、良好なウィークスケーリングが見て取れる。432 ノードから 147456 ノードを増やすと、Allreduce の待ち時間のため、経過時間は 0.5 秒（約 7%）増加する。147456 ノードでのベンチマークテストは計 5 回実施され、経過時間はそれぞれ 0.8000 ミリ秒、0.7998 ミリ秒、0.7982 ミリ秒、0.7989 ミリ秒、0.7978 ミリ秒であった。結果として、京でのベースライン測定結果と比較して、38.3 倍の性能向上を達成した。平均電力は約 20 MW で、電力効率は 5 GFLOPS/W であった。

QCDJDD 解法 1 反復の経過時間の内訳を表 4.144 に示す。上記の 5 回の本番計測とは別に実施された計測結果である。時間計測処理のオーバーヘッドのため、合計の経過時間は本番計測の 0.8000 ミリ秒より若干長い。

最後に京に対する性能倍率及び消費電力を表 4.145 にまとめる。

表 4.143: IMB–MPI1 を用いた Allreduce ベンチマークテスト結果。$48 \times 12 \times 48$ ノードを用いて、Tofu ハードウェアによる 3 要素までの MPI_Allreduce の高速処理機構が「あり」、「なし」でそれぞれ測定。繰り返し回数は 10000 回で、最小 (min)、最大 (max)、平均 (avg) 時間をそれぞれ示す。byte は MPI_Allreduce1 回あたりのメッセージ長で、count は要素数、MPI_Allreduce の data type は MPI_FLOAT である。

		高速処理機構あり			高速処理機構なし		
byte	count	min [μs]	max [μs]	avg [μs]	min [μs]	max [μs]	avg [μs]
0	0	0.09	0.14	0.10	0.10	0.16	0.12
4	1	7.60	11.33	9.46	55.69	69.05	62.83
8	2	8.25	10.79	9.50	55.79	68.93	62.91
12	3	8.25	10.93	9.57	55.89	69.02	62.94
16	4	58.99	66.95	62.68	56.42	69.71	63.51
32	8	61.50	72.34	66.32	78.24	97.57	88.14
64	16	61.61	72.38	66.31	78.63	97.84	88.42
128	32	63.70	74.45	68.43	80.46	99.56	90.10

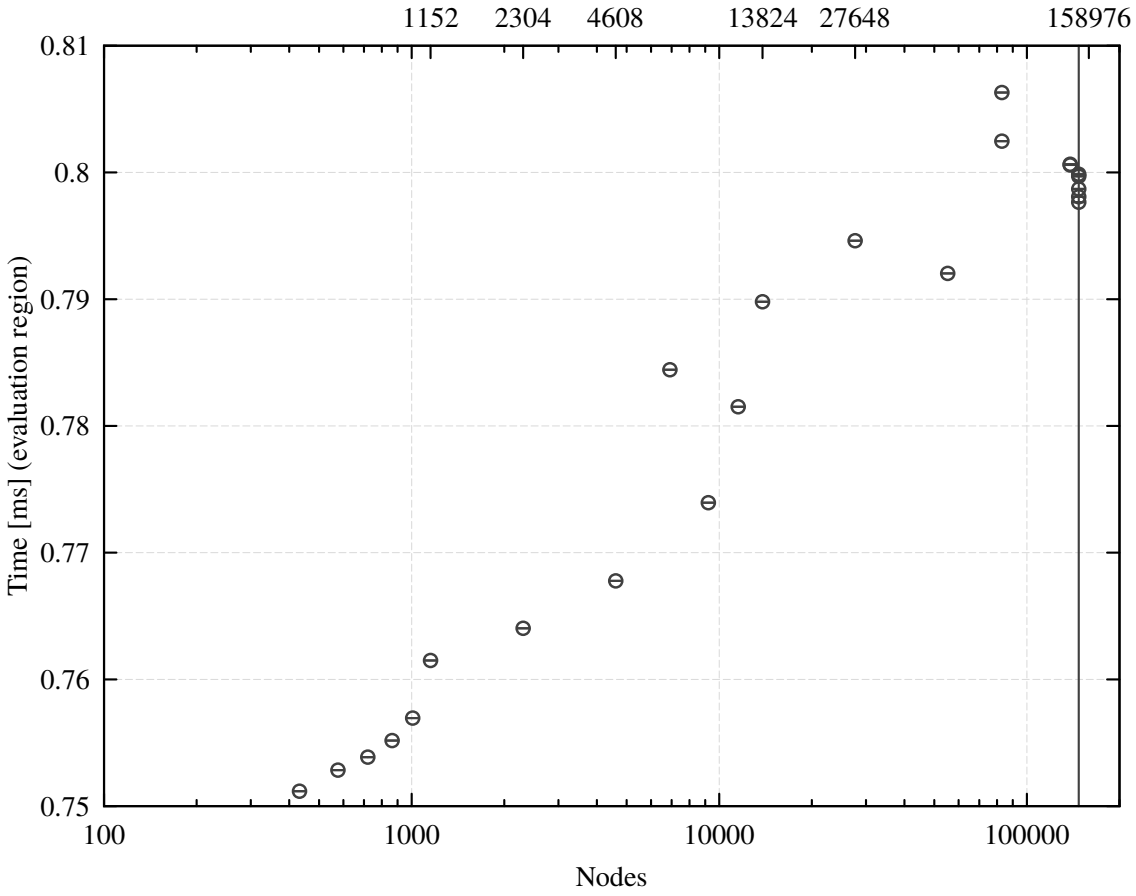


図 4.113: LQCD のウィークスケーリング（QCDJDD 解法 1 反復）。158976 ノードは富岳の全ノード数である。147456 ノードの縦線は、LQCD のターゲット問題を実行するノード数を示すためのものである。

表 4.144: LQCD の経過時間内訳（QCDJDD 解法 1 反復）。

区間	経過時間 [ミリ秒]
演算区間	0.400
通信区間	0.407
comlib_isend_all_c	0.029
comlib_recv_wait_all_c	0.254
comlib_send_wait_all_c	0.062
s_drbicgstab_dd_hpc_iter_reduc1_	0.015
s_drbicgstab_dd_hpc_iter_reduc2_	0.016
s_drbicgstab_dd_hpc_iter_reduc3_	0.031
合計	0.807

表 4.145: LQCD の対京性能倍率と消費電力

アプリケーション	性能倍率（注）	消費電力（注）
LQCD	38 倍	20 MW

（注）ターゲット問題実行時における対京性能倍率：ブーストモードかつエコ無効モードで実行

4.10.9 主筆者・執筆協力者

4.10.9.0.1 主筆

- 中村 宜文（理化学研究所計算科学研究センター）

4.10.9.0.2 執筆協力者 （５０音順）

- 石川 健一（広島大学先進理工系科学研究科）
- 金森 逸作（理化学研究所計算科学研究センター）
- 辻 美和子（理化学研究所計算科学研究センター）
- 似鳥 啓吾（理化学研究所計算科学研究センター）
- 松古 栄夫（高エネルギー加速器研究機構計算科学センター）

表 4.146: LQCD の電力制御各モードでの比較

アプリケーション	電力制御各モードにおける性能倍率と電力 上段:性能倍率、下段:システム消費電力平均値および最大値							
	エコモード　無効				エコモード　有効			
	ブーストモード		通常モード		ブーストモード		通常モード	
	性能倍率		性能倍率		性能倍率		性能倍率	
	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力	平均電力	最大電力
LQCD	38 倍		36 倍		36 倍		33 倍	
	21 MW	26 MW	19 MW	26 MW	16 MW	26 MW	15 MW	26 MW

第5章　新たなる付加価値：PCI を使った将来拡張の可能性

5.1　はじめに

本プロジェクトにおいて、半導体製造技術の変更により開発スケジュールに遅延が生じた際に、ポスト京の新たなる付加価値として、外部拡張デバイスの接続による「PCI を使った将来拡張の可能性」について調査検討を行うこととなった。本章では、この付加価値検討に関して行った取組みの概要を述べる[1]。

本取組では、ポスト京の新たなる付加価値検討の一環として、

- PCI による外部デバイスの拡張可能性
- 外部デバイス拡張のための技術的課題とその解決法

を明らかとすることを目的とし、ポスト京の拡張部構成方式、ポスト京による拡張部の利用方法、拡張部により得られる付加価値について調査を行った。特に、PCI により接続する拡張デバイスとして回路再構成可能半導体である FPGA (Field-Programmable Gate Array) を想定し、以下の各項目の研究開発や準備を実施した。

- 拡張方式
- 拡張部アーキテクチャと試作システム
- アプリケーション
- 共同研究に向けた準備

5.2　拡張方式

ポスト京への外部拡張デバイスの接続方式に関し、調査検討を行った。

図 5.1 に、ポスト京のラックと利用可能な PCI スロットを示す。接続にはポスト京ラック内とラック外の 2 通りが考えられるが、現行のポスト京シャーシでは PCI スロットの寸法および電力が不十分であり、またシャーシの改造には大幅な設計見直しと追加の費用が必要となることから、ラック内での拡張は実施困難と判断した。このため、PCI スロットを経由してラック外部にてデバイスを接続する案を検討したが、専用のプロトコルとハードウェアを使用した PCI の透過的な延伸は、PCI デバイスの接続に技術的な問題が生じ得ることと、そのような特殊な技術は将来に渡って利用可能性を担保出来無いという技術の継続性の問題により、見送ることとした。

これに代わり、商用製品として利用可能でかつ将来に渡りより高性能な製品が入手可能と期待されるものとして、汎用の高速ネットワークを用いた外部デバイス接続が考えられる。この方式に対する技術的要件をまとめた上で、現在利用可能な技術として、Ethernet プロトコルに基づい

[1] 詳細については、「次世代超高速電子計算機システムに関する付加価値検討最終報告書」を参照のこと。

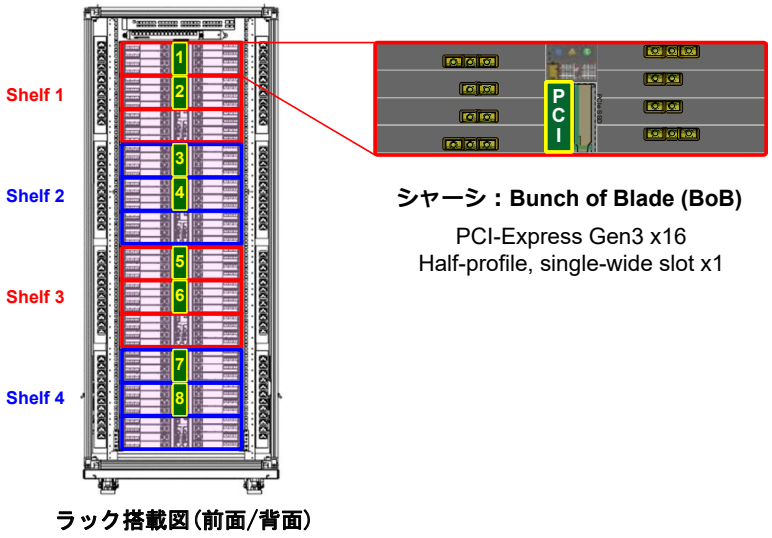


図 5.1: ポスト京ラックと PCI が利用可能なシャーシ

た PCI の延伸ハードウェアである ExpEther 、およびデバイスドライバのサービスを InfiniBand 経由で中継するソフトウェアブリッジの 2 つの方式について、実現可能性を検討することとした。

加えて、ポスト京ノードから拡張部を利用する方式として、遠隔手続き呼び出し (Remote Procedure Call, RPC) の検討を行った。拡張部分の機能を利用する遠隔呼び出しの手続きとしては、BLAS のような一般的なライブラリの他、それぞれのアプリの一部分を想定する。RPC の実装には、OmniRPC を用いた。RPC を用いて効率的に実行するためのプログラミングモデルとしてタスク並列モデルを検討した。

また、拡張部を利用するジョブを拡張部に接続された特定の計算ノードに割り当てる方法について、検討を行った。ポスト京には、例えば電力等の任意の資源をカスタム資源として定義し、それを指定して計算ノードに割り当てる機能がある。しかしながら、拡張部接続ノードは Tofu ネットワークにおいて隣接していないため、拡張部を利用するジョブに対してはこの機能をそのまま利用することができなかった。このため、以下の機能拡張を行い、拡張部デバイスというカスタム資源を指定することにより、離散的な該当計算ノードへのジョブ割り当てが可能となるようにした。

1. 計算ノード単位でカスタム資源を指定可能とする機能
2. 隣接しない計算ノードにもジョブ割り当てを可能とするジョブスケジューリング機能

5.3　拡張部アーキテクチャと試作システム

5.3.1　拡張部アーキテクチャの検討

本取組では、拡張部の基本アーキテクチャとして、大規模かつ高性能な FPGA を多数相互接続した FPGA クラスタを検討対象にすることとした。FPGA は、回路再構成機能により様々な計算問題に対して専用のハードウェア構造を実装可能な汎用デバイスであり、ポスト京のメニーコア CPU が不得意とする処理に対して高性能を実現できる見込みがある。

図 5.2 に、FPGA の構造を示す。FPGA には、ルックアップテーブルにより構成された論理要素、オンチップメモリ、演算器、I/O ブロックの他に、これらを自由に接続するための配線やス

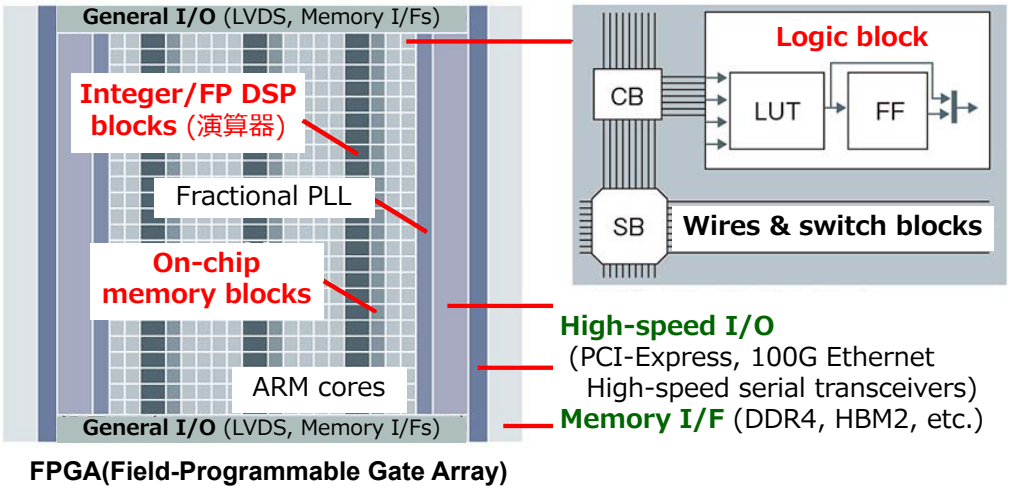


図 5.2: FPGA の構造

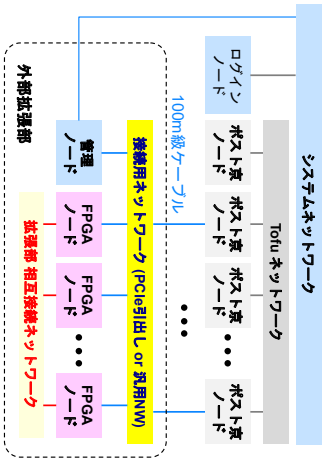


図 5.3: ポスト京拡張部のアーキテクチャ

イッチブロックが多数搭載されている。このようなハードウェア資源を用いて自由に回路を構築可能な FPGA は、特に汎用マイクロプロセッサが苦手とするような計算問題に対して、最適化された計算回路やメモリシステムを構築することにより高性能かつ高効率の計算を実現できるものと期待されている。このため、FPGA は、グルーロジックや VLSI のエミュレーション、携帯電話の基地局での利用といった当初の用途に加え、CPU では不十分な処理の加速機構として、高性能リコンフィギャラブルシステムを構築するデバイスとしての検討が行われてきている。特に近年では、最先端の半導体技術で製造されると共に HBM2 等の最先端のメモリと組み合わせて従来よりも大規模かつ高性能化が進む FPGA に対し、大量の検索処理、画像処理、機械学習処理を低電力で高速に行う産業応用として、データセンターや FPGA の採用事例が増えている。また、科学技術計算への応用のための HPC システム構築の試みも行われている。

FPGA クラスタそのもののアーキテクチャに加え、ポスト京との接続方法として、ポスト京と FPGA クラスタ間に汎用ネットワークを設ける構成の検討を行った。さらに、より詳細な技術的評価のために、検討結果である図 5.3 のアーキテクチャに基づく実験システムを開発した。本システムは、ARM 命令セット CPU を搭載したサーバ群のポスト京相当部、x86 命令セット CPU 搭載のサーバをホストとした FPGA クラスタ部、この 2 つを接続する汎用ネットワーク部、の 3 つ

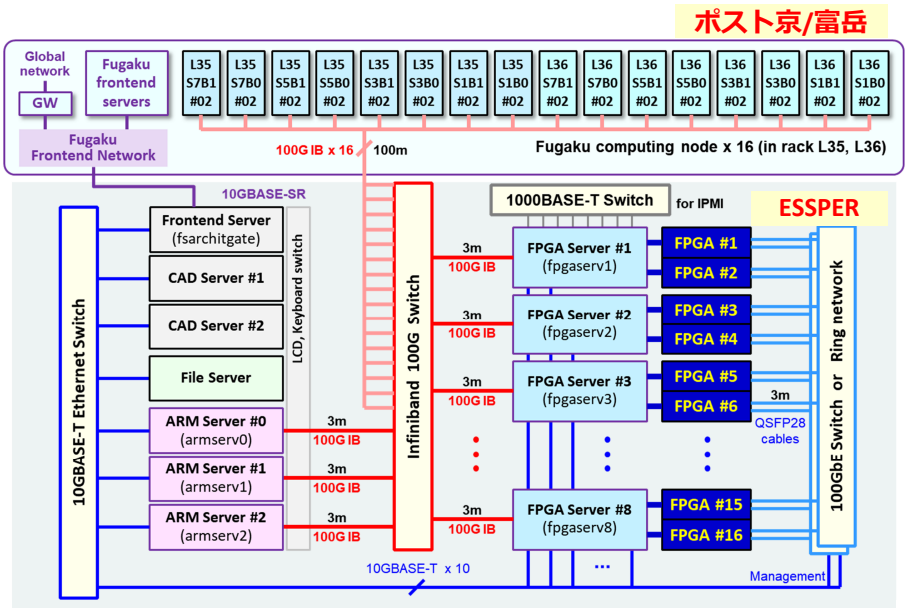


図 5.4: FPGA クラスタ試作システム ESSPER とポスト京との接続

の部分から構成される。本システムを用いてポスト京相当部と FPGA の接続方法について実機検証を行ったところ、ExpEther および 40Gb/s Ethernet ネットワークを用いた PCI の透過的延伸は、FPGA デバイスが認識されないなどの技術的問題が生じ得ることが判明したため不適との結論に至った。

一方、汎用ネットワークとソフトウェアブリッジを用いた接続の実現可能性を調査するために、Intel 社製のオープンソース FPGA ドライバである OPAE (Open Programmable Acceleration Engine) を InfiniBand ネットワークにより遠隔化した Remote-OPAE (以下、R-OPAE) を開発した。試験の結果、R-OPAE と 100Gb/s InfiniBand ネットワークを用いた接続により FPGA を遠隔で利用可能であることを確認した。以上より、ポスト京から FPGA クラスタの付加価値を利用するための拡張方式として、広帯域の InfiniBand ネットワークと R-OPAE によるソフトウェアブリッジを用いた接続のみが有望であり実現可能性が高いと判断し、ポスト京の実機を使用した試験に進むこととした。

5.3.2　試作システムの開発と実機試験

実機試験では、実験システムを再構成して構築した図 5.4 の FPGA クラスタ試作システム ESSPER (Elastic and Scalable System for High-PErformance Reconfigurable Computing) をポスト京ノードに実際に接続し、ポスト京との接続可能性、ポスト京と拡張部間の帯域等の性能、および運用可能性に関する試験を行った。図 5.5 に、ESSPER のシステムスタックを示す。その結果、FPGA 利用可というカスタムノード資源を指定可能なポスト京ジョブスケジューラやソフトウェアブリッジ化した FPGA ドライバである R-OPAE により、ポスト京の通常運用に悪影響を与えずに、拡張部である FPGA クラスタをポスト京の計算ジョブから利用可能であることが確認された。

FPGA に計算問題をオフロードし実行するための方式に関しては、独自のデータフローコンパイラ SPGen (Stream Processor Generator) 、および Intel 社の高位合成コンパイラを利用した

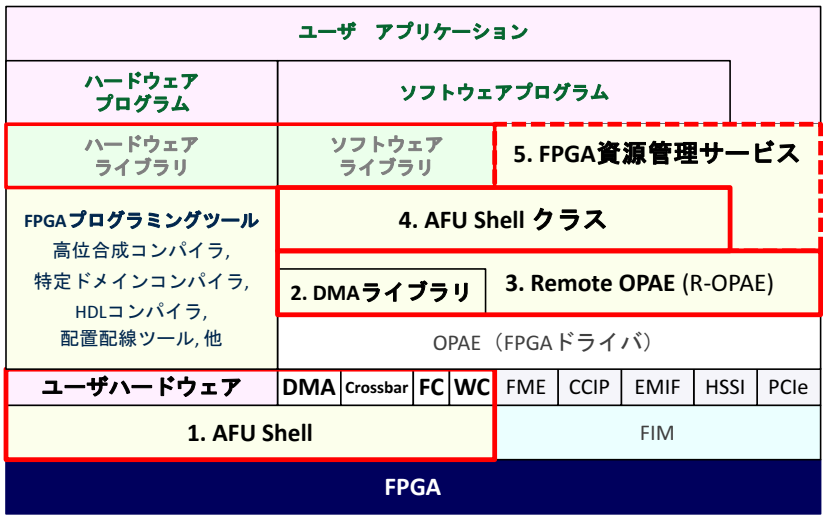


図 5.5: 実験システムのシステムスタック

FPGA プログラミング方法について検討を行うと共に、FPGA 上システムオンチップ (SoC) ハードウェアの設計と実装を行った。ユーザプログラムにおいてデバイスのオープンやホストとのデータ転送などの SoC の制御を簡単に記述可能とするために、レジスタマップといったハードウェアの詳細を隠蔽して異る SoC に対しても統一的なインタフェースを提供し、FPGA 上 SoC をオブジェクトとして取り扱うことのできるクラスライブラリを開発した。これは、FPGA 上 SoC ハードウェアの抽象化レイヤーに相当する。これらの動作検証として、SPGen コンパイラを用いて実装した 2 次元の流体計算ハードウェア等を実機にて動作させた。

また、FPGA クラスタにおいて高スケーラビリティを実現するのに求められる FPGA 間ネットワークについて、直接網と間接網の 2 種類を検討し、それぞれについて機能設計および FPGA 上 SoC の開発を行った。直接網のネットワークに関しては、100Gb/s の光ケーブルにより直結した FPGA 間においてフロー制御有りの高速シリアル通信を可能とする SoC を開発した。これにより、2 つのネットワークポートを持つ FPGA ボードをリング状に接続し、複数デバイスによるストリーム計算を実現できる。関節網によるネットワークに関しては、100Gb/s の Ethernet スイッチを用いて FPGA を相互に接続するための FPGA SoC を開発した。

加えて、拡張部として取り付けた FPGA などの演算加速機構をポスト京のノードから効率的に利用するためのプログラミングモデルとして、非同期の遠隔手続き呼び出し (Remote Procedure Call: RPC) を演算加速機構の API とするタスク並列モデルについて検討を行った。さらに、拡張部の FPGA を利用するジョブをポスト京における FPGA 接続の計算ノードに割り当てて実行可能とするために、ポスト京スケジューラのカスタム資源とスケジューリングの機能について、拡張部カスタム資源の指定と離散的なノード割り当てを可能とする拡張を行った。

5.4 アプリケーションの検討

本取組では、ポスト京の付加価値を高めるために、メニーコアプロセッサが不得意とするような処理をオフロードし FPGA クラスタ上に構築した専用回路を用いて高速化を行うことを目指している。このためには、対象とする処理を専用回路として実装することによる優位性が見い出せるようなアプリケーション、またはカーネルを選定する必要がある。

表 5.1: FPGA クラスタにより付加価値が得られそうな計算問題の候補

計算問題	ハードウェアアルゴリズムによる加速	特殊演算・特殊精度	高効率データ参照・参照・再利用	低遅延処理による恩恵	強スケーリング
ステンシル計算	◎		○		○
多体問題	◎		○		○
3D FFT	◎		◎		○
グラフ処理	△	○	△		△
ゲノムデータ処理	◎	◎	◎		○
機械学習:推論	○	◎	○		○
機械学習:学習	○	◎	○	○	○
データベース処理 *	○	◎	○		○

◎:大いに期待できる，○:期待できる，△:要調査，未記入:期待できない　* 正規表現マッチ, ソート, 近傍探索等

PCI による FPGA クラスタの拡張により付加価値が得られそうな計算問題は、以下の特徴を持つものと考えられる。

1. パイプライン、シストリックアレイ、データフロー、ストリーム処理等の、並列性・局所性・規則性の高いハードウェアアルゴリズムにより加速できる計算問題
2. マイクロプロセッサにハードウェア演算器が搭載されていないような、特殊演算処理または特殊精度を用いる計算問題
3. 専用メモリシステムを用いた高効率のデータ参照・移動・再利用により加速できる計算問題
4. フィードバック制御等にみられるように、低遅延処理が必要な計算問題
5. FPGA 間の専用ネットワークを用いた高効率の通信や同期により、強スケーリング可能な計算問題

以上の可能性のある計算問題として本取組において検討中のものは、表 5.1 の通りである。各計算問題に特化し最適化された専用回路の設計および実装は、ハードウェアアーキテクチャの探索が必要となることから、ソフトウェアプログラミングと比べて困難な作業である。このため、可能性のある計算問題から有望かつハードウェアアーキテクチャや最適化に見当が付くものを選定し、実機評価のために、FPGA クラスタに対する設計と実装を行う。

現段階では、

1. ステンシル計算や多体問題計算のストリーム処理
2. データベース処理 (グラフの幅優先探索処理)
3. 3D FFT

4. ゲノムデータ処理

等について FPGA 向け専用ハードウェアの設計と実装に取り組んでいる。1 は、これまでの研究において旧世代の FPGA での実装や評価がなされているもので、現在、ESSPER への移植と評価を行っているところである。予備的な結果として、独自開発の特定ドメイン言語向けコンパイラである SPGen (Stream Processor Generator) により格子ボルツマン法による 2 次元流体力学シミュレーションハードウェアを実装し、Stratix10 FPGA 単体で 1.09 TFlops、リング接続した 16 個の Stratix 10 で 17.3 TFlops の単精度浮動小数点実効性能を達成した。動作周波数が 150 MHz に低下する問題を解決できれば、3〜4 倍程度高い実効性能が見込まれる。

2 のグラフの幅優先探索処理は、Intel 社高位合成コンパイラ i++により ESSPER の Stratix10 FPGA 向けにプログラムを行っているところであり、ソフトウェアと同じ動作を FPGA で行う初期実装が完了した段階である。ハードウェアアルゴリズムの検討や最適化を進めている。

3,4 の FFT およびゲノムデータ処理についても、Intel 社高位合成コンパイラ i++により ESSPER の Stratix10 FPGA 向けにプログラムを行っており、初期実装に続いて最適化や複数 FPGA による並列化、性能モデルの検討、および HBM2 メモリが搭載された高速メモリ FPGA に置き換えた場合の性能見積等を行っている。

以上に加え、アプリケーションに関して、少なくとも以下が明らかとなった。SPGen や Intel 社高位合成コンパイラ i++を用いることにより、C や C++またそれに近い記述で専用ハードウェアをプログラミングでき、ESSPER のシステムソフトウェアと組合せることにより FPGA にタスクをオフロード可能であることが確認された。また、複数の FPGA を接続するリングネットワークは正常に動作しており、複数 FPGA を用いた並列化が十分に可能であることが確認された。

今後は、各アプリケーションの設計および実装を進めると共に、詳細な性能評価を行い、ポスト京との比較を実施する予定である。

5.5 共同研究に向けた準備

FPGA クラスタによりアプリケーションを高速化し、かつプログラミングや運用の生産性を高めるには、様々な研究および開発が必要となる。特に、アプリケーションの高速化には、対象となる処理毎に適したハードウェアアルゴリズムの設計と、それに基く専用回路の適切な実装や最適化が求められる。

回路再構成可能型計算（リコンフィギャラブルコンピューティング）というこれまでと異る計算機アーキテクチャの研究拠点を構築しながら、回路再構成可能デバイスによる実運用可能システムと各種アプリケーション向け専用ハードウェアに関する研究開発を加速するために、国内大学教員らと共同研究を、または、共同研究に向けた準備を進めているところである。

5.6 おわりに

本課題において研究開発を行った拡張部アーキテクチャおよび接続方式により、FPGA クラスタ試作システムをポスト京に接続し、改修したジョブスケジューラを利用してポスト京の計算ノードから利用可能であることが確認できた。これは、付加価値向上のためのポスト京拡張における基本的な技術的課題が解決されたことを意味する。

また、アプリケーションに関しては、専用回路として実装することにより優位性が見い出せるような幾つかの計算問題を検討し、特に、FFT とゲノムデータ処理について初期実装や最適化検討、および性能見積を行った。今後はこれらの研究開発をさらに進め、ポスト京との比較を実施する予定である。

最後に、本取組は、回路再構成可能型計算（リコンフィギャラブルコンピューティング）というこれまでと異る計算機アーキテクチャの研究開発に相当し、実運用可能システムの構築と専用回路化によるアプリケーションの高速化を実現するには多くの努力が必要となる。このため、本件に関する研究拠点の構築をも見据えながら、国内外の大学や研究機関と共同研究を行うべくその準備を進めているところである。以上の取組により、ポスト京に新たな付加価値を与える拡張システムの研究開発自体に加え、計算機科学分野における新しい計算方式・アーキテクチャを中心とした学術のより一層の進展が期待されるものと考えている。

参考文献

[1] Eclipse Foundation: https://www.eclipse.org/.

[2] CCS QCD Solver benchmark program. https://www.ccs.tsukuba.ac.jp/qcd/ccsqcdsolverbenchmic/.

[3] CoWoS (Chip-on-Wafer-on-Substrate) Services. https://www.tsmc.com/english/dedicatedFoundry/services/cowos.htm.

[4] Fiber Miniapp Suite. RIKEN R-CCS fiber-miniapp.github.io.

[5] FrontISTR : Open-Source Large-Scale Parallel FEM Program for Nonlinear Structural Analysis. https://gitlab.com/FrontISTR-Commons/FrontISTR.

[6] . Genesis. https://www.r-ccs.riken.jp/labs/cbrt/. Retrieved 2021-02-26.

[7] International Exascale Software Projec. https://www.exascale.org/iesp/Main_Page.

[8] NTChem Overview. https://www.r-ccs.riken.jp/software_center/software/ntchem/overview/.

[9] Power api — a reference implementation for the power api specification. http://github.com/pwrapi/.

[10] Sandia power api. http://powerapi.sandia.gov/.

[11] Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience. https://salmon-tddft.jp/.

[12] SPECFEM3D GLOBE. https://geodynamics.org/cig/software/specfem3d_globe/.

[13] The Model for Prediction Across Scales (MPAS). https://mpas-dev.github.io/.

[14] The Weather Research and Forecasting (WRF) Model. https://www.mmm.ucar.edu/weather-research-and-forecasting-model.

[15] 文部科学省：HPCI 計画推進委員会 議事要旨・議事録・配付資料. https://warp.ndl.go.jp/info:ndljp/pid/11293659/www.mext.go.jp/b_menu/shingi/chousa/shinkou/020/gijilist/index.htm.

[16] スーパーコンピュータ「富岳」における OS の強化機能, 10 月 13 日 2020 年. https://www.fujitsu.com/jp/about/resources/publications/technicalreview/2020-03/article06.html.

[17] 文部科学省：今後の HPC 技術の研究開発の検討について．HPCI 計画推進委員会（第 5 回）資料 2, 7 月 14 日 平成 23 年. https://warp.ndl.go.jp/info:ndljp/pid/11293659/www.mext.go.jp/b_menu/shingi/chousa/shinkou/020/shiryo/1309108.htm.

[18] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, and Tomohiro Inoue. The tofu interconnect d. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 646–654, 2018.

[19] D. Alejandro, A. Eduard, B. Rosa M, L. Jesus, M. Luis, M. Xavier, P. Judit. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters, Vol.21, pp.173–193, 2011.

[20] M. Dupuis, J. Rys, and H. F. King. Evaluation of molecular integrals over gaussian basis functions. *J. Chem. Phys.*, Vol. 65, No. 1, pp. 111–116, 1976.

[21] FUJITSU Ltd. Development Studio uTofu User's Guide. (2020, Mar.).

[22] Balazs Gerofi, Kohei Tarumizu, Lei Zhang, Takayuki Okamoto, Masamichi Takagi, Shinji Sumimoto, and Yutaka Ishikawa. Linux vs. Lightweight Multi-Kernels for High Performance Computing: Experiences at Pre-Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[23] H. Jasak, A. Jemcov, Z. Tukovic et al. OpenFOAM:A C++ library for complex physics simulations. International workshop on coupled methods in numerical dynamics, vol. 1000. IUC Dubrovnik, Croatia, pp.1–20, 2007.

[24] Hitoshi Murai, Masahiro Nakao, Hidetoshi Iwashita, Mitsuhisa Sato. Preliminary performance evaluation of coarray-based implementation of fiber miniapp suite using xcalablemp pgas language.

[25] Hitoshi Murai, Mitsuhisa Sato. An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language. 7th International Conference on PGAS Programming Models,Edinburgh, Scotland, UK (2013, Oct.).

[26] K. Ishimura and S. Nagase. A new algorithm of two-electron repulsion integral calculations: a combination of pople–hehre and mcmurchie–davidson methods. *Theo. Chem. Acc.*, Vol. 120, No. 1-3, pp. 185–189, 2007.

[27] M. Iwasawa, D. Namekata, K. Nomura, M. Tsubouchi, and J. Makino. Extreme-scale particle-based simulations on advanced hpc platforms. *CCF Transactions on High Performance Computing*, Vol. 2, No. 2, pp. 183–195, Jun 2020.

[28] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, and J. Makino. Implementation and performance of FDPS: a framework for developing parallel particle simulation codes. *Publications of the Astronomical Society of Japan*, Vol. 68, p. 54, August 2016.

[29] J. Jung, C. Kobayashi, and Y. Sugita. Kinetic energy definition in velocity verlet integration for accurate pressure evaluation. *J. Comput. Chem.*, Vol. 148, p. 164109, 2018.

[30] J. Jung, T. Mori, and Y. Sugita. Efficient lookup table using a linear function of inverse distance squared. *J. Comput. Chem.*, Vol. 34, No. 28, pp. 2412–2420, 2013.

[31] J. Jung, T. Mori, and Y. Sugita. Midpoint cell method for hybrid (mpi+openmp) parallelization of molecular dynamics simulations. *J. Comput. Chem.*, Vol. 35, No. 14, pp. 1064–1072, 2014.

[32] Keisuke Tsugane, Jinpil Lee, Hitoshi Murai, Mitsuhisa Sato. Multi-tasking execution in pgas language xcalablemp and communication optimization on many-core clusters.

[33] Yuetsu Kodama, Masaaki Kondo, and Mitsuhisa Sato. Evaluation of spec cpu and spec omp on the a64fx. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 553–561, 2021.

[34] Yuetsu Kodama, Tetsuya Odajima, Eishi Arima, and Mitsuhisa Sato. Evaluation of power management control on the supercomputer fugaku. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 484–493, 2020.

[35] L. E. McMurchie and E. R. Davidson. One- and two-electron integrals over cartesian gaussian functions. *J. Comput. Phys.*, Vol. 26, No. 2, pp. 218–231, 1978.

[36] K. Mikami, K. Ono, and J. Nonaka. Performance evaluation and visualization of scientific applications using pmlib. In *6th International Workshop on Large-scale HPC Application Modernization at CANDAR 2018*.

[37] M.Katouda and T. Nakajima. Mpi/openmp hybrid parallel algorithm of resolution of identity second-order moller-plesset perturbation calculation for massively parallel multicore supercomputers. *J. Chem. Theory Comput.*, Vol. 9, No. 12, pp. 5373–5378, 2013.

[38] M.Katouda, A. Naruse, Y. Hirano, and T. Nakajima. Massively parallel algorithm and implementation of ri-mp2 energy calculation for peta-scale many-core supercomputers. *J. Comput. Chem.*, Vol. 37, No. 12, pp. 5373–5378, 2013.

[39] Daichi Mukunoki and Toshiyuki Imamura. Implementation and Performance Analysis of 2.5D-PDGEMM on the K Computer. In *Proc. Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science*, Vol. 10777, pp. 348–358, 2018.

[40] Daichi Mukunoki and Toshiyuki Imamura. Performance Analysis of 2D-compatible 2.5D-PDGEMM on Knights Landing Cluster. In *Proc. ICCS 2018, Lecture Notes in Computer Science*, Vol. 10862, pp. 853–858, 2018.

[41] T. Nakajima, M. Katouda, M. Kamiya, and Y. Nakatsuka. Ntchem: A high-performance software package for quantum molecular simulation. *Int. J. Quantum Chem.*, Vol. 115, No. 5, pp. 349–359, 2015.

[42] K. Ono and K. Mikami. Pmlib - performance monitor library. https://github.com/avr-aics-riken/PMlib.

[43] J. A. Pople and W. J. Hehre. Computation of electron repulsion integrals involving contracted gaussian basis functions. *J. Comput. Phys.*, Vol. 27, No. 2, pp. 161–168, 1978.

[44] R. Sedlak, T. Janowski, M. Pitoňák, J. Řezáč, P. Pulay, and P. Hobza. Accuracy of quantum chemical methods for large noncovalent complexes. *J. Chem. Theory Comput.*, Vol. 8, No. 9, pp. 3364–3374, 2013.

[45] B. Temelso, K. A. Archer, and G. C. Shields. Benchmark structures and binding energies of small water clusters with anharmonicity corrections. *J. Phys. Chem. A*, Vol. 115, No. 43, p. 12034–12046, 2011.

[46] J. Řezáč, K. E. Riley, and P. Hobza. Benchmark calculations of noncovalent interactions of halogenated molecules. *J. Chem. Theory Comput.*, Vol. 8, No. 11, pp. 4285–4292, 2010.

[47] J. Řezáč, K. E. Riley, and P. Hobza. S66: A well-balanced database of benchmark interaction energies relevant to biomolecular structures. *J. Chem. Theory Comput.*, Vol. 7, No. 8, p. 2427–2438, 2011.

[48] X. Gonze, B. Amadon, P.-M. Anglade, J.-M. Beuken, et. al. ABINIT: First-principles approach to material and nanosystem properties. Computer Physics Communications 180, 2582 - 2615 (2009). Abinit Project home page: https://www.abinit.org/.

[49] H. Yashiro, K. Terasaki, T. Miyoshi, and H. Tomita. Performance evaluation of a throughput-aware framework for ensemble data assimilation: the case of nicam-letkf. *Geoscientific Model Development*, Vol. 9, No. 7, pp. 2293–2300, 2016.

[50] 石川 裕. ポスト「京」の開発状況, 1 月 29 日 平成 28 年. https://www.r-ccs.riken.jp/r-ccssite/wp-content/uploads/2016/01/4ishikawa.pdf.

[51] 内閣府. 「エクサスケール・スーパーコンピュータ開発プロジェクト」の概要. 評価専門調査会（第 103 回）資料 6-2, 11 月 20 日 平成 25 年. https://www8.cao.go.jp/cstp/tyousakai/hyouka/haihu103/haihu-si103.html.

[52] 文部科学省 HPCI 計画推進委員会. 次期フラッグシップシステムに係るシステム検討ワーキンググループ報告書, 10 月平成 26 年. https://www.mext.go.jp/b_menu/shingi/chousa/shinkou/037/gaiyou/__icsFiles/afieldfile/2014/12/26/1353622_1.pdf.

[53] 鈴木惣一朗, 伊東聰, 稲田由江, 池田奈生, 三吉郁夫, 松葉浩也, 石川裕, 宮野悟. DNA アライメントプログラム BWA-MEM の Arm SVE 移植と「富岳」試作機での性能評価. 情報処理学会研究報告, Vol. 2020-HPC-173, No. 16, pp. 1–8, 2020.

[54] 鈴木惣一朗, 伊東聰, 酒井憲一郎, 稲田由江, 三吉郁夫, 石川裕, 宮野悟. ヒト全ゲノム解析プログラム Genomon のスーパーコンピュータ「富岳」向け最適化と性能評価. 情報処理学会研究報告, Vol. 2021-HPC-178, No. 18, pp. 1–9, 2021.

[55] 鈴木惣一朗, 伊東聰, 池田奈生, Balazs Gerofi, 三吉郁夫, 丸山直也, 滝澤真一朗, 村瀬洋介, 石川裕, 宮野悟. ヒトゲノム解析プログラム Genomon-fusion の大規模スーパーコンピュータ向け最適化と性能モデル化. ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集, pp. 15–26, 2016.

第6章　主な共同研究

フラッグシップ2020プロジェクトにおいては、以下の機関との共同研究を行った（カッコ内にテーマを示す)。

- 東京大学（ポスト京のプロセッサアーキテクチャ、電力制御技術、システムソフトウエアおよび数値計算ライブラリに関する研究）
- 筑波大学（ポスト京の並列プログラミング環境およびネットワークに関する研究）
- 京都大学 (ポスト京の高並列SIMD機構およびプロセッサアーキテクチャに関する研究)
- 東洋大学 (プロセッサアーキテクチャの設計・実装技術に関する研究)
- 九州大学・ZIB（Study on Optimization for Graph problem on Supercomputer ”Fugaku”）
- 九州大学（集団通信の最適化手法に関する研究）
- 名古屋大学（ポスト「京」のDeep Learning応用に関する研究）
- 東京大学（ポスト「京」におけるインターネットアクセス環境に関する研究）
- 千葉工業大学（性能チューニングおよび性能解析のための解析支援ツールの研究開発およびそのツールを用いたスーパーコンピューター「富岳」向けアプリケーションの性能チューニング・性能解析）

このうち、主な共同研究として、東京大学、筑波大学、京都大学との共同研究の詳細について報告する。

6.1　東京大学

「ポスト京のプロセッサアーキテクチャ，電力制御技術，システムソフトウエアおよび数値計算ライブラリに関する研究」

6.1.1　概要

6.1.1.1　プロセッサ・アーキテクチャ

2020年度の研究では，ポスト京コンピュータのプロセッサ・コンパイラの性能評価やコードチューニングを行う上で利用され，また将来的なシステム開発にも有益である，シミュレーション環境の機能拡張を行った。特に，電力推定ツールに対して変更を加え，A64FXプロセッサを3nmまでスケーリングさせた際の電力，面積削減効果を見積り，その上で，チップ電力，面積等の制約の

下，スケーリングによるチップのスループット向上効果について，特に，コア数，SIMD 幅，FP パイプライン幅をスケールさせて検証を行なった。その結果，コア数の増大による性能向上は見込めるものの，SIMD 幅，パイプライン幅の増大については，コアの複雑さ，特に L1 データキャッシュの複雑さの増大により，性能向上が頭打ちとなる事を確認した。さらに，これら定量的評価結果を元に，今後の HPC 向け SIMD プロセッサの方向性についても検討を行なった。

6.1.1.2 電力制御技術

スーパーコンピュータ「富岳」，さらには将来の大規模高性能計算システムは，消費電力がシステム設計や実効性能を制約する最大の要因となっており，洗練された電力マネジメント手法が不可欠である。一般に大規模システムの電力マネジメントは，スケジューラによるノード/ジョブへの電力配分最適化 (グローバルな制御) 及び，各ノード/ジョブにおける与えられた電力予算の有効活用 (ローカルな制御) からなり，これまでそれらに関して研究を行ってきたが，2020 年度は特にローカルな制御に注力して検討を行った。具体的には，低精度演算の積極的利用による電力効率の改善に向け，近年の GPU が持つ混合精度行列演算回路を用いた場合の電力を，QR 分解を行うアプリケーションを用いて評価した。さらには，ノード内の電力モード選択の最適化についても検討を行った。

6.1.1.3 数値計算ライブラリ

多重格子法（Multigrid Method）は，反復回数が問題規模によらないスケーラブルな手法であり，大規模問題向け手法として注目されている。2020 年度は，並列多重格子法のメニィコアクラスタにおけるノード内性能の最適化に注目し，SIMD 演算向けに提案された疎行列格納形式である SELL-C-σ （参考文献 [30]）を実装し，Oakforest-PACS システム（最先端共同 HPC 基盤施設）（参考文献 [27]）等を使用した評価を実施した [4] [11] [12] [13]。

エクサスケールシステムにおける高性能数値アルゴリズム実現には，メモリ・ネットワークの階層の深化に対応した通信最適化（Serial，Parallel）とともに省電力・省エネルギーに向けた検討が必要である。近年，科学技術計算において，低精度演算を積極的に活用することにより，計算時間を短縮し省電力・省エネルギーを図る試みが活発に行われている。数値計算による近似解（数値解）は様々な計算誤差を含み，計算結果の信頼性の観点から，数値解の正しさを数学的に保証する必要があり，低精度・変動精度使用時，悪条件問題には重要であるが，実問題で現れる大規模疎行列・H 行列への応用例はほとんどない．本研究では，有限要素法，有限体積法から導出される大規模疎行列を係数行列とする連立一次方程式の前処理付き反復法（ICCG 法）による求解に着目し，疎行列格納手法，問題規模と低精度演算による性能改善の関係に注目し，様々な計算機環境下での検討を実施した。本研究では,「富岳」と同じ A64FX を搭載している Fujitsu PRIMEHPC FX700 システム（東京大学情報基盤センター）を使用して半精度演算（FP16）の効果について検討した [3] [8] [9] [10] [13]。更に，前述した SELL-C-σ について FX700 上での低精度演算に関するフィージビリティスタディを実施した。

本研究は，SELL-C-σ を前進後退代入を含む IC/ILU 系の計算プロセスに適用したものとしては，世界で初めての事例である。

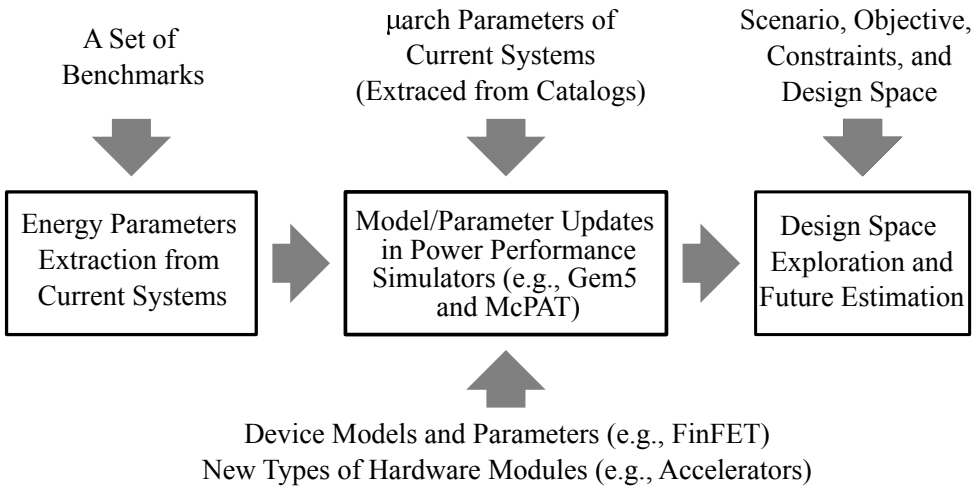


図 6.1: 評価における全体的なワークフロー

6.1.1.4 システムソフトウェア

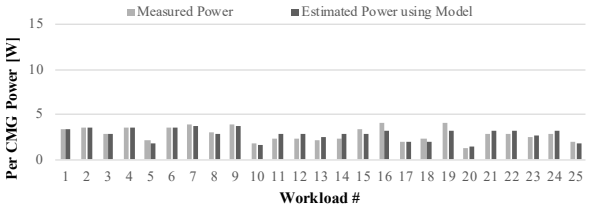
2020 年度研究で開発した SELL-C-σ に基づく並列多重格子法ソルバーを，2019 年度に引き続き，最先端共同 HPC 基盤施設 (JCAHPC) で運用している Oakforest-PACS システムにおいて，理研 R-CCS において開発中のシステムソフトウェア群のうち IHK/McKernel を使用して AM-hCGA 法の性能評価を実施した。IHK/McKernel の利用によって最大 11%の性能改善が得られることを示した [4]。

6.1.2 プロセッサアーキテクチャ・電力制御技術

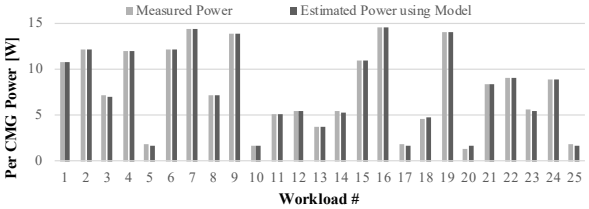
2020 本年度の研究では，ポスト京のプロセッサアーキテクチャや電力制御に関連した研究開発を実施した。以下，その内容に関して報告する。

6.1.2.1 プロセッサアーキテクチャ

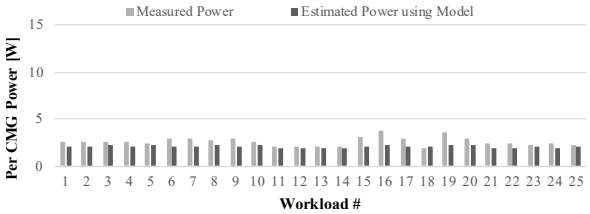
本年度は，将来的な HPC システム向けプロセッサの性能，電力の予測を行うための，プロセッサアーキテクチャ評価環境の整備を行い，さらに実際にこれを用いた電力性能評価を，特に半導体製造プロセスの微細化によるコストの増大，電力供給限界等を勘案しつつ行った。具体的には，現行の FinFET トランジスタによって到達可能と見られている 3nm までを対象とし，A64FX プロセッサをスケーリングさせた際の電力・面積削減効果を見積り，その上で，チップ電力及び，面積制約の下，スケーリングによるチップのスループット向上効果を，特に，コア数，SIMD 幅，FP パイプライン幅をスケーリングさせて検証した。さらに，これら定量的評価結果を元に，今後の HPC 向け SIMD プロセッサの方向性について検討を行なった。以上の研究成果については，HPC 研究会にて発表 (成果 [5]) するとともに，査読付きの国際会議にて発表 (成果 [6]) を行った。加えて，キャッシュリ置換アルゴリズムに関しても検討を行い，査読付き国際会議にて発表を行った (成果 [1])。また，スクラッチパッドメモリの有効利用法についても国際会議にて発表を行なった (成果 [19])。



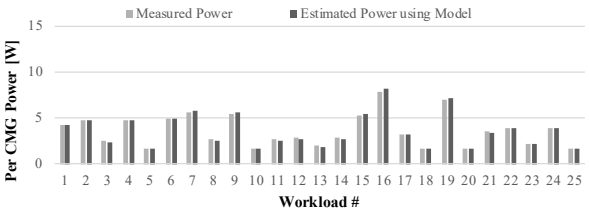
Sreaming アクセス時の L2 の電力比較 (CMG 単位)



Sreaming アクセス時の HBM の電力比較 (CMG 単位)



ランダムアクセス時の L2 の電力比較 (CMG 単位)



ランダムアクセス時の HBM の電力比較 (CMG 単位)

図 6.2: 計測電力とモデルによる予測値の比較

6.1.2.1.1 電力性能予測の概要

図 6.1 に，本研究における電力性能予測のための，全体的なワークフローを示す。中央部分は，電力性能予測のためのシミュレーション環境である。本研究ではこれまで通り，性能評価には Gem5，電力面積評価には McPAT を用いた。ここでの目標は，A64FX プロセッサを元に，HPC 向けプロセッサの将来の進むべき方向性を確認することであり，これを行うために，上記ツール内のパラメタやコードの修正を行った。具体的には，A64FX のマイクロアーキテクチャパラメタ及び電力パラメタの双方を，これらの評価環境に入力し，性能，電力，面積の見積もりを行った上で，これらのスケーリングに応じた変化について，FinFET デバイスモデルを用いて評価を行う。ここで，マイクロアーキテクチャパラメタについては，富士通から公開されている A64FX Microarchitecture Manual(文献 [21]) を用いてパラメータチューニングされた RIKEN Simulator(文献 [22]) を用い，電力パラメタについては図に示す通り，シンセティックなベンチマークコードを用いて実機からパラメタを推定することによって行った。

6.1.2.1.2 A64FX 実機を用いた電力パラメタ推定

本研究では，特に L2 キャッシュ，メインメモリ (HBM)，及び FPU の電力パラメタを実機から推定し，シミュレーション環境に反映した。ここでは，特に重要なこれらのコンポーネントに限定しているが，ハードウェアパフォーマウンスカウンタの対応状況によっては，ベンチマークを適切に用意することで，その他コンポーネントの電力パラメタの推定も可能である。

L2 キャッシュ・HBM の電力パラメタ推定:

一般的に，キャッシュやメインメモリを含むメモリシステムの電力 (P_*) は以下の様な数式で表すことができる (*は l2 や hbm 等対象とするメモリ階層)。

$$P_* \quad = \quad E_*^{dy} F_* + P_*^{st} \tag{6.1}$$

ただし，E_*^{dy}，F_*，P_*^{st} は対象メモリ階層に対する 1 回当たりのアクセスエネルギー (キャッシュ

ライン単位)，単位時間当たりのアクセス回数 (キャッシュライン単位)，スタティック電力を表す。

一方で A64FX では，ハードウェアパフォーマンスカウンタの値から，L2 及びメインメモリ (HBM) の消費エネルギーの推定値を取得することが可能であり，さらに，これらのコンポーネントに対するアクセス数も取得することが可能である (CMG 単位での集計となる)。これらを実行時間で割ることにより，前述の P_* 及び F_* を求めることができ，これらペアを複数用いて回帰分析を行うことで，電力パラメタ E_*^{dy} 及び P_*^{st} を推定することができる。ただし，その他の商用プロセッサでも同等の機能を有するのが一般的であるため，本手法は一般的に成り立つものであることに注意されたい。

そこで，実際に A64FX を搭載するシステム (FX700) を用いて当該実験を行なった。具体的には，配列へのストリーミングアクセスを行うコードを用意し (B[]+=A[])，各メモリ階層へのアクセス頻度が変化する様に，データサイズ (16MiB, 64MiB, 256MiB, 1GiB, 4GiB) 及びスレッド数 (4, 8, 12, 16, 48) を変化させて実行し，実行毎に対象区間のカウンタ値を取得し，得られた値を用いて回帰分析を行なった。実機評価環境は，表 6.1 の通りである。スレッドアフィニティの設定は scatter であり，ストリーミング，ハードウェアパフォーマンスカウンタへのアクセスは PAPI6.0(文献 [23]) を用いて行なった。その結果得られたパラメタを表 6.2 に示す。

さらに，この評価結果の当てはまりについて，ストリーミング (上記と同様) 及びランダムアクセスを行うコード (A[I[]]+=x) を用いて検討を行なった。その評価結果を図 6.2 に示す。ただし，横軸の番号はある実行条件 (スレッド数，データサイズ) に対応しており，全部で 25 通りである。ストリーミングとランダムアクセスを比較した場合，L2，HBM 共にランダムアクセス時の電力が低くなっているが，これはランダムアクセスでは一般的にアクセス頻度が下がるためである。また，L2 のランダムアクセスでは，誤差が若干増大しているが，これは，キャッシュコヒーレンスや CMG を跨ぐトラフィック増加等の影響であると考えられ，それによる影響を F_* に反映する等により，対処することができる。

FPU の電力パラメタ推定:

浮動小数点演算 1 回当たりに要する FPU のエネルギーを算出するため，図 6.3 に示すようなコードを用いた。図中の最上ループはコピーのみを行うものであり，それ以外は，ロード・ストア数不変のまま，1 ループあたり各々積和 (mla)，和 (add)，積 (mul) を 1 命令追加で行うものである。ここで，本評価環境では，対応した sve 命令が追加されることを確認している。前述同様，各々のループの前後で PAPI を用いてハードウェアカウンタにアクセスし，総浮動小数点演算回数 (N_*) 及びコア部分の消費電力 (P_*) を各々のコードで取得できる (* = mla, add, mul, copy)。ただし，$N_{copy} = 0$ となる。

表 6.1: FX700 システム構成

Name	Remarks
CPU Package	A64FX@1.8GHz x1 socket, 4CMGs, 12 cores/CMG, 8MiB L2/CMG (= total 48 cores, 32MiB L2)
Main Memory	8GiB HBM@256GiB/s per CMG (= total 32GiB, 1TiB/s)
Compiler	Fujitsu C/C++ Compiler 4.2.0, **Options:** -Kfast -Kopenmp_simd -Kopenmp -O3
OS	Red Hat Enterprise Linux release 8.0

```
//copy loop
#pragma omp parallel for simd
for (i = 0; i < N; i ++)
        B[i] = A[i];

//add loop
#pragma omp parallel for simd
for (i = 0; i < N; i ++)
        B[i] = A[i] + 2.0;
```

```
//mla loop
#pragma omp parallel for simd
for (i = 0; i < N; i ++)
        B[i] = A[i] * A[i] + 2.0;

//mul loop
#pragma omp parallel for simd
for (i = 0; i < N; i ++)
        B[i] = A[i] * A[i];
```

図 6.3: 浮動小数点演算に要するエネルギーの算出用コード

各浮動小数点演算を 1 要素実行するのにかかるエネルギー (fetch から commit までの全パイプラインステージを含む) を E_* とおくと (* = mla, add, mul)，上述のコピーのみを行うループと比較することにより，これは以下の数式で表すことができる。ただし，T_* は総実行時間であり，$T_* \simeq T_{copy}$ (* = mla, add, mul) となることを確認している。

$$E_* = (P_* - P_{copy})T_*/N_* \qquad (* = mla, mul, add) \tag{6.2}$$

上述のエネルギー E_* のうち，FPU での演算を除いた部分のエネルギーを E_{else} とおき，FPU での 1 要素の演算に要するエネルギーを E_{flop} とおけば，E_* (*=mla, mul, add) は以下の様に表すことができる。

$$E_* = \begin{cases} 2E_{flops} + E_{else}, & (* = mla) \\ E_{flops} + E_{else}, & (* = mul, add) \end{cases} \tag{6.3}$$

これら定式化を元に，E_{flop} 及び E_{else} を求めた。具体的には，E_* (* = mla, add, mul) を求める際には，総データサイズ 1GiB，2GiB，4GiB，8GiB の 4 通りについて，各ループを 1K 回実行し，平均を取ることでこれを行い (48 スレッドで実行)，E_{else} は $E_{add} + E_{mul} - E_{mla}$ から求め，E_{flops} は $E_{mla} - E_{add}$ と $E_{mla} - E_{mul}$ の平均を取ることにより求めた。この評価結果をまとめたものを表 6.3 に示す (ただし，倍精度の結果である)。FX700 では，1 コアで 56GFLOPS，チップ全体で 2.7TFLOPS の浮動小数点演算性能を有していることから，表のパラメタを用いることで，FPU の動作消費電力は 1 コア当たり最大約 0.71W 程度，チップ全体で最大約 34W と推定され，E_{else} 部分を含めると，前者は約 0.87W，後者は約 42W と推定される。

表 6.2: 推定されたメモリ電力パラメタ

Component	Parameters
L2 cache	E^{dy}_{l2}=2.44 ± 0.31[nJ],P^{st}_{l2}=1.51 ± 0.18[W]
HBM	E^{dy}_{hbm}=14.8 ± 0.1[nJ],P^{st}_{hbm}=1.62 ± 0.04[W]

6.1.2.1.3 シミュレーション環境と評価内容

まず，A64FX プロセッサにおけるアプリケーション実行時の性能や各コンポーネントの利用率等詳細な統計情報を取得するため，理研等における先行研究を踏襲し，富士通から公開されている A64FX Microarchitecture Manual(文献 [21]) を用いてパラメータチューニングされた RIKEN Simulator(文献 [22]) を用いる。一方，電力や面積評価についてはマイクロアーキテクチャ分野で広く用いられている McPAT，特にその FinFET 拡張版 (文献 [24]) を用い，7nm における，FPU，L2 キャッシュのエネルギーが前述の評価結果と同等になるように，シミュレータ内で用いられているモデルに係数を掛けることで行なった。その他，メモリについてはメモリコントローラの PHY 部分の面積，電力が HBM 向けのそれとなる様に係数を掛けて調整し (文献 [25])，HBM 部分の電力評価は McPAT には存在しないため，この部分についてはスケーリングとは独立に，前述の実測に基づく線形回帰モデルをそのまま用いている。また，スケーリングによる静電容量等のパラメタの変化はツールのものを用いている。

評価については，まず，7nm における A64FX プロセッサの電力 (本稿では，DGEMM または STREAM コード実行時)，及び面積について実機とシミュレーション結果を比較し，その上でさらに，3nm までスケーリングさせた場合の面積，電力削減効果を見積もる。ここで，DGEMM については，富士通コンパイラの BLAS ライブラリを用いており，STREAM については，配列のコピーのみを行う自作のコードを用いた。

その上で，3nm スケーリング時における，チップの浮動小数点演算スループットの向上効果について，特に，電力，面積制約を考慮しつつ行う。将来的な HPC 向けプロセッサについては，アクセラレータや異なるメモリデバイスを含むヘテロジニアスな構成も有効であるが，本稿でのスコープは図 6.4 に示すとおり，A64FX の設計を踏襲した，アウト・オブ・オーダーコア部分や HBM 部分のみに限る。このアウト・オブ・オーダーコア部分について，総浮動小数点演算性能のピーク値 (R_{peak}) は，演算器が 1 クロック内に 1 要素に行える最大演算数 (α)，1SIMD 演算のビット数 (W_{simd})，コア内の FP パイプライン数 (W_{pipe})，1 要素のビット数 (N_{bit})，コア数 (N_{cores})，動作周波数 (F) を用いて次の式で表される。

$$R_{peak} = \alpha \times W_{simd}/N_{bit} \times W_{pipe} \times N_{cores} \times F \tag{6.4}$$

本稿では，まずコア数 N_{cores} のみスケーリングさせた場合に到達できるピーク性能の向上について考え，その上で，コア内の浮動小数点性能，すなわち，SIMD 幅 (W_{simd})，及び FP パイプライン幅 (W_{pipe}) を向上させた場合の更なる面積，電力効率の向上について検証を行う。このスケーリングに関する詳細は，図 6.5 に示す通りである。まず，FP レジスタファイル (FRF) は複数要素 (評価では倍精度で 8 要素) ごとに分割され，各々の FP レジスタファイルは対応する FPU 群に密に接続された状況を考える。SIMD 幅をスケーリングさせる場合，この固定されたサイズの FPU 群及び FP レジスタファイルのペアの数をスケーリングする事に相当する。一方，FP パイプライン数をスケーリングさせた場合には，FP レジスタファイルに接続される FPU 数も増える事になり，その結果として必要なポート数もそれに応じて増やす必要があり，その上，各種リソースの幅やエントリ数も増やす必要がある。これらに加えて，L1 データキャッシュへのアクセスについ

表 6.3: 推定された倍精度浮動小数点演算にかかる各種エネルギー

E_{mla}	E_{add}	E_{mul}	E_{flops}	E_{else}
31.10[pJ]	18.83[pJ]	17.94[pJ]	12.71[pJ]	5.676[pJ]

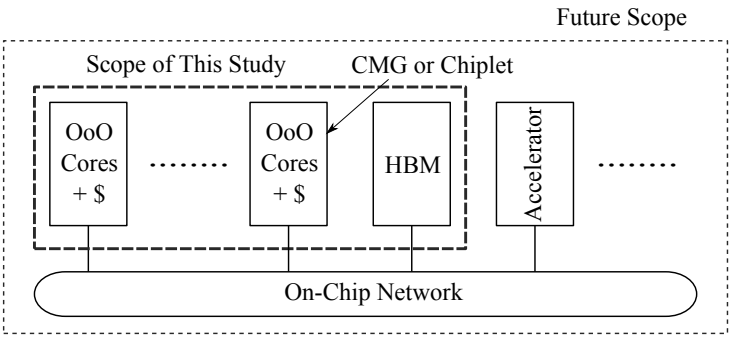


図 6.4: 評価のスコープ

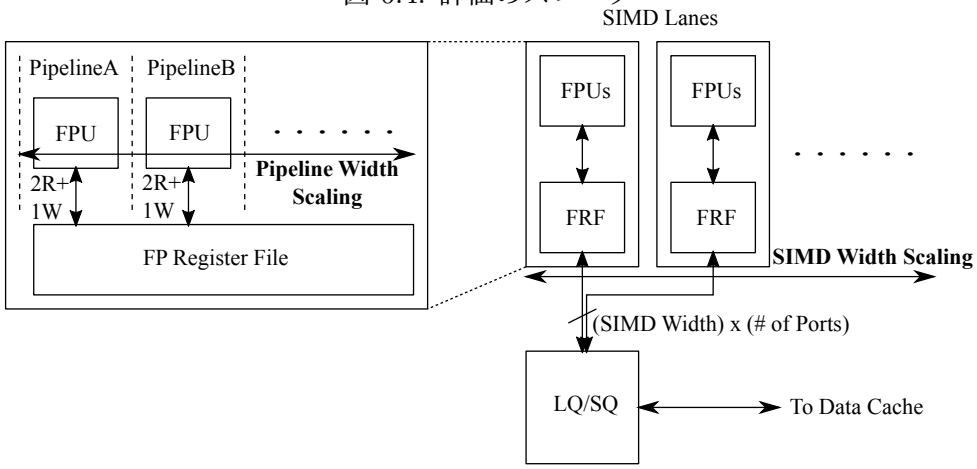


図 6.5: SIMD 幅及び FP パイプライン数のスケーリング

ても，スループットを向上させる必要があり，本評価では，ラインサイズを SIMD 幅及び FP パイプライン幅に応じてスケーリングさせている。さらには，L1 データキャッシュの容量についても，配置できるパイプライン当たりのベクトル本数を一定に保つため，スケーリングさせている (そうでなければ，ベクトル化がより困難となるためである)。これら，リソースの変更についてまとめたものを表 6.4 に示す。

6.1.2.1.4 評価結果

図 6.6 では，STREAM/DGEMM 実行時の実機で取得した電力，シミュレータが算出した電力を比較しており，さらにスケーリングによる電力削減効果をシミュレーションにより見積もっている。ただし，縦軸の電力は 1CMG 当たりのものである。DGEMM では特にシミュレーションによる数値の方が小さく出力されているが，これは，実機ではセクタキャッシュ等の利用によりコア当たりの性能が 48.4GFLOPS 出ているのに対し，シミュレーションではその様な最適化には対応しておらず，32GFLOPS 程度が限界となるためである。FPU 及び FP レジスタファイルの利用率が 100%となった場合には，シミュレーションにおいても，実機の DGEMM と同等のコア電力を消費することを確認している。また，シミュレーションについて，STREAM 実行時のコアの消費電力は L1 データキャッシュが支配的なのに対し，DGEMM では，L1 データキャッシュの電力はそれと比較すると少なくなっており，FPU や FP レジスタファイルの電力も大部分を占めている。また，HBM 部分については，HBM2.0 で固定としているため，電力はほぼ変化していないが，こ

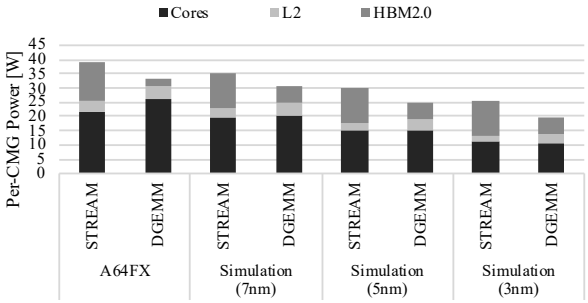


図 6.6: STREAM/DGEMM 実行時の電力比較及びスケーリングによる電力削減効果

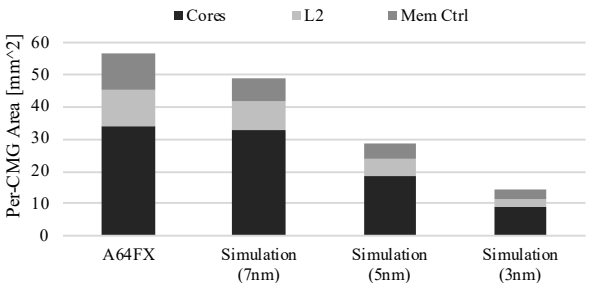


図 6.7: 面積比較及びスケーリングによる面積削減効果

れを HBM3.0 等に置き換えることで，電力削減効果が見込める。

さらに，実機とシミュレーションの面積比較，及びスケーリングによる面積削減効果のシミュレーション結果を図 6.7 に示す。ただし，ここでは，チップ内の 1CMG に着目しており，コア部分の総面積，L2 キャッシュ，メモリコントローラ部分について示している。ここで，実機の面積については，A64FX プロセッサのパッケージのサイズが 60mmx60mm であることが分かっており，さらに，パッケージや CPU ダイの画像のピクセル数をカウントすることで，パッケージサイズに対する CPU ダイサイズの比率及び，CPU ダイサイズに対する各コンポーネントのサイズの比率を算出することができるため，これらを掛け合わせることで面積を算出した。図に示す通り，特にコア部分については，実機とほぼ同等の面積をシミュレータが出力しており，さらに，スケーリングによる面積削減効果も見込めることが分かった。

図 6.8 に FPU 及び各メモリ階層における，1SIMD 要素の操作に必要なエネルギーを，特に 7nm 及び 3nm の場合について示す。ここで，HBM については，7nm では HBM2.0，3nm では HBM3.0 を用いており，後者のエネルギーについては文献 [26] を参考にしている。また，図の縦軸はログスケールであることに注意されたい。図に示す通り，HBM へのアクセスエネルギーは他と比べて 1 桁以上相対的に大きくなっている。各コンポーネントに対する電力バジェットは限りがあるため，それによりスループットが制限を受け，結果として浮動小数点演算スループットに対するメモリバンド幅が足りなくなる。従って，FPU と同等のエネルギー消費に抑えられている，L1 データキャッシュや FP レジスタファイルにできるだけデータを格納し，計算を行うことが重要であることが，電力・エネルギーの観点からも分かる。また，3nm にスケーリングすることで，各階層で半分程度にエネルギーを削減できているが，倍精度ではなく，単精度，半精度の浮動小数点演算

表 6.4: FP リソースのスケーリング評価に用いる設定

W_{simd}xW_{pipe}	Parameter Settings
512x2	A64FX default
1024x2	(linesize)x2, (L1 dcache size)x2, (FRF size)x2
2048x2	(linesize)x4, (L1 dcache size)x4, (FRF size)x4
512x4	(linesize)x2, (L1 dcache size)x2, (FRF size)x2, (FRF # of ports)x2, (commit/issue width)+=2, (# of hardware threads)x2

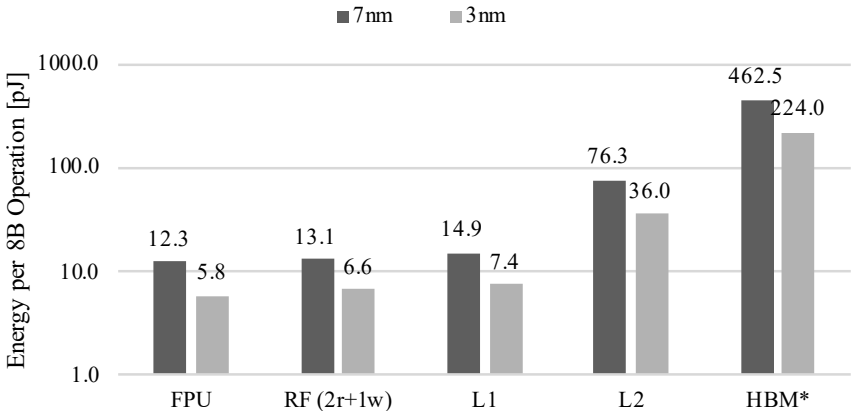


図 6.8: SIMD1 要素の操作に要するエネルギー (倍精度)

を用いることで，同様に半分，1/4 に削減できるため，これらを用いるのは数世代スケーリングを推し進めるのと同等の効果があることが分かる。

次に，電力バジェットに対するコア数のスケーリング効果を特に 5nm 及び 3nm の場合について図 6.9 に示す。ただし横軸は与えられた電力バジェットを表しており，縦軸はコア数を表している (いずれも 7nm 時と比較した場合の相対値)。また，ここでの電力バジェットは，L2 キャッシュやメモリコントローラ等を除いたコア群全体のみを指している。図に示す通り，電力バジェットをこれまでと同等に設定した場合でも，コア数のスケーリング効果はあり，3nm では，2.5 倍程度にコア数をスケールさせることができる。

一方，図 6.10 では，面積バジェットに対するコア数のスケーリング効果を特に 5nm 及び 3nm の場合について示す。ただし，ここでも，横軸は与えられた面積バジェットを表しており，縦軸はコア数を表しており，いずれも 7nm 時と比較した場合の相対値である。また，ここでの面積バジェットも同様に，L2 キャッシュやメモリコントローラ等を除いたコア群全体のみに限定する。スケーリングによる面積削減効果は期待できるものの，チップの製造コストはスケーリングに応じて増大していることが指摘されており，そのため，コストを固定した場合の面積バジェットは削減される。仮に，スケーリング 1 世代毎に面積バジェットが 0.8 倍になると仮定した場合，図に示す通り，3nm 時点ではコア数は 2 倍強にスケーリングされる。

以上により，電力・面積の両観点から，スケーリングによるコア数の増大効果は 2 倍程度見込めることが分かったが，ここでは，さらに，コア内のリソース量の変化 (特に SIMD 幅及び FP パイプライン幅) についても確認し，更なるチップのスループット向上効果が見込めるかどうかを確認する。この評価の詳細については前述の通りであり，表 6.4 に具体的な設定を示している。これらをスケーリングさせた際の電力の変化を図 6.11 に (ただし最大電力である)，そして面積の変化を図 6.12 に示す。前述の通り，演算律速のワークロードの場合，FPU，FP レジスタファイル (FRF)，及び L1 データキャッシュのバランスが重要であり (図 6.8 参照)，ここでは特にこれらに着目する。図に示す通り，SIMD 幅，FP パイプライン幅スケーリングのいずれの場合も，L1 データキャッシュのオーバーヘッドが大きく，面積，電力の面で足を引っ張ることが分かる。これは，RAM の面積，エネルギーが，一般的に，容量やポート数やラインサイズ等，バンド幅関連の変数の多項式関数で表され，特に 2 乗の項の影響が無視できないためである。これは，配線の影響と考えられ，特にスケーリングが進んだ際には支配的となる。また，FP パイプライン幅のスケーリングについては，パイプライン幅の増大に応じてポート数を増やす必要があり，結果として，FP

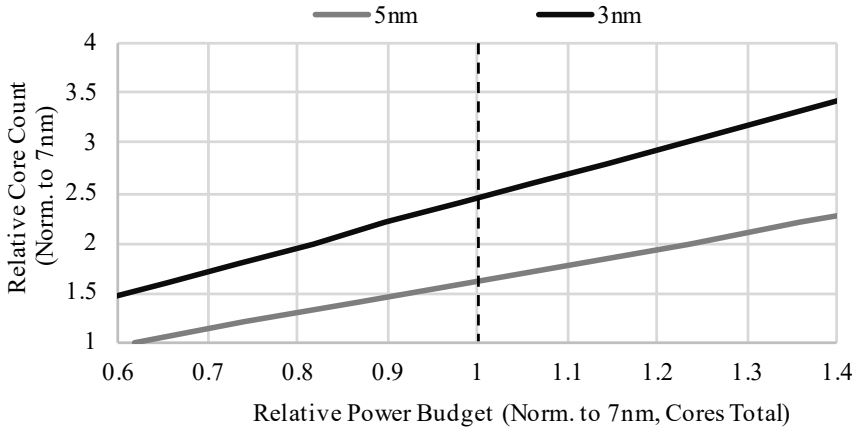


図 6.9: 電力バジェットに対するコア数のスケーリング効果

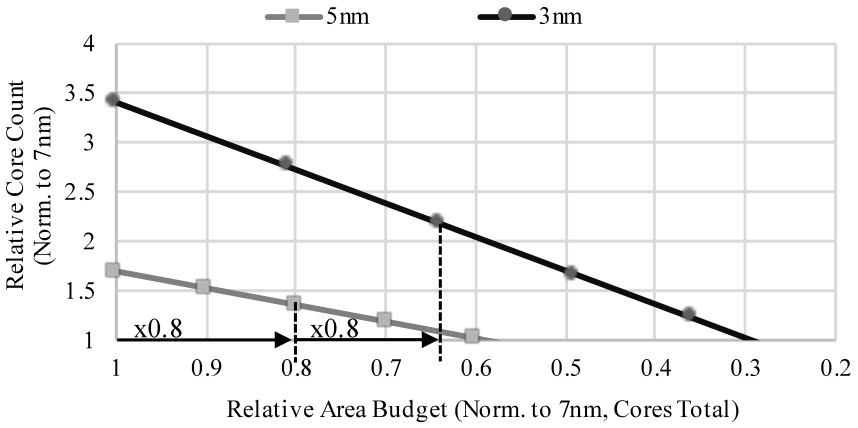


図 6.10: 面積バジェットに対するコア数のスケーリング効果

レジスタファイルのオーバーヘッドが他と比較して大きくなっている。

次に，図 6.13 に，これらのコア設定について，面積当たりの性能及び電力当たりの性能を比較したものを示す。ただし，面積，電力については，前述同様にコア部分のみに限っており，性能はピーク倍精度浮動小数点演算性能を表している。図に示す通り，SIMD 幅や FP パイプライン幅をスケーリングさせても，前述のオーバーヘッドにより，面積対性能，電力対性能ともに向上しないことが分かる。特に，FP パイプライン幅のスケーリングについては，アウト・オブ・オーダー実行リソースの増大 (幅，エントリ数) が必要となり，そのためのオーバーヘッドが追加されるため，得策ではない。一方で，図 6.14 に，SIMD 幅や FP パイプライン幅をスケーリングさせても，L1 データキャッシュの面積や電力が増えないという理想的な状況の場合の面積対性能，電力対性能も示しておく。図に示す通り，この場合は，SIMD 幅スケーリングについては，面積対性能，電力対性能の向上が見込めることが分かる。

6.1.2.1.5 HPC 向け SIMD プロセッサの方向性

SIMD 幅のスケーリングについては，アウト・オブ・オーダー実行機構の複雑さを増やすことなく，FP 演算性能を向上させることができるため，一見すると効果的の様に思われるが，実際には現時点の 512bit で限界に近づいており，この方向性は効果的でない。評価で示した通り，その分

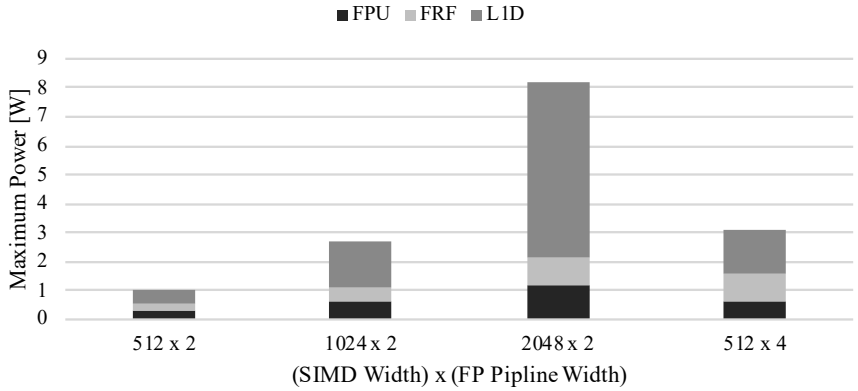


図 6.11: SIMD 幅，FP パイプライン幅のスケーリングに応じた電力の変化

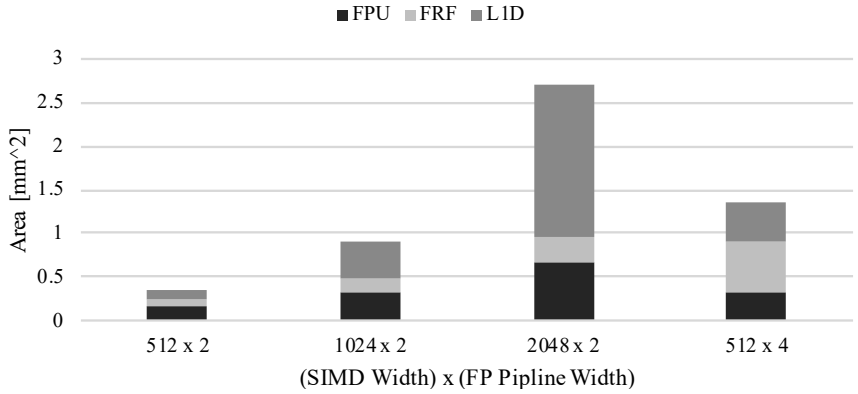


図 6.12: SIMD 幅，FP パイプライン幅のスケーリングに応じた面積の変化

のレジスタファイル，L1 データキャッシュ等 RAM に対する負担が大きくなり，電力・面積効率の向上は難しい。また，FP パイプライン数を増やした場合には，FP レジスタファイルのポート数の増大，アウト・オブ・オーダー実行資源の幅やエントリ数の増大を招くために得策ではない。RAM への負担を減らすには，逆に，コア当たりの FP 演算の幅を減らす方が得策であるが，その場合，演算器数の減少から，アウト・オブ・オーダー実行機構のオーバーヘッドが顕著になり，結果，制御機構をシンプルにしたイン・オーダー CPU や GPU 等のスループットに特化したアクセラレータに近いデザインが望ましい。しかし，現在の A64FX プロセッサのデザインであっても，コア面積の大部分を FPU や FP レジスタファイルが占めているため，この方向に進むことによって得られる性能向上も実際にはそれ程大きくなく，プログラマへの負担に見合った性能向上が得られない可能性がある。結果として現在のデザインを維持したまま，コア数を増やすのが最善策の一つとなる。

一方，1 要素の操作にかかるエネルギーについて着目すると，FPU が問題という訳ではなく，FPU にデータを送り込むための，メモリ階層全体での電力消費及び RAM 面積の肥大化・非効率化こそが問題であり，これはスケーリングが進むにつれて顕著化してきた問題である。これを解決するためには，データの情報量について着目しつつ，その転送量を SW/HW の両観点から削減することこそが眶となり，昨今の半精度や単精度の利用は一形態であると言える。一方で，それ以外の方向性についても，例えば，データ・キャッシュ圧縮等を利用するのは有力であり，例えば圧縮は HBM 側，及び解凍は演算直前で行う方式等も考えられ，さらには単純な操作は HBM 側

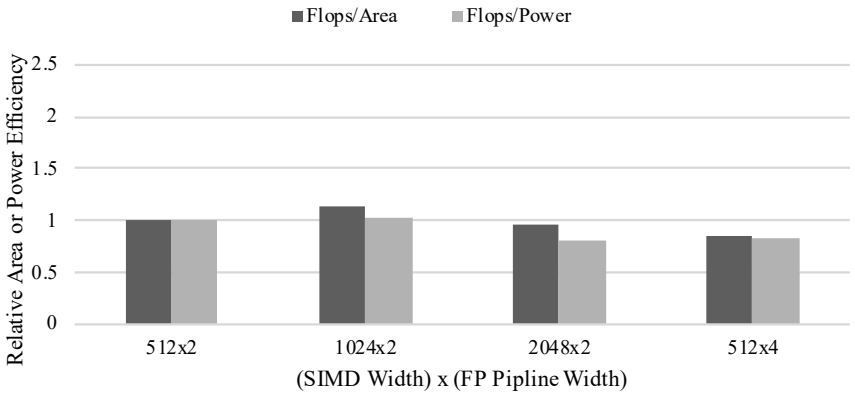


図 6.13: コア設定毎の面積対性能，電力対性能の比較

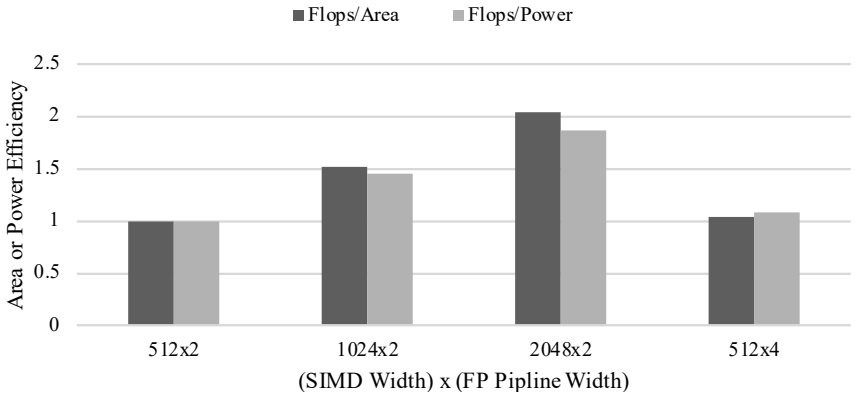


図 6.14: コア設定毎の面積対性能，電力対性能の比較 (Ideal)

で行う Processing-in-memory 等も将来長い目で見れば有力な手段であると言える。特に，圧縮に関しては，レイテンシを多少犠牲にしつつも，バンド幅や容量をメモリ階層全体で稼ぐことができ，演算律速・メモリ律速アプリケーションの両者に有効である。

以上をまとめると，A64FX プロセッサの様な HPC 向け SIMD プロセッサの場合，現在のコア・アーキテクチャを維持したまま，コア数をスケールさせるのが有力であり，さらに，半精度・単精度の利用や圧縮等によりデータ転送量を削減し，見かけのバンド幅やキャッシュ容量を増やすことによって，メモリシステムの負担を減らすことが重要であり，これを前提とした設計によって FPU の割合を増やすことで，更なる FP 演算性能の向上も見込める。

6.1.2.2 電力制御技術

スーパーコンピュータ「富岳」のようなポストペタスケール高性能計算システムは，消費電力がシステム設計や実効性能を制約する最大の要因となっている。地球規模の省エネ要求や現大型計算機センターの電力設備状況を鑑みると，将来的に 100MW 級の電力供給能力を持つ計算機センターを配することは不可能であり，エクサスケール級のシステムでも数 10MW と現在とほぼ同程度のオーダーの供給電力のもと，持続的な性能向上を達成していくことが求められる。このような背景のもと，スーパーコンピュータ「富岳」や，その後の HPC システムでは，電力制御技術が必須である。そこで，スーパーコンピュータ「富岳」を含む次世代 HPC システムのための電力

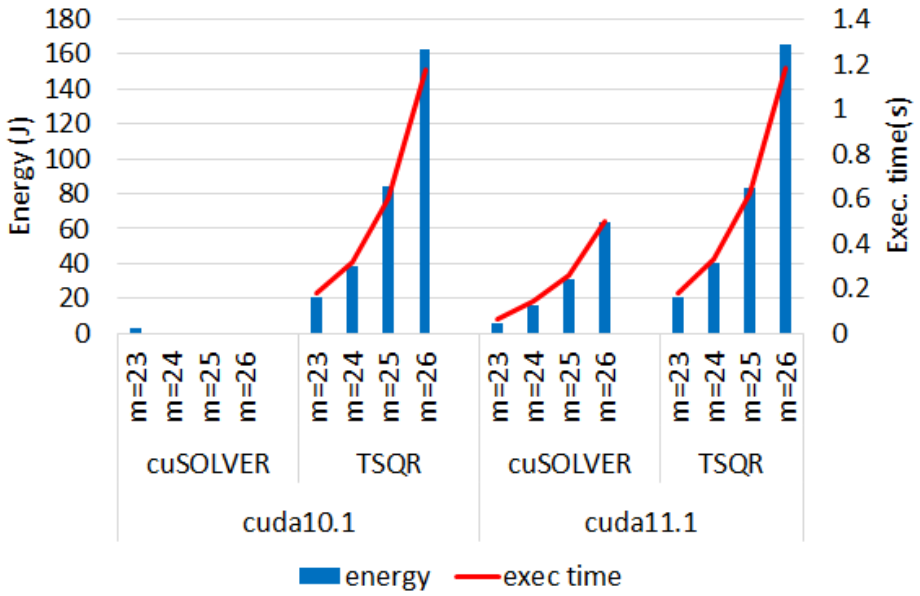


図 6.15: TSQR の実行時間と消費エネルギー

制御最適化の一環として，これまでオーバープロビジョニング環境におけるシステムレベルでの電力管理と，ジョブレベルでの電力最適化との連携に関する検討，低精度演算の積極的な利用による電力効率改善の評価，およびノード内の電力モード選択最適化に焦点を当てて研究を実施してきた。2020 年度は，特に低精度演算利用による電力効率改善の一環として，混合精度行列積演算回路を用いた電力評価 (2.2.1) と「富岳」を利用したノード内の電力モード選択最適化に関する評価 (2.2.2) を実施した。以下その詳細について述べる。

6.1.2.2.1 低精度演算の積極的利用による電力効率改善の検討

NVIDIA Volta アーキテクチャ世代より混合精度行列積演算回路である TensorCore が NVIDIA GPU に搭載されている。そこで，これらの TensorCore を用いて効率的に $m >> n$ な $m \times n$ 行列に対する QR 分解をおこなう Tall-Skinny QR(TSQR) を GPU 実装した TSQR for TensorCore の電力評価を行った。演算精度には float(32bit) を用い，TSQR for TensorCore と cuSOLVER による QR 分解の電力評価を行う。さらに，Cuda 環境として Cuda 10.1 と Cuda 11.1 のバージョンの異なる 2 つの環境を用いて評価を行う。Cuda 11.1 は cuSOLVER の実装が刷新しており，QR 分解関数は Cuda 10.1 と比較して大きく高速化されていることが知られている。

はじめに，消費エネルギーの評価結果を図 6.15 に示す。データサイズは $m = 23$ から $m = 26$ とした。Cuda 10.1 環境における cuSOLVER の $m = 24$ 以降は cuSOLVER の実装の制約によりメモリが確保できず実行不可であった。また，Cuda 11.1 環境の cuSOLVER は実装が刷新されており，TSQR よりも実行時間が短い結果となった。また，消費エネルギーにおいても実行時間の短縮により Cuda 11.1 の cuSOLVER がエネルギー消費も小さい結果となった。

次に平均消費電力の評価結果を図 6.16 に示す。Cuda 11.1 環境下では cuSOLVER が TSQR よりも平均消費電力が低い結果となった。また，データサイズが小さいほうが電力が小さいことがわかる。

最後に，コアやメモリの資源利用率を図 6.17 に示す。Cuda 11.1 環境下において cuSOLVER

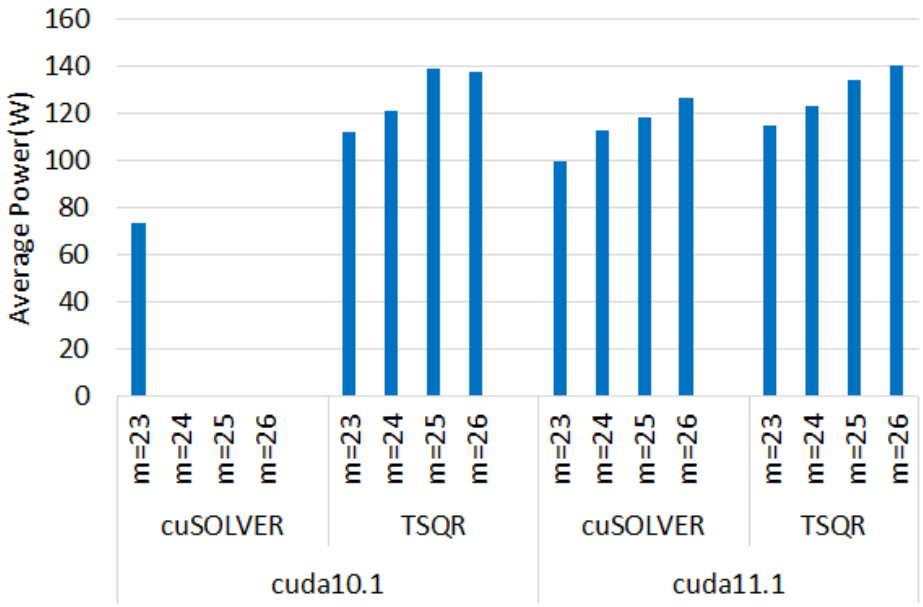


図 6.16: TSQR の平均消費電力

と TSQR の結果を比較すると TSQR はコア (gpu util.) の利用率が高く，cuSOLVER はメモリ (memory util.) の利用率が高いことがわかる。

6.1.2.2.2 ノード内電力モード選択最適化

電力制約型 HPC システムのスループットを最大化するには，如何に各ノードに電力予算を割り振るかというスケジューラによる最適化だけではなく，如何に与えられた電力を上手く使い切るかというノード単位での電力割り当て最適化も重要である。また，ノード内の各コンポーネントはいくつかの電力モードを持っていることが一般的であり，これらの中から最適なものを選択するための，フレームワーク，電力・性能モデリング，モード選択アルゴリズム等は重要である。その一環として，コンポーネント間のパワーシフティングに関する研究を実施しており，2020 年度は，国際会議 ISC'20 にて発表し，意見交換を行った (文献 [20])。

一方，A64FX プロセッサはいくつかの電力モードを持っており，これらを適用した際の電力削減効果や性能への影響を正しく理解することが，これを省電力運用等に適用していく上では重要である。これに関しても，コアリテンション，エコモード，ブーストモード等の組み合わせによる影響や，ノード間の電力ばらつきに関する評価を行った論文を，理研と共著で発表している (成果 [2, 7])。

6.1.3 数値計算ライブラリ

本研究では東京大学情報基盤センター・JCAHPC のスーパーコンピュータシステム（Oakforest-PACS）を使用した（参考文献 [27]）。

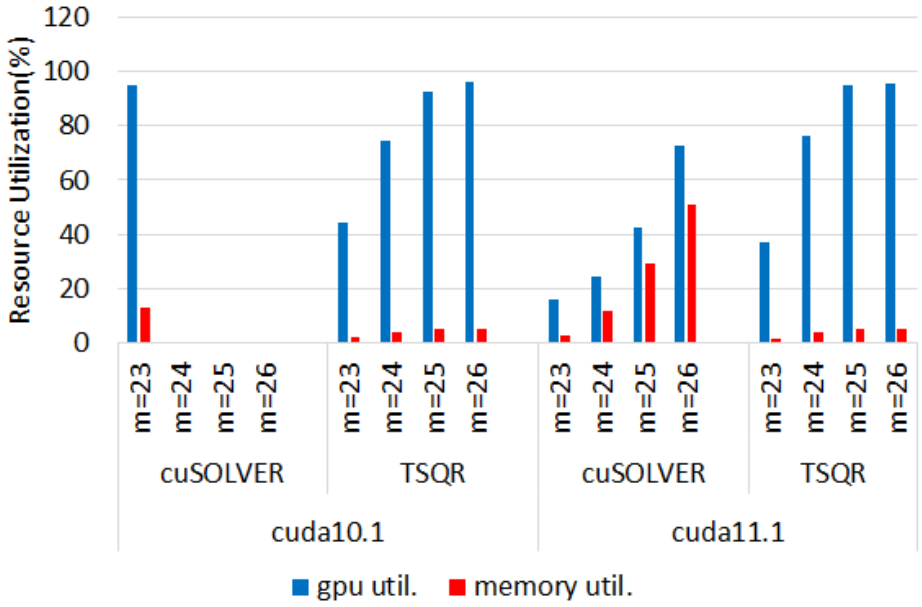


図 6.17: TSQR における資源利用率

6.1.3.1 メニィコアクラスタ向け並列多重格子法

並列多重格子法は，エクサスケールシステムにおける大規模問題向け数値解法として注目されている。著者による先行研究（参考文献 [28]）では，Sliced Ellpak-Itpack（ELL）（参考文献 [29]）に基づく新しい疎行列格納手法によりメモリアクセスの最適化を実現した。先行研究（参考文献 [28]）は，それまで疎行列ベクトル積に主に使用されていた Sliced ELL を前進後退代入を含むプロセス（IC(0) smoother）に導入した最初の事例の一つである。また著者等は，有限体積法による熱伝導方程式ソルバーから導かれる対称正定な疎行列を係数とする連立一次方程式を不完全コレスキー分解前処理付き共役勾配法（Preconditioned Conjugate Gradient Method by Incomplete Cholesky Factorization，ICCG 法）によってメニィコアクラスタで効率よく解くための研究として，GPU を含む様々な環境において，疎行列格納形式，演算精度等が，計算効率，消費電力，消費エネルギーに与える影響について検討を実施している他，精度保証，最適精度選択のための自動チューニングに関する研究を実施している [8] [10]。

SELL-C-σ （参考文献 [30]）は，ELL，Sliced-ELL を SIMD 演算向けに拡張した手法であり，GPU，Intel Xeon Phi 等において高い演算性能を発揮することが示されているが，主として疎行列ベクトル積に適用されており，前進後退代入を含むプロセス（ICCG 法）に適用したのは，著者等による先行研究（参考文献 [28]) が初めである．本研究では，SELL-C-σ による疎行列格納形式を，先行研究において使用した IC(0) を smoother とする多重格子法前処理付き共役勾配法（MGCG）法へ適用することを試み，予備的評価を実施した。

本研究では，三次元不均質多孔質媒体中の地下水流れ（図 6.18）を有限体積法によって解くプログラム *pGW3D-FVM*（参考文献 [28]）のポアソン方程式求解部の多重格子法前処理付き共役勾配法ソルバー（MGCG）に適用している。8 つのメッシュから 1 つのメッシュを生成する Octree 法を採用し，V-Cycle に基づく幾何学的多重格子法（Geometric Multigrid）を適用している。最も細かい格子レベル（level）を 1 とし，粗くなるにつれてレベルが増加する。最も粗いレベルにお

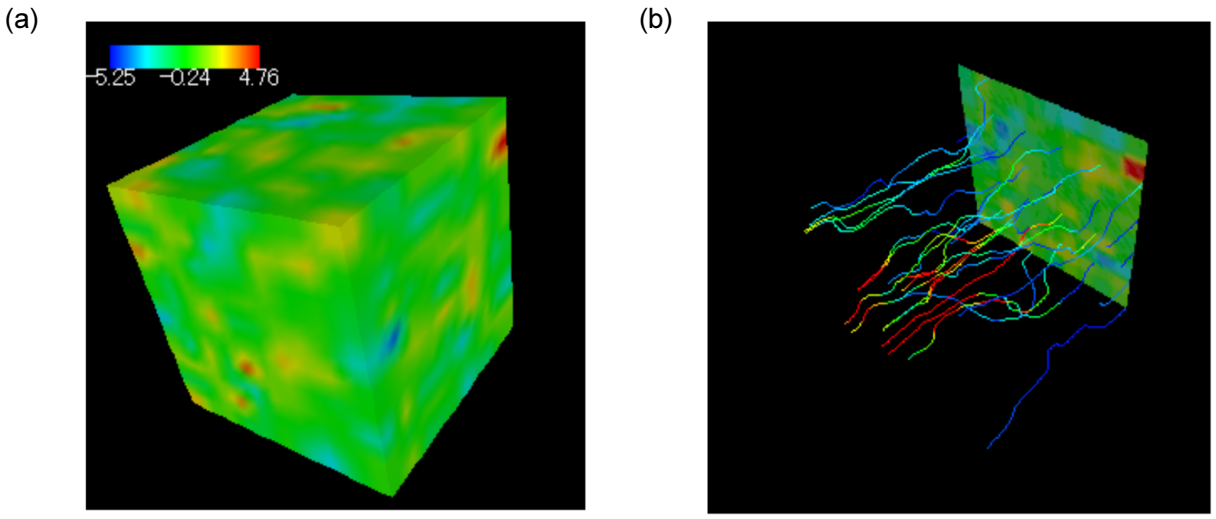


図 6.18: 不均質多孔質媒体中の三次元地下水流れ　(a) 透水係数分布，(b) 流線分布

ける自由度数は各 MPI プロセスにおいて 1 となる。ブロック Jacobi 型局所 IC（0）法を加法シュワルツ法と組み合わせた緩和演算子（smoothing operator）を各レベルにおいて適用している（参考文献 [28]）。OpenMP/MPI ハイブリッド型の並列プログラミングモデルが適用されている。

著者等による初期の手法は，最も粗い格子レベルで各 MPI プロセスにおける自由度数が 1 になると，全プロセスの情報を 1 コアに集約して更に多重格子法により求解を継続していた。1 コアに集約後の修正方程式のソルバーを *Coarse Grid Solver* と呼ぶ。プロセス数が増加すると Coarse Grid Solver の負荷が高くなり，また粗いレベルでは通信のオーバーヘッドが顕著となる。そこで，*Coarse Grid Aggregation*（CGA）法として，V-Cycle のレベルの小さい（従って細かい）段階で Coarse Grid Solver に移行する手法を提案した（図 6.19）。CGA 法によって収束の安定化，通信オーバーヘッドの削減が期待できるものの Coarse Grid Solver の扱う問題サイズは大きくなることから，Coarse Grid Solver について各 MPI プロセス上の全コアを使用し，OpenMP によってマルチスレッド化をすることによって，より高速に Coarse Grid Solver の計算を実施可能とした。

疎行列計算は間接参照を含むため memory-bound なプロセスである。従って疎行列演算において，演算性能と比較してメモリ転送性能の低い昨今の計算機の性能を引き出すことは困難である。係数行列の格納形式が性能に影響することは広く知られており，様々な手法が提案されている。Compressed Row Storage（CRS）形式は，図 6.20（a）に示すように疎行列の非零成分のみを記憶する方法である。*Ellpack-Itpack*（*ELL*）形式は各行における非零非対角成分数を最大非零非対角成分数に固定する方法であり（図 6.20（b））, 実際に非零非対角成分が存在しない部分は係数 =0 として計算する。CRS と比較して高いメモリアクセス効率が得られることが知られているが，計算量，必要記憶容量ともに増加する。

これまで，行列格納形式に関する研究は行列ベクトル積に関するものが主であったが，著者等は IC 法，ILU 法（Incomplete LU Factorization，非対称行列向けの前処理手法）等の前処理のようなデータ依存性を有するプロセスについて検討を実施している（参考文献 [28]）。ELL は，やや非効率的であり，特に不規則形状向けに，ELL 形式を拡張し，複数の配列を使用して，より効率的な計算を実施する手法として，Sliced-ELL 形式（参考文献 [29]) が提案されている（図 6.20（c））。pGW3D-FVM においては，Sliced-ELL の適用により，ELL と比較して，Oakleaf-FX 上で L2 キャッシュミス率が約 28%減少，計算性能が約 22% 向上することが報告されている（参考

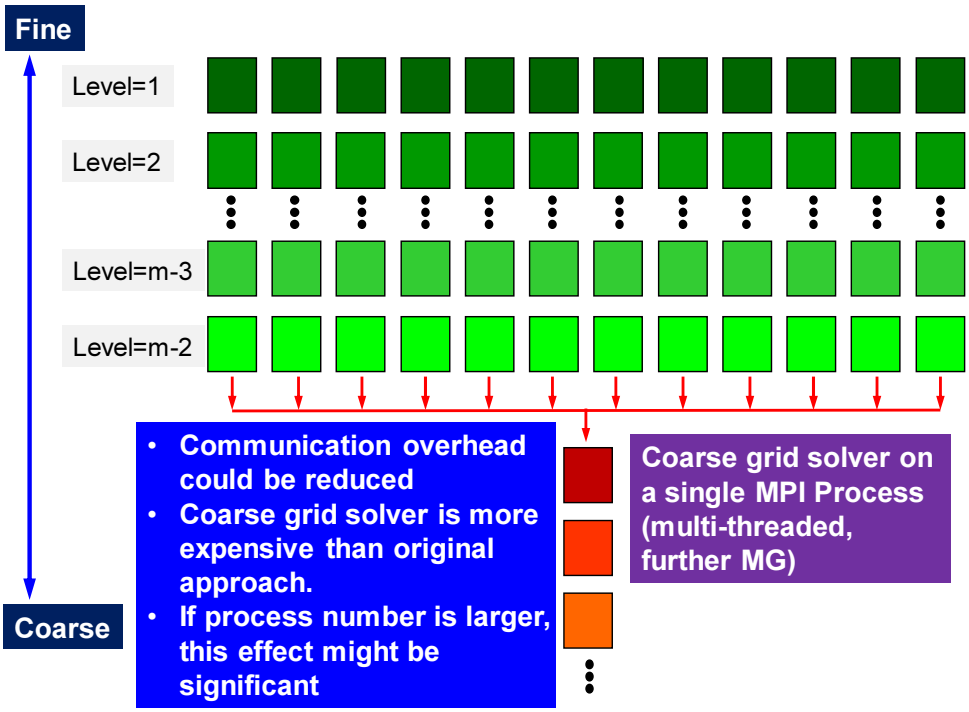


図 6.19: 並列多重格子法と Coarse Grid Solver（参考文献 [28]）

文献 [28])。

SELL-C-σ （参考文献 [30]) は，ELL 及び Sliced ELL（SELL）を SIMD プロセッサ向けに拡張した疎行列格納方式である（図 6.20（d））。C（chunk size）は計算を実行する単位であり，σ（sorting scope）は疎行列の非零非対角成分の分布によって決定されるパラメータである。Intel Xeon Phi のようなアーキテクチャでは，C を SIMD 幅（倍精度実数の場合 8（=512 bit/64））に設定するとベクトル化の効率が高まる。

本研究では，東京大学情報基盤センター・JCAHPC のスーパーコンピュータシステム（Oakforest-PACS，OFP）を使用した（参考文献 [27])。本研究では，MGCG の性能を OFP の最大 1,024 ノードまでを使用して実施した。*Flat MPI* および *Hybrid* $M \times N$ (HB M × N) 並列プログラミングモデルを適用した。ここで M は各 MPI プロセスにおける OpenMP スレッド数であり，N は各ノードにおける MPI プロセス数である。Flat/Quadrant Mode，MCDRAM のみを使用し，各ノード 68 コアのうち 64 コアをしようしている。Flat MPI, HB 2× 32, HB 4×16 and HB 8×8 が適用されている。

計測は 5 回実施し，計算時間最短の場合を採用している。各 MPI プロセスにおいてメッシュは各レベルにおいて CM-RCM(2)（参考文献 [28]）によって色づけ，並び替えされている。コア当たりの問題サイズは 64 × 32 × 32 (=65,536) に固定されている。性能評価（弱スケーリングは）は最大 1,024 ノード（65,536 コア），最大問題サイズ 4,294,967,296 メッシュまで実施されている。

本研究では，以下の 4 種類の疎行列格納手法について検討した。SCS（SELL-C-σ）については C=σ=8 である，SELL-8-8 を採用した。SCS-a はループが行方向（row-wise），SCS-b は列方向（column-wise）である（図 6.21 参照)。

- CRS (row-wise)

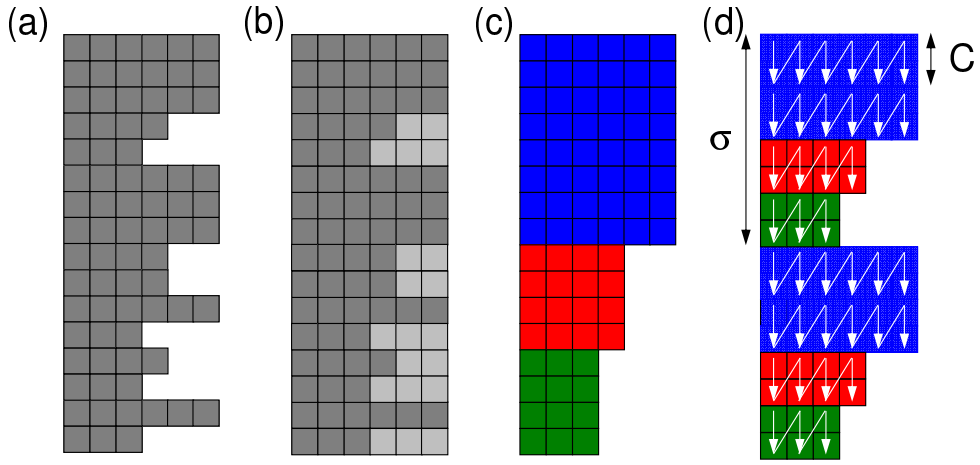


図 6.20: 様々な疎行列格納法（a）CRS, (b）ELL,（c）Sliced ELL, (d）SELL-C-σ（参考文献 [29] [30]）

- Sliced ELL (row-wise)
- SCS-a (SELL-8-8, row-wise)
- SCS-b (SELL-8-8, column-wise)

図 6.22 は 1 ノードにおける HB 4 × 16 の MGCG 計算結果である。*Lev.h* (h=1 (最も細かい（fine）レベル)〜4 (粗い（coarse）レベル)) は h 番目のレベルにおける Smoother（緩和演算子）の計算時間を示す。ここで *Rest* は通信時間，Coarse Grid Solver の計算時間，共役勾配法（CG）残り部分（行列ベクトル積，内積等）の計算時間を示している。MGCG ソルバーの CRS に対する計算性能改善率は，それぞれ 43.3% (Sliced ELL), 68.9% (SCS-a)，77.1% (SCS-b) である。最も計算時間の多い *Lev.1* については， 51.5% (Sliced ELL), 94.7% (SCS-a)，98.6% (SCS-b) である。

MGCG 法ソルバーの *SCS-a* の *Sliced ELL* に対する改善率は 28.5%であり，*SCS-b* によるそれは 31.1%である。全般的に SELL-C-σ による性能改善の効果は顕著である。図 6.23 は，各ケースにおける *SCS-b* の MGCG 法計算時間である。1,024 ノードでは Flat MPI による結果は得られていない。HB 8 × 8 においては 65,536 コア (1,024 ノード）まで良好なスケーラビリティを示しているが，ノード数が少ない場合は Flat MPI や，HB 2 × 32 のほうがむしろ計算性能は良い。CGA 法（図 6.19）では，Coarse Grid Solver は 1MPI プロセス上で動作し，問題サイズは総 MPI プロセス数に比例する。従って，Coarse Grid Solver のコストは，Flat MPI や HB 2 × 32 のように，のように MPI プロセス当たりのスレッド数が少ない場合に，ノード数が増加すると顕著に増加する（参考文献 [28]）Flat MPI や HB 2 × 32 がノード数（コア数）が増加した場合に性能が低下するのはこのような原因による。特に Flat MPI では，128 ノード（8,192 コア）以上では，性能劣化が顕著である。

図 6.24 は，*SCS-a* 及び *SCS-b* による，MGCG 法の性能の *Sliced ELL* に対する改善率を示す。8,192 コア（128 ノード）以下では，改善率は 20%を超えているが，ノード数が増加すると低下する。これは通信によるオーバーヘッドの増加とともに，Coarse Grid Solver のオーバーヘッドが増加するためである。現状では Coarse Grid Solver には *Sliced ELL* が使用されており，*SCS-a/SCS-b* を使用することにより性能が向上する可能性がある。

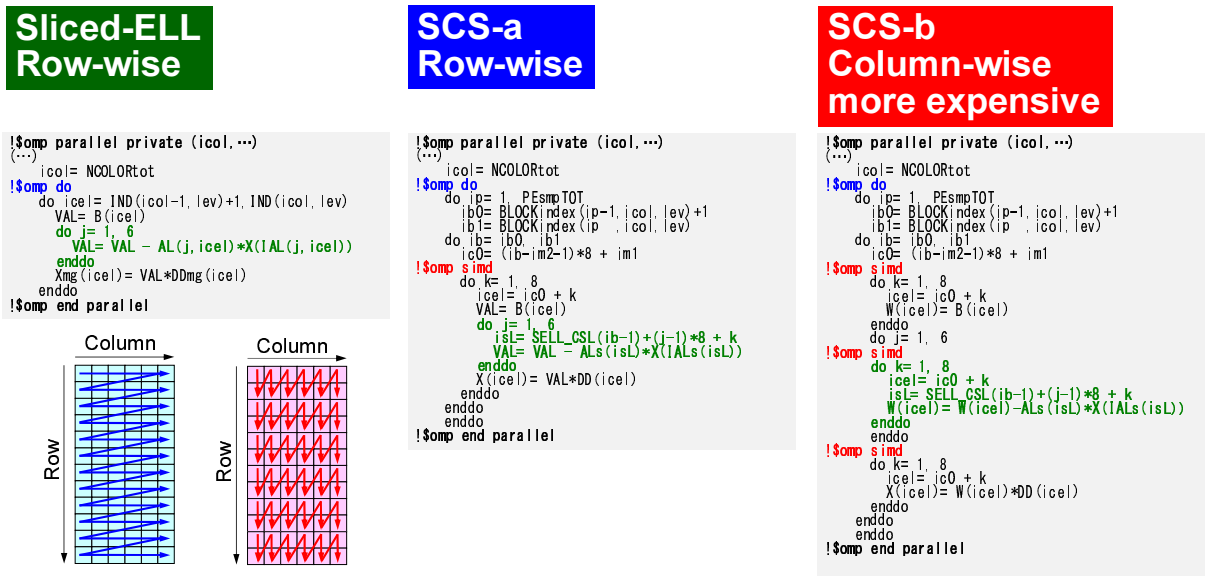


図 6.21: 各疎行列格納法における前進代入部分の記述

6.1.3.2 FX700 における低精度前処理演算

6.1.3.2.1 半精度前処理演算に関する予備的検討

著者等は，有限体積法によるポアソン方程式ソルバーから導かれる対称正定な疎行列を係数とする連立一次方程式を不完全コレスキー分解前処理付き共役勾配法（Preconditioned Conjugate Gradient Method by Incomplete Cholesky Factorization，ICCG 法）によってメニィコアクラスタで効率よく解くための研究を実施し，GPU を含む様々な環境での評価を実施している [8] [10]。近年は，低精度演算に着目し，倍精度・単精度演算の比較，消費電力，消費エネルギーの測定を実施し，条件の比較的良い問題では，単精度演算により倍精度の場合と比較して，50〜60%程度の計算時間，消費エネルギーで正しい計算を実施することができた。最新の研究 [8] では，実装方法，問題規模と低精度演算による性能改善の関係に注目し，5 種類の計算機環境下での検討を実施した。実装方法（疎行列格納形式），問題規模，計算機環境によって，低精度演算を使用することにより，計算時間，消費電力（W），消費エネルギー（J）への様々な効果があることがわかった。特に，問題規模が大きく，演算密度が場合には単精度演算による速度向上効果が顕著であることがわかった [8]。本研究では，特に半精度演算（FP16）の前処理への適用に着目した例を紹介する。

本研究では，図 6.25 に示す差分格子によってメッシュ分割された三次元領域において，不均質な熱伝導率（λ_1，λ_2）を有する場を想定して，定常熱伝導方程式を解くアプリケーション（P3D）を対象とし，導出された対称正定な疎行列を係数行列とする大規模連立一次方程式を不完全コレスキー分解前処理付き共役勾配法（ICCG）法によって解く。本研究では λ_1/λ_2 の値を変動させて実施した。形状は規則正しい差分格子であるが，プログラムの中では，一般性を持たせるために，有限体積法に基づき，非構造格子型のデータとして考慮する

P3D アプリケーションを対象とした先行研究ではプログラムの全変数を倍精度または単精度とした場合について比較を実施してきた。前処理付き反復法では，前処理部分のみ他の部分と演算精度を変える，混合精度演算がすでに行われている（参考文献 [31]）。本研究では，前処理部分（前進後退代入）について低精度演算を適用した例を紹介する。実施例は，表 6.5 および図 6.26 に示す通りである。FX700 では半精度（FP16）演算を手軽に実行できるが，半精度は有効桁が 3 桁程

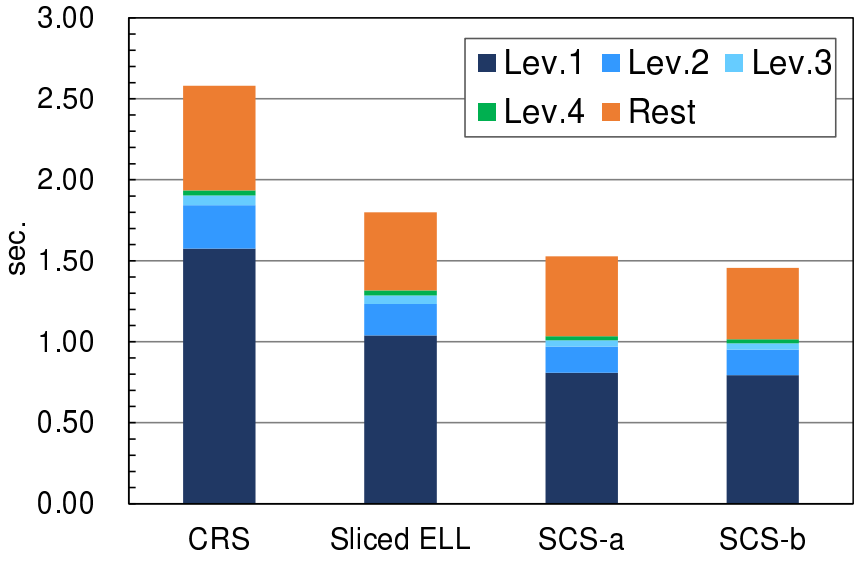


図 6.22: OFP 1 ノードによる MGCG 法計算時間，HB 4×16

度であるため，ベクトルを半精度とすると解が発散する．ここでは係数行列のみ半精度とし，ベクトルは単精度として扱った（図 6.26（b,c））。疎行列格納形式は CRS である。

図 6.27，図 6.28 は半精度演算を導入した場合の効果を示したものである。図 6.27 は図 6.25 における Bottom 点（原点）における計算結果を倍精度＋倍精度前処理（D-D）の場合の解に対する相対誤差として計算したものである。前処理に半精度演算を適用した D-H，S-H は λ_1/λ_2=10^6 の場合は収束解が得られていない他は，D-S，D-H は D-D と比較して計算誤差は無い。一方で S-S，S-H は λ_1/λ_2=10^3 以下の場合は誤差も 1%以下であるが，λ_1/λ_2 が増加すると急速に誤差が増加している。図 6.28 は，各ケースにおける反復回数であり，D-S，D-H では解が得られる範囲では，反復回数は D-D の場合と変化していないことがわかる。S-H は S-S とほぼ同様である。

表 6.5: 混合精度演算の効果

ケース名	反復法主要部	前処理	前処理部分ベクトル	備考
D-D	倍精度	倍精度	倍精度	
D-S	倍精度	単精度	単精度	図 6.26(a)
D-H	倍精度	半精度	単精度	図 6.26(b)
S-S	単精度	単精度	単精度	
S-H	単精度	半精度	単精度	図 6.26(c)

6.1.3.2.2 SELL-C-σ の低精度前処理演算への適用

次に、SELL-C-σ で実装した IC 前処理にに低精度演算の適用した場合の性能向上と、適した実装方法について述べる。なお、本評価に際して、低精度演算は IC 前処理部のみに適用しており、CG 部の演算精度は倍精度固定としている。

本評価で使用する前進代入の実装 (Base) を図 6.29 に示す。本実装では一行辺りの非零要素数は一定となっているため、図 6.29 に示すような実装となっている。この実装を基準として、本評価で

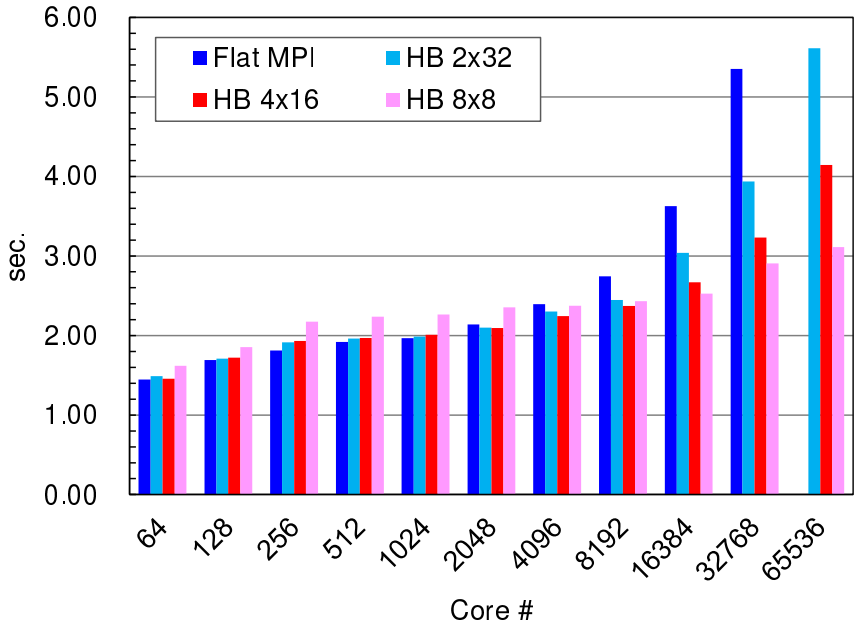


図 6.23: SCS-b 適用時の MGCG 法計算時間

はさらに二つの異なる実装を用意した。一つは最内ループ k を展開した実装 (opt1) であり、もう一つはループ k をループ l の外に移動させた実装 (opt2) である。Base 実装はこれまでに Oakforest-PACS などで実績があるが、SIMD 化対象のループ内が複雑な場合、FX700 では Floating operation wait が増加する傾向があることを fujitsu 製プロファイラ FAPP を使用して確認している。よって、FX700 のので十分な SIMD 化による効果をえるためには、ループ内の演算量が十分でり。また単純な状態が望ましい。opt1 は SIMD 化対象のループにおいて、base から演算量をそのままに演算単純にする意図があり、opt2 は演算を単純にする意図がある。

図 6.30〜図 6.34 に評価結果を示す。本評価で対象とした問題は $256^3 \simeq 16$M 自由度であり、CG 法の反復は相対残差が 10^{-7} 以下となったときに終了とした。図 6.30 は base 実装の Chunk size C に対する前処理部の計算時間を、図 6.31 と図 6.32 はそれぞれ opt1 と opt2 実装での計算時間を示している。本評価では様々なデータ型で FX700 に適した Chunk size (C) を確認するために、Chunk size を 8〜8192 の間で変化させており、表 6.6 は各データ型、実装での最適な Chunk size を示している。また、図 6.33 および図 6.34 はそれぞれのデータ型、実装の最適な Chunk size での前処理部の計算時間および ICCG 法全体の計算時間を示している。これらの図において、各データ型の組合せは”行列-ベクトル”で示しており、例えば、”double-single”であれば、前処理の行列のデータ型は倍精度、ベクトルのデータ型は単精度である。まず、各実装での Chunk size に対する傾向は、base および opt1 では 128〜512 が最適となっており、opt2 はそれよりもより大きな 512〜4096 が最適となっている。いずれの結果でも基本的にデータ型が低精度であるほど大きい Chunk size が最適という結果を確認している。低精度なデータ型ほど SIMD 幅を埋めるために多くのデータが必要であり、妥当な結果である。Oakforest-PACS での結果では、Chunk size は 8 が最適であり、また Chunk size に対する感度も低かったが、FX700 では最適な Chunk size が比較的大く、また Chunk size に対する感度の高さが確認できており、これが A64FX の特徴と推察する。次に、低精度化による効果だが、Base の実装では、行列に関してその効果を確認できるが、ベクトルの低精度化は逆に性能を悪化させる結果となった。対して、opt1 実装ではベクトルの低

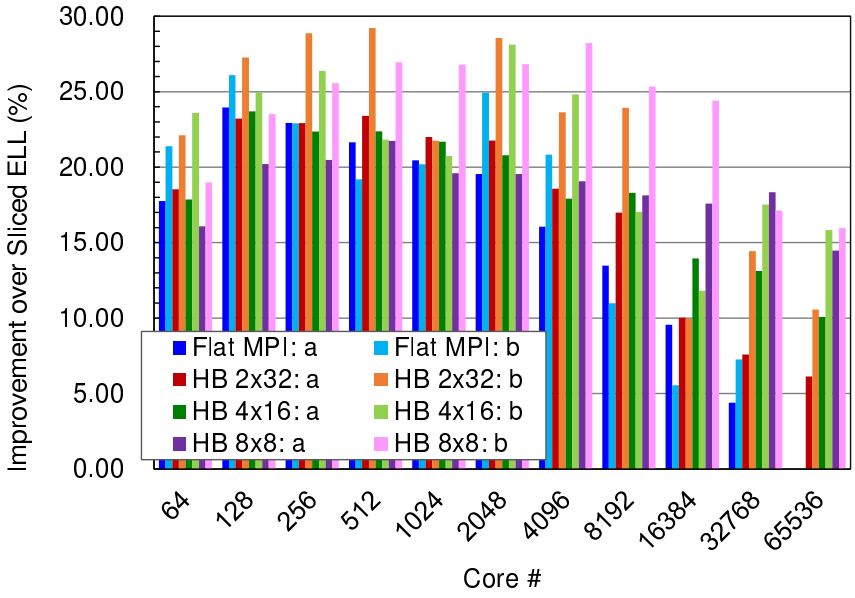


図 6.24: MGCG 法部分の SCS-a (:a)，SCS-b (:b) による *Sliced ELL* に対する改善率

精度化による効果が得られており、またどのデータ型でも Base 実装より高速になっている。opt2 も base より高い性能を示しているが、opt1 が最適な実装となった。

本評価で最も高速であった opt1 のデータ型の比較では、前処理部のみの比較で double-double に対して single-single は 18.9%、half-sngle は 30.6%の計算時間短縮を、全体ではそれぞれ 7.6%、10.9%の計算時間短縮を確認した。

6.1.4 システムソフトウェア

6.1.4.1 IHK/McKernel について

メニーコアアーキテクチャにおいて，並列プログラム実行のためのオーバーヘッドや OS ノイズを低減し，多数のコアを用いてスケーラブルな性能を実現する必要がある。そこで，R-CCS において，メニーコア向け OS カーネルとして McKernel が開発されている。McKernel では，実際に計算を行う Light Weight Kernel (LWK) と OS サービスを行う Linux OS 部分との資源を分離

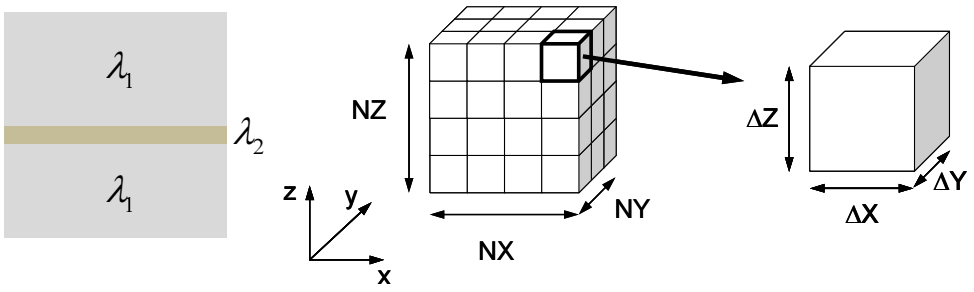


図 6.25: 三次元ポアソン方程式ソルバーの解析対象：差分格子の各メッシュは直方体（辺長さは ΔX,ΔY,ΔZ），X,Y,Z 各方向のメッシュ数は NX,NY,NZ

(a)

```
!$omp parallel do private(ip,i)
      do ip= 1, PEsmpTOT
      do i= SMPindex((ip-1)*NCOLORtot)+1, SMPindex(ip*NCOLORtot)
        Ws(i,Z)= Ws(i,R)
      enddo
      enddo

!$omp parallel private(ic,ip,ip1,i,WVALs,k)
      do ic= 1, NCOLORtot
!$omp do
        do ip= 1, PEsmpTOT
          ip1= (ip-1)*NCOLORtot + ic
        do i= SMPindex(ip1-1)+1, SMPindex(ip1)
          WVALs= Ws(i,Z)
          do k= indexL(i-1)+1, indexL(i)
            WVALs= WVALs - ALh(k) * Ws(itemL(k),Z)
          enddo
          Ws(i,Z)= WVALs * Wh(i,DD)
        enddo
        enddo
      enddo
!$omp end parallel
```

(b)

```
!$omp parallel do private(ip,i)
      do ip= 1, PEsmpTOT
      do i= SMPindex((ip-1)*NCOLORtot)+1, SMPindex(ip*NCOLORtot)
        Ws(I,Z)= W(I,R)
      enddo
      enddo

!$omp parallel private(ic,ip,ip1,I,WVALs,k)
      do ic= 1, NCOLORtot
!$omp do
        do ip= 1, PEsmpTOT
          ip1= (ip-1)*NCOLORtot + ic
        do i= SMPindex(ip1-1)+1, SMPindex(ip1)
          WVALs= Ws(i,Z)
          do k= indexL(i-1)+1, indexL(i)
            WVALs= WVALs - ALs(k) * Ws(itemL(k),Z)
          enddo
          Ws(i,Z)= WVALs * Ws(i,DD)
        enddo
        enddo
      enddo
!$omp end parallel

(Backward Substitution)

!$omp parallel do private(ip,i)
      do ip= 1, PEsmpTOT
      do i= SMPindex((ip-1)*NCOLORtot)+1, SMPindex(ip*NCOLORtot)
        W(I,Z)= Ws(I,Z)
      enddo
      enddo
```

(c)

```
!$omp parallel do private(ip,i)
      do ip= 1, PEsmpTOT
      do i= SMPindex((ip-1)*NCOLORtot)+1, SMPindex(ip*NCOLORtot)
        Ws(i,Z)= W(i,R)
      enddo
      enddo

!$omp parallel private(ic,ip,ip1,i,WVALs,k)
      do ic= 1, NCOLORtot
!$omp do
        do ip= 1, PEsmpTOT
          ip1= (ip-1)*NCOLORtot + ic
        do i= SMPindex(ip1-1)+1, SMPindex(ip1)
          WVALs= Ws(i,Z)
          do k= indexL(i-1)+1, indexL(i)
            WVALs= WVALs - ALh(k) * Ws(itemL(k),Z)
          enddo
          Ws(i,Z)= WVALs * Wh(i,DD)
        enddo
        enddo
      enddo
!$omp end parallel

(Backward Substitution)

!$omp parallel do private(ip,i)
      do ip= 1, PEsmpTOT
      do i= SMPindex((ip-1)*NCOLORtot)+1, SMPindex(ip*NCOLORtot)
        W(I,Z)= Ws(I,Z)
      enddo
      enddo
```

図 6.26: 混合精度演算（前進代入部分）(a) 倍精度＋単精度前処理 (D-S), (b) 倍精度＋半精度前処理 (D-H), (c) 単精度＋半精度前処理 (S-H), 緑字：倍精度，赤字：単精度，青字：半精度

し，LWK 上で実行されているアプリケーションからの OS サービスの一部を Linux OS 部分に依頼する機構を持つ。これにより，Linux API サポートを維持しつつ，OS ノイズの低減などによる性能の安定性や，オーバーヘッド低減によりスケーラビリティを改善するとともに，新しいハードウェアにも迅速に対応できる。

IHK/McKernel は，Linux のカーネルモジュールである IHK(Interface for Heterogeneous Kernel) と軽量カーネル McKernel から構成される。IHK により計算ノードは Linux カーネルおよび Linux カーネル上のデーモンが動作するコア群および物理メモリとアプリケーションプログラムと軽量カーネルが動作するコア群およびメモリに分割される。McKernel は IHK に依存した作りになっているが IHK は McKernel に対する依存性はなく，McKernel 以外に新規の軽量カーネルが開発できるよう設計されている。McKernel はアプリケーション性能に影響を及ぼすメモリ管理，プロセス・スレッド管理，同期，シグナル処理などを処理し，その他 Linux API は IHK 経由で Linux カーネル側に処理を移譲する (図 6.35)。

2019 年度に引き続き，2020 年度も最先端共同 HPC 基盤施設 (JCAHPC) が運用する Oakforest-PACS システム 24 ノード分の利用負担金を計上した。

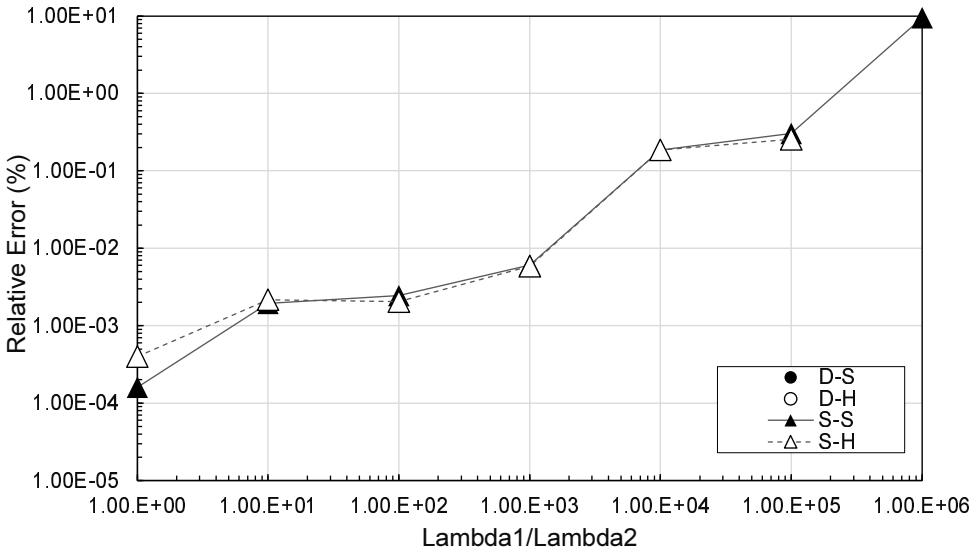


図 6.27: Bottom 点（図 6.25 の原点）での（D-D）の結果との相対誤差，D-S，D-H は解が得られる範囲では誤差は無い

6.1.4.2 並列多重格子法

2019 年度研究では，並列多重格子法について，Oakforest-PACS 上で IHK/McKernel の効果についての検証を実施し，ScalA19 Workshop （10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems）（SC19 併設）で査読付き論文を発表した（参考文献 [33]）。2019 年度は Flat MPI のみへの適用であったため，2020 年度は OpenMP/MPI ハイブリッド並列プログラミングモデルの場合における評価を実施した。図 6.36 は第 3 章で実施したケースに IHK/McKernel を適用した場合の性能改善率である。

2019 年度に得られた知見によれば，MPI プロセス当たりの問題規模が小さく，かつノード数が多い場合の方が，通信によるオーバーヘッドの効果が大きく，IHK/McKernel の効果もより大きいと考えられる。*h*CGA（参考文献 [28] [33]）による MGCG 法では OFP の 2,048 ノード（=131,072 コア）を使用した場合に，コア当たりの問題サイズが同じ場合の IHK/McKernel による性能改善率は 4%であった。図 6.36 は *SCS-b* を適用した場合の MGCG 法の IHK/McKernel による性能改善率を示している。HB 8 × 8 ではより性能改善は顕著で 65,536 cores (1,024 nodes). の場合に 11%である。現状では IHK/McKernel は 512 ノード（32,768 コア）以上では図 6.36 に示すように不安定であり，原因については調査中である。

6.1.5 今後の展望

6.1.5.1 プロセッサ・アーキテクチャ

本研究にて拡張したシミュレーション環境は，ポスト富岳を含む次世代 HPC システムのプロセッサの設計空間探索や電力・面積評価のために有効利用できる。また，本環境を用いた評価の結果，現在の 7nm から 3nm へとテクノロジーノードをスケーリングさせた場合には，コア数の増大による性能向上は見込めるものの，SIMD 幅，パイプライン幅の増大については，L1 データキャッシュ等の複雑さの増大により，性能向上が頭打ちとなる事を確認しており，これは今後の HPC 向

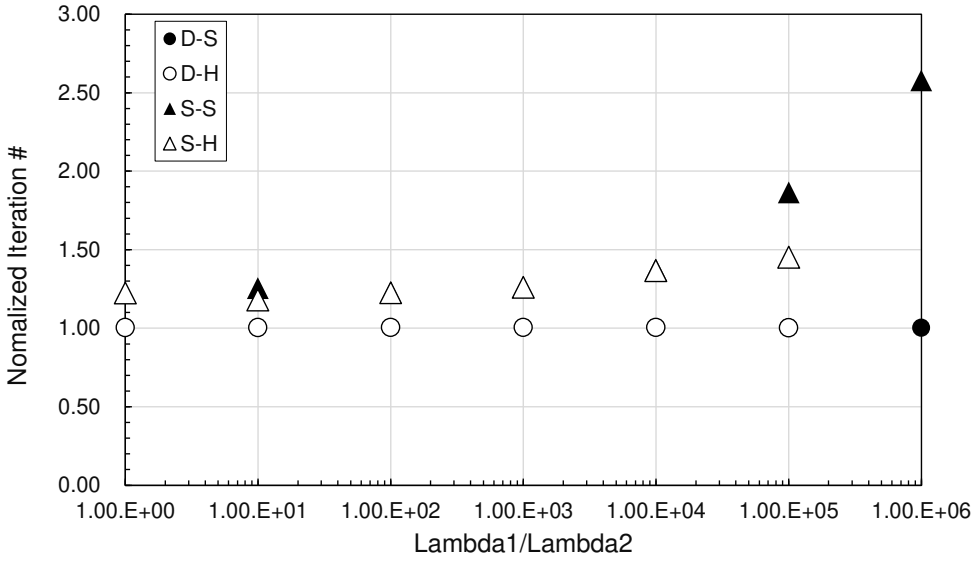


図 6.28: 各ケースにおける反復回数（D-D の反復回数で無次元化）

け SIMD プロセッサ設計の指針となるものである。また，今後のプロセッサ・アーキテクチャの研究開発の方向性として，SIMD コア数の増大だけではなく，メモリ階層全体でのデータ転送量の削減こそが重要であり，その為に低精度演算やデータ圧縮等の利用が重要となることを確認できた事も重要である。以上を考慮した上で，プロセッサ・アーキテクチャの最適化を検討する際には，本評価環境が有益となることは明白である。

6.1.5.2 電源制御技術

電力制御技術については，システムレベルとジョブレベルでの電力制御をそれぞれ検討し，いくつかの成果を得ることができたが，今後は両者を行うツール群を連携させ，システム全体として電力資源を有効利用しつつ高い性能・電力効率を達成できる手法，およびそのツールかを行う必要があると考えている。低精度・混合精度演算を積極的に利用するためのツールを組み合わせることも含め，今後も継続して研究開発を行っていく予定である。

また，ノード内の電力モード選択最適化についても，提案 (既発表文献 [20]) のフレームワーク，モデリング手法，アルゴリズム等の方法論に関しては一般性があり，富岳やその先の次世代 HPC システムへの応用展開も期待でき，加えて，理研と共同で行った，A64FX プロセッサの電力モード評価については (成果 [2, 7])，富岳やその他 A64FX ベースのシステムの省電力運用の際の参考になるものである。

6.1.5.3 数値計算ライブラリ

数値計算ライブラリについては，H 行列アルゴリズム，並列多重格子法，疎行列の低精度・混合精度演算について基礎的な研究開発を実施しており，2018 年度後半からは,「富岳」に向けた検討として，富岳シミュレータ，富岳実機，FX700 等を使用した検討を実施してきた。

並列多重格子法については，2020 年度は SELL-C-σ の実装を完了した。同手法は「富岳」においても有効な手法と考えられる。FX700 における検討も含めて継続して実施していく予定である。

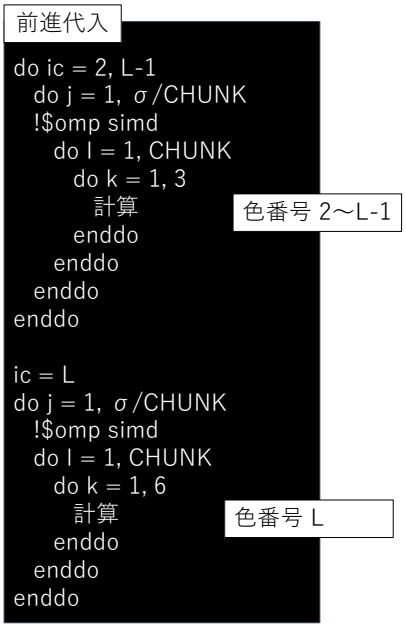


図 6.29: 前進代入のサンプルコード

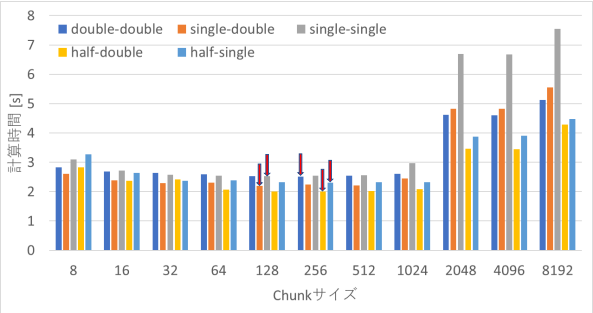


図 6.30: Base 実装の Chunk size に対する計算時間

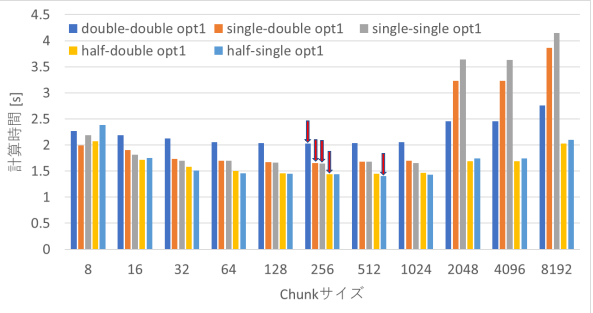


図 6.31: Opt1 実装の Chunk size に対する計算時間

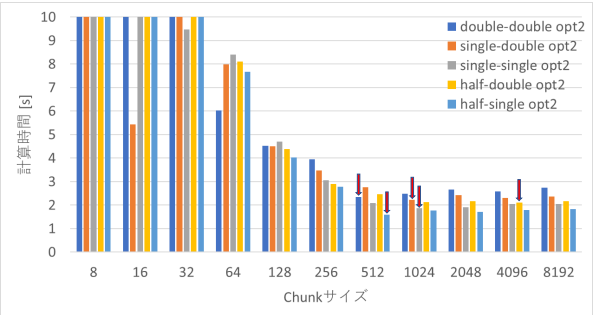


図 6.32: Opt2 実装の Chunk size に対する計算時間

表 6.6: 各データ型、実装での最適な Chunk size C

data-type	base	opt1	opt2
double-double	256	256	512
single-double	128	256	1024
single-single	128	256	1024
half-double	256	256	4096
half-single	256	512	512

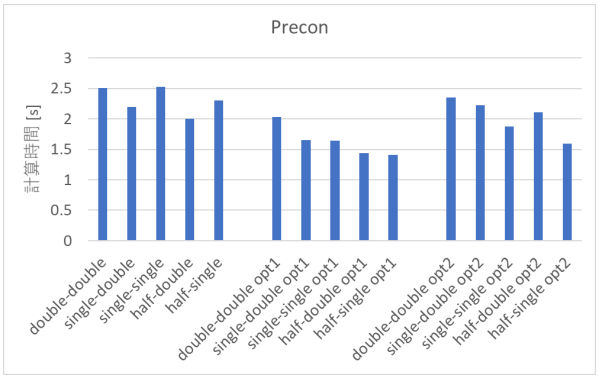


図 6.33: IC 前処理部の計算時間の比較

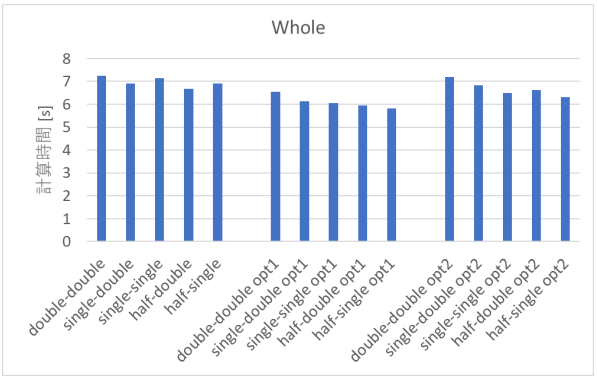


図 6.34: ICCG 法の全体の計算時間の比較

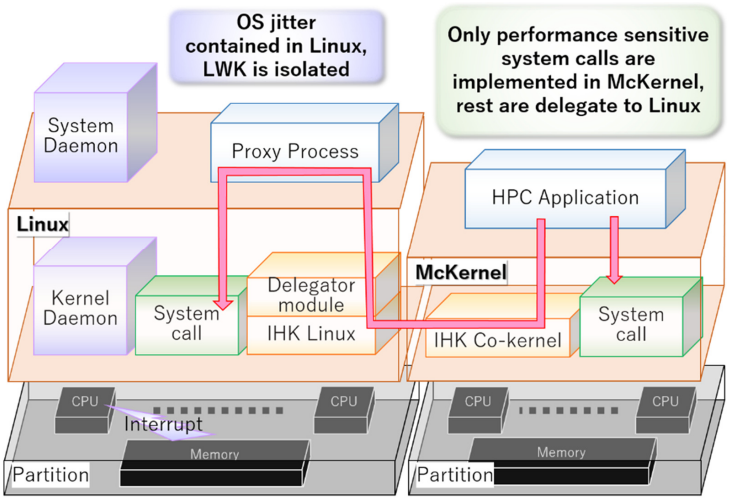


図 6.35: IHK/McKernel 概要・システムコール移譲機能概略 参考文献 [32]

疎行列の低精度・混合精度演算については，連立一次方程式を ICCG 法によってメニィコアクラスタで効率よく解くための研究を実施し，GPU を含む様々な環境での評価を実施してきた。特に近年は，低精度演算に着目し，倍精度・単精度演算の比較，消費電力，消費エネルギーの測定を実施し，条件の比較的良い問題では，単精度演算により倍精度の場合と比較して，50〜60%程度の計算時間，消費エネルギーで正しい計算を実施することができた。また，最新の研究 [8] では，実装方法，問題規模と低精度演算による性能改善の関係に注目し，5 種類の計算機環境下での検討を実施した。実装方法（疎行列格納形式），問題規模，計算機環境によって，低精度演算を使用することにより，計算時間，消費電力（W），消費エネルギー（J）への様々な効果があることがわかった。特に，問題規模が大きく，演算密度が場合には単精度演算による速度向上効果が顕著であることがわかっている [8]。2020 年度研究では，FX700 上での半精度演算（FP16）に関する評価を実施した。FP16 は有効桁が 3 桁程度であるため，安定性の問題があるが，前処理付き共役勾配法の前処理部分に使う限りでは，比較的条件の良い問題であれば，反復回数が増加することも無く，有効であることがわかった。

更に，SELL-C-σ で実装した IC 前処理への低精度演算の効果は最適な Chunk size、実装を用いることで実現できることを確認した。ただし、Chunk size の変更や最適化による効果は、Oakforest-PACS での評価と比較して、より効果が大きくまた好感度であることも確認した。よって、A64FX

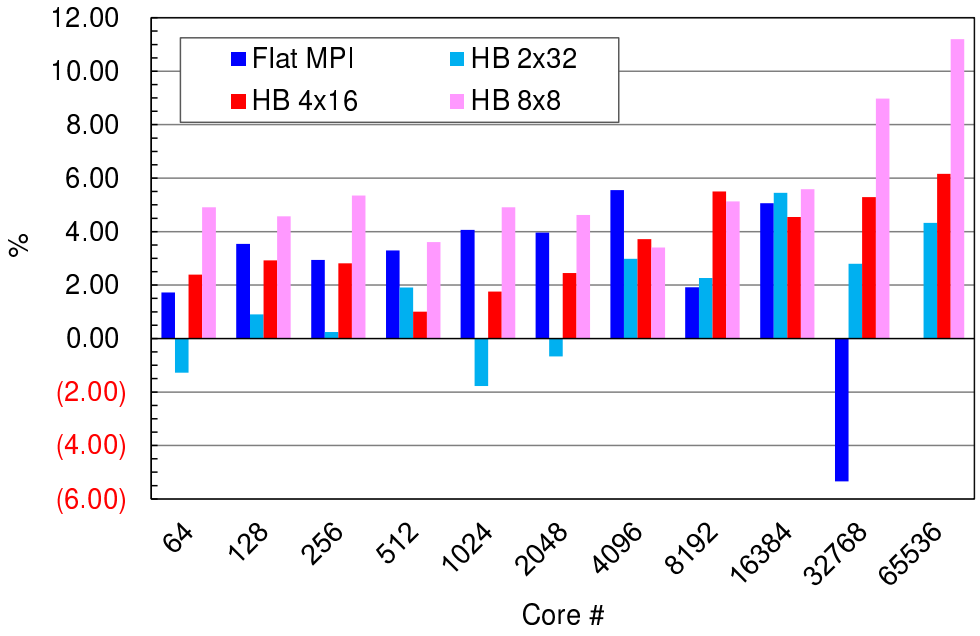


図 6.36: Performance Improvement of MGCG solver with SCS-b by IHK/McKernel

で高い性能を得るためには適したパラメータ、実装が必要であり、オートチューニングの整備などが重要になると考える。

FP16 の活用は「富岳」の計算性能を引き出す上で非常に重要な技術であり，FX700 における検討も含めて継続して実施していく予定である。

6.1.5.4 システムソフトウェア

システムソフトウェアに関しては，R-CCS においてメニーコア向け OS カーネルとして開発されてきた IHK/McKernel の Oakforest-PACS 上での整備，検証を継続して実施してきた。特に 2019 年度からは，並列多重格子法による大規模シミュレーションによる検証を実施し，通信オーバーヘッド及び OS ノイズ削減に大きな効果が得られることを確認した。Oakforest-PACS のようなメニィコアクラスタにおいて，IHK/McKernel のような軽量カーネルは，通信オーバーヘッドの削減，OS ノイズ低減によって並列アプリケーションの安定で効率的な実行には非常に効果的であり，特にノード数，MPI プロセス数が増加し，通信が支配的となる場合には，有効である。

IHK/McKernel は「富岳」にも搭載される予定であり，本研究において得られた知見をもとに「富岳」上での評価，更なる改良を継続して実施していく予定である。

6.2 筑波大学

「ポスト京の並列プログラミング環境およびネットワークに関する研究」

6.2.1 概要

本共同研究では理化学研究所におけるポスト京（「富岳」）の開発において、生産性が高く効率的な超並列プログラミング言語及びコンパイラに関する研究及びポスト京コンピュータの演算性能を加速する演算加速装置に関する研究を行った。以下、各研究テーマの概要について報告する。

6.2.2 タスク並列モデルに基づくマルチコア・アーキテクチャのプログラミングと性能評価に関する研究

本研究では、近年注目を高めている OpenMP タスク並列モデルに着目し、タスク処理系ごとの性能や特性の評価を行った。評価においては、複数の CPU アーキテクチャを利用し、また複数のタスク処理系を利用した。利用するタスク処理系においては OS スレッドを用いる処理系とユーザスレッドを用いる処理系の 2 つを利用し、それぞれの優位性についての評価も行った。

6.2.2.1 対象のアーキテクチャと OpenMP 処理系

Intel 社製 x86_64 CPU（Broadwell、Skylake、Xeon Phi KNL）、AMD 社製 x86_64 CPU（EPYC Naples）、Marvell 社製 ARMv8 CPU（ThunderX2）、富士通社製 ARM CPU（A64FX）を使用する。Intel CPU においては、カーネルスレッドを用いたタスク処理系を提供する GNU OpenMP、Intel OpenMP、LLVM OpenMP、またユーザスレッドライブラを用いたタスク処理系を提供する BOLT の 4 つの評価を行う。BOLT とは、LLVM OpenMP をベースとしユーザスレッドライブラリ Argobots を用いるように書き換えられたものである。AMD CPU、Marvel CPU においては GNU OpenMP、LLVM OpenMP、BOLT の 3 つを利用する。また、富士通 CPU においては、富士通コンパイラが提供する 2 つのモード（trad モード、clang モード）を利用する。BOLT は A64FX CPU において正常に動作しないため現時点では計測できていない。

評価においては、タスク並列モデルを用いて記述される代表的なプログラムであるブロックコレスキー分解を用いる。ブロックコレスキー分解は、portf、trsm、syrk、gemm の 4 つの BLAS ルーチンで構成される。問題サイズは、8192x8192、16384x16384 の 2 つを利用し、それぞれ 32x32 のブロックに分割する。A64FX の評価においては 4096x4096、8192x8192 を利用する。演算精度は単精度浮動小数点演算とする。

6.2.2.2 評価

はじめに Broadwell CPU について示す。図 1 は Xeon E5-2660v4（2.0GHz、14 コア）を 2 ソケット搭載したシステムを用いて評価を行った結果である。14 スレッドまでの並列実行においては 1 ソケットのみ利用する。OS スレッドを利用する GNU OpenMP（gcc と表記）、LLVM OpenMP（llvm と表記）、Intel OpenMP（intel と表記）はどちらの問題サイズにおいてもコア数に応じて性能がよくスケールする一方、BOLT は問題サイズが小さい場合は特に性能がスケールしていな

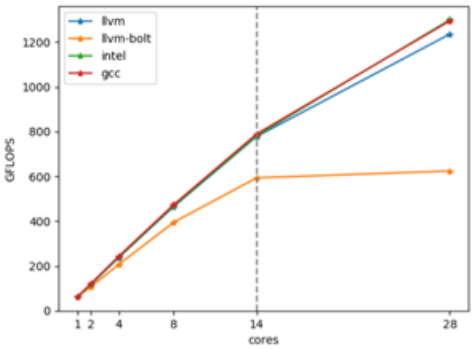


Figure 1: 8192-256 on Broadwell

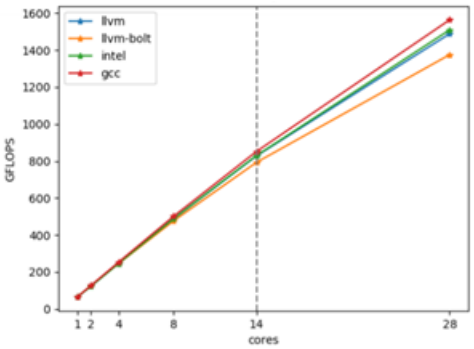


Figure 2: 16384-512 on Broadwell

図 6.37: Xeon Broadwell CPU を用いた評価

いことが確認できる。どちらにおいても、GNU OpenMP の性能が最もよいこと、OS スレッドを用いる処理系のほうが高い性能を示すことが確認できる。

1 ソケットのみを用いてそれぞれの処理系を用いて実行した際のタスクの実行の流れを可視化したものを図 2 に示す。白は potrf、緑は trsm、青は syrk、黄色は gemm のタスクである。OS スレッドを用いる処理系と比較し、BOLT は黒色の箇所、つまり計算タスク以外の処理（タスク管理等）の割合が高いことが確認できる。また、GNU OpenMP においては trsm のタスクが同時刻に複数コアで処理されている一方、他の 3 つにおいてははじめの箇所を除きバラバラに実行されていることが確認できる。これは GNU OpenMP は FIFO 形式のキューを用いてタスクを管理する一方、他は LIFO 形式のキューを用いて管理しているためであると考えられる。

次に Xeon Skylake を用いた評価を図 3 に示す。評価では、筑波大学計算科学研究センターが運用する Cygnus を利用する。Cygnus には、各ノードにおいて Xeon Gold 6126（2.60GHz、12 コア）が 2 ソケット搭載されている。12 スレッドまでの並列実行では 1 ソケットのみを用いている。結果においては、Broadwell を用いた結果と傾向が類似しており、特に問題サイズが 16384x16384 の場合において GNU OpenMP が最もよい性能を示すこと、また BOLT については他の 3 つと比べてスケールが悪く、性能も最も低いことが確認できる。

次に Xeon Phi Knights Landing を用いた評価を図 4 に示す。評価では、JCAHPC が運用する Oakforest-PACS を利用する。Oakforest-PACS には、各ノードに Xeon Phi 7250（1.40GHz、68 コア）が 1 ソケット搭載されている。問題サイズが 8192x8192 の場合は GNU OpenMP が最も性能を示す一方、64 コアで実行した際に性能が低下していることが確認できる。16384x16384 の問題サイズにおいては、Intel OpenMP および LLVM OpenMP がコア数に応じてよくスケールしている一方、GCC および BOLT は 32 コアから 64 コアにかけての性能向上率が下がっていることが確認できる。この結果より、Xeon Phi においては、OS スレッドを用いる処理系に着目した場合は FIFO 形式のキューを用いる GNU OpenMP よりも LIFO 形式のキューを用いる LLVM OpenMP や Intel OpenMP がより高い性能を示すことが確認できる。一方 BOLT においては、実装のベースとして利用している LLVM と比較し低い性能を示しており、利用する Argobots のタスク処理性能に何かしらの原因があると推測される。

次に AMD Naples を用いた場合の性能を図 5 に示す。ここでは、EPYC 7551（2.00GHz、32 コア）を 1 ソケット搭載したシステムを利用する。EPYC 7551 はこれまでの Intel 社製 CPU とは異なり、チップレット構成となっており、4 つの CCD（CPU Compute Die）にそれぞれ 8 コアの

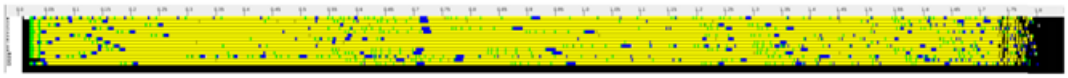


Figure 3: LLVM OpenMP on Broadwell

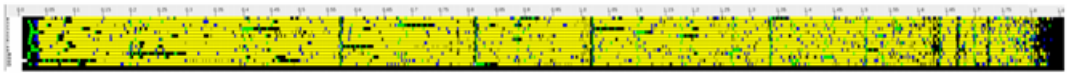


Figure 4: LLVM-BOLT OpenMP on Broadwell

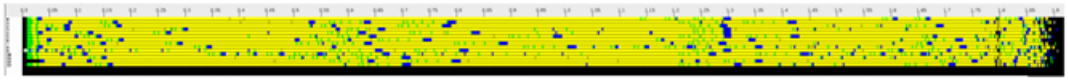


Figure 5: Intel OpenMP on Broadwell

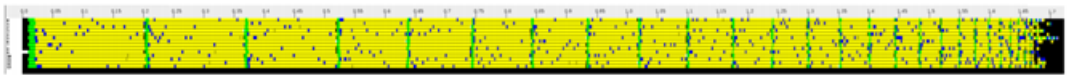


Figure 6: GCC OpenMP on Broadwell

図 6.38: タスク実行の流れ

CPU が搭載されている。また、それぞれの CCD は 1 つの NUMA として認識される。結果を確認すると、GNU OpenMP が最も性能がよく複数の NUMA を利用した場合も性能が十分スケールしていること、また LLVM と比較し BOLT がより高い性能を示していることが確認できる。BOLT が LLVM より高い性能を示す要因は解明できていないが、タスク並列処理系においてユーザスレッドライブラリが必ずしも優位性が低いわけではないということが推測される。LLVM においては、特に問題サイズが大きい場合において、16 コアから 32 コアにかけて性能のスケールが下がっていることが確認できる。

次に Marvel ThunderX2 の評価を図 6 に示す。ここでは、ThunderX2 CN9980（2.2GHz、32 コア）を 2 ソケット搭載したシステムを利用する。32 スレッドまでの並列実行においては 1 ソケットのみを利用する。Xeon Phi を除くこれまでの結果と同様 GNU　OpenMP が最も高い性能を示していることが確認できる。また、Thunder X2 においては、LLVM-BOLT を用いる場合の性能が最も悪く、特にソケットをまたいた際の性能が低下していることが確認できる。

最後に富士通 A64FX を用いた評価を図 7 に示す。この評価では理化学研究所に設置された FX700 システムを利用する。該当システムには、48 コアが 2.0GHz で動作する A64FX CPU が搭載される。FX700 に搭載される A64FX は 4 つの CMG（Compute Memory Group）から構成され、それぞれ 12 コアと 8GB の HBM メモリを持つ。富士通コンパイラが提供する 2 つのモードについて、trad モードにおいては富士通の OpenMP ランタイムが、clang モードにおいては LLVM OpenMP をベースとした OpenMP ランタイムが利用される。なお、本評価ではこれまでと異なり、スレッドアフィニティとして compact および scatter の 2 つを利用する。compact は可能な限り同じ NUMA ノード内にスレッドを配置するもので、これまでの計測で利用しているものと同等である。Scatter は可能な限り複数の NUMA ノードを利用するものである。結果では、4096x4096 においては clang モードの 2 つがほぼ同一の性能を示しており、また trad モードより高い性能を示していることが確認できる。8192x8192 においてはわずかながら、特に 12 コア、24 コアにお

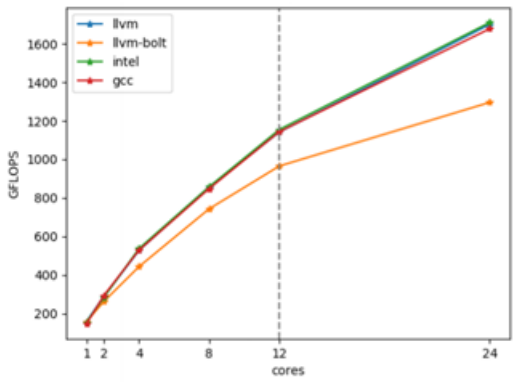


Figure 7: 8192-256 on Skylake

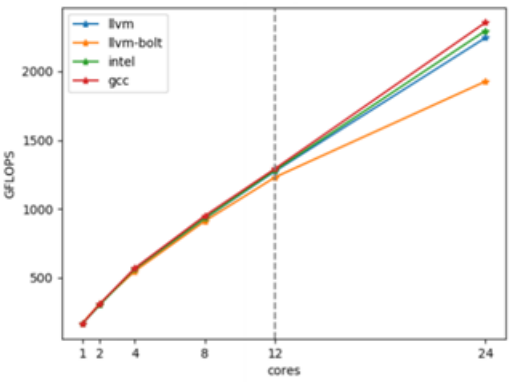


Figure 8: 16384-512 on Skylake

図 6.39: Xeon Skylake CPU を用いた評価

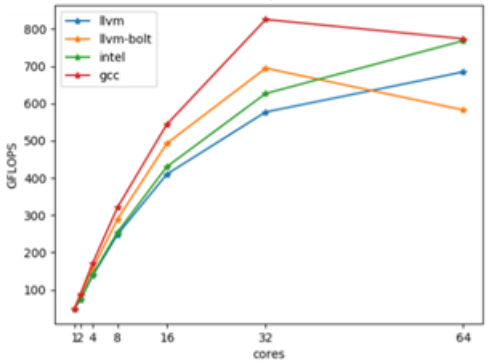


Figure 13: 8192-256 on KNL

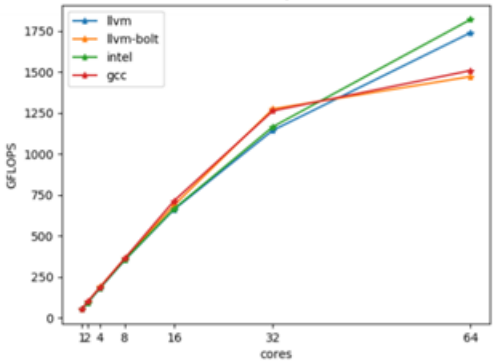


Figure 14: 16384-512 on KNL

図 6.40: Xeon Phi KNL を用いた評価

いて、clang モードのアフィニティを scatter と設定したものが最も高い性能を示している。これはより多くの CMG を利用することでメモリバンド幅が向上したためであると推測される。今後、BOLT が A64FX で正常に動作しない理由を調査・修正し、比較を行う予定である。

6.2.2.3 まとめ

本研究では、4 つの x86_64 アーキテクチャCPU、2 つの ARM アーキテクチャCPU を対象として、複数の OpenMP 処理系の性能比較を行った。A64FX を除いた結果として、Xeon Phi 以外のアーキテクチャにおいては GNU OpenMP がもっとも高い性能を示すことを確認した。GNU OpenMP は、用いた OpenMP 処理系の中で唯一 FIFO 形式のタスクキューを用いている。ただし現段階において、対象の計算はブロックコレスキー分解の 1 つのみであるため、FIFO キューを用いた処理系の優位性については他の複数の計算を用いた上で総合的に評価を行う必要がある。また、OS スレッドを用いる処理系と比較し、ユーザスレッド Argobots を用いる BOLT は、タスク

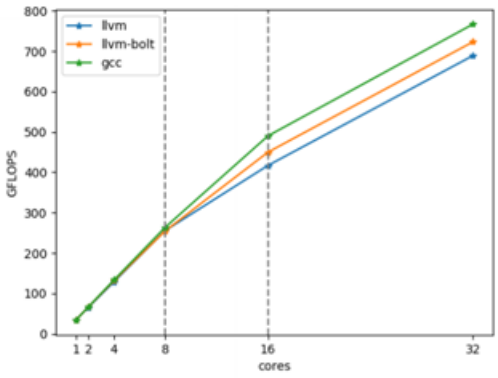


Figure 19: 8192-256 on EPYC

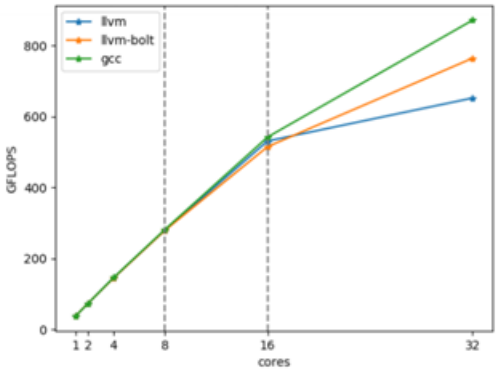


Figure 20: 16384-512 on EPYC

図 6.41: EPYC Naples を用いた評価

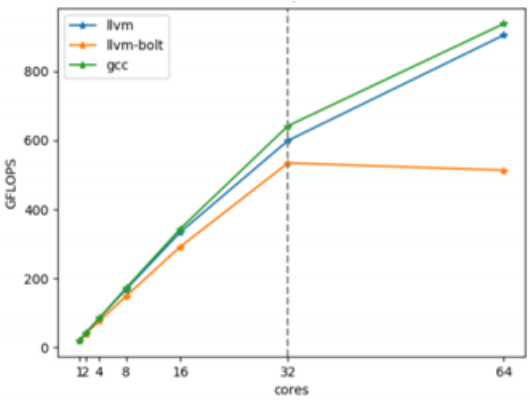


Figure 24: 8192-256 on ThunderX2

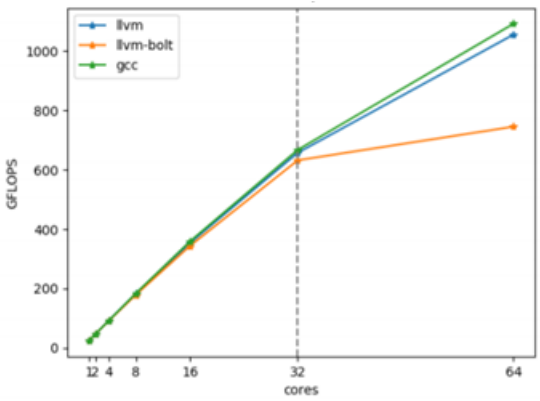


Figure 25: 16384-512 on ThunderX2

図 6.42: ThunderX2 を用いた評価

並列処理においては性能の優位性が低いことが確認された。現段階においては要因を解明できておらず、また Argobots そのものにおいては OS スレッドと比較して優位であることが報告されているため、さらなる調査を行う必要がある。

6.2.3 FPGA と GPU を混載した複合型演算加速システムにおける OpenACC プログラミング環境に関する研究

本研究では、Cooperative Acceleration by Multi-device Programming (CAMP) を提案し、GPU と FPGA を両方用いたプログラムを OpenACC により統一して記述可能な環境について研究開発を行う。まず、本研究で実装している MHOAT について説明し、本年度より追加実装中の機能、及び新たに行った評価について記す。

GPU は演算性能に対する電力効率が高いため、演算加速装置として多く用いられている。しかし、GPU の不得手な処理は依然として存在しており、性能向上の妨げとなっている。そこで、処理

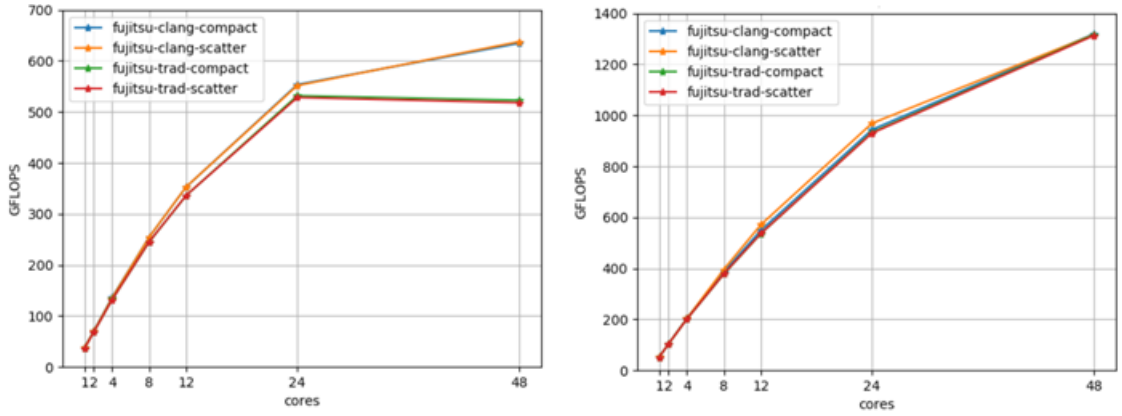


図 6.43: A64FX を用いた評価

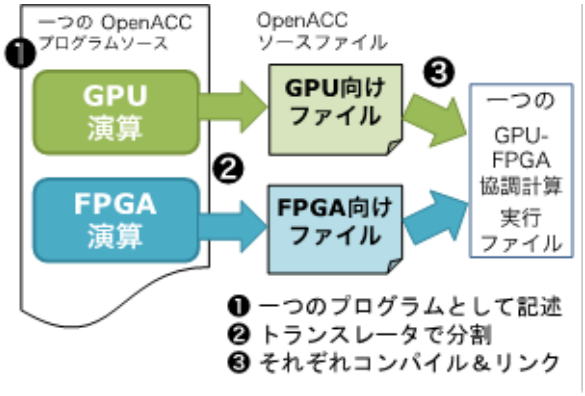


図 6.44: プログラミング環境の概要

内容に対して任意の論理回路がプログラム可能な FPGA に、GPU が不得手とする処理を行う回路を実装し、GPU と FPGA を協調的に演算させることにより、さらなる高速化を実現しようと我々は試みている。ところが、GPU と FPGA を効果的に用いるには、それぞれ CUDA と OpenCL といった異なるプログラミング言語で開発しなければならず、アプリケーション開発者に多大な負担を強いる。よって、本研究では GPU と FPGA を統一的に記述できるプログラミング環境の開発を行った。

CUDA や OpenCL はハードウェアに関する指定をいくつもしなければならず、単体でもプログラミングコストが高い。一方で、OpenACC では抽象化されたコードを OpenMP のようにディレクティブ形式で記述することにより、より簡潔なコードで演算加速プログラミングが可能である。よって、本研究では統一的な API として OpenACC を用いた。

6.2.3.1 MHOAT

本研究において開発しているプログラミング環境の概要を図 8 に示す。

まず、OpenACC C により、CPU 及び GPU と FPGA で行う演算を記述したプログラムを作成する（① ）。次に、メタコンパイラによりこのプログラムを解析し、GPU 向けと FPGA 向けの 2 つのバックエンドコンパイラへの入力用ファイルへそれぞれ分割する（② ）。最後にこれらのファ

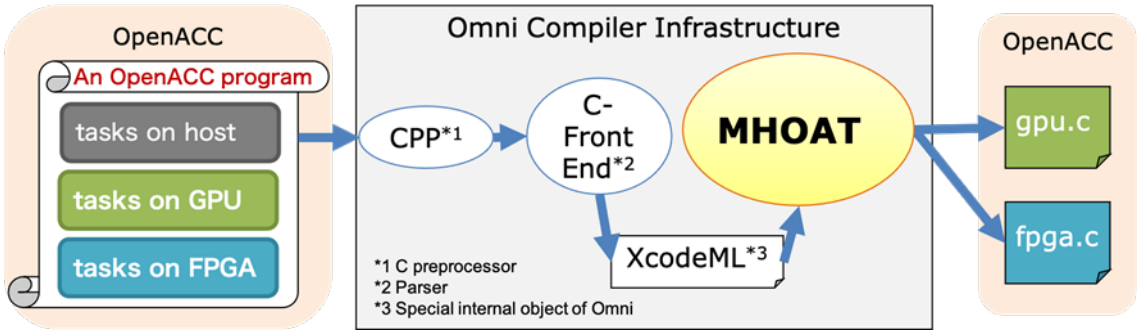


図 6.45: MHOAT の処理の流れ

イルをバックエンドコンパイルし、2 つのオブジェクトファイルをリンクすることで、一つの実行ファイルを生成する（③ ）。ここで、バックエンドコンパイルには、GPU 向けには PGI Compiler、FPGA 向けには米オークリッジ国立研究所の開発している OpenARC を用いる。2 つの異なるコンパイラが必要な理由は、両方のアクセラレータを使用するコードを生成できる OpenACC コンパイラーが現状存在しないためである。また、FPGA 向けの OpenACC プログラムに対応しているコンパイラは、現在 OpenARC のみである。② でファイルを分割しなければならないのはこのためであり、本研究ではここで用いるメタコンパイラの開発に理研と筑波大学が共同研究している Omni Compiler Infrastructure を用いた。

本研究で開発したメタコンパイラは MHOAT（Multi-Hybrid OpenACC Translator）と呼ぶ。処理の流れを図 9 に示す。まず、Omni Compiler 共通の処理として、C Preprocessor（CPP）でヘッダーファイルのインクルードやマクロの展開が行われる。次に C-FrontEnd により C から XcodeML と呼ばれる中間コード形式の XML ファイルに変換される。このファイルを MHOAT で読み込み、オフロード先を指定するディレクティブを解析して 2 つのファイルを出力する。なお、CPU で行う処理については、バックエンドコンパイルの都合上、GPU 向けコンパイル用の出力ファイルに書き込まれる。

オフロード先を指定するディレクティブは現状の OpenACC には存在しないため、本研究では以下の OpenACC の拡張ディレクティブを定義した。

```
#pragma accomn ondevice(DEVICE)
```

“ accomn ”は通常の OpenACC ディレクティブを示す“ acc ”に Omni Compiler 独自の拡張であることを示すために“ omn ”を追記したものである。DEVICE に当たる値は予約語のような形で、GPU か FPGA どちらか一方を記述する。図 10 に使用例を示す。この拡張ディレクティブは、現段階では main 関数以外の関数定義の先頭に一つだけ記述するようになっている。また、OpenACC ではデータを転送するディレクティブも存在するが、転送先のデバイスを指定することはできない。よって、データの転送はこの拡張ディレクティブの適用範囲以下に記述する必要があり、デバイス毎に変数のスコープが異なってしまう。しかし、ホストのメモリを介すことで、デバイス同士のデータ転送自体は可能なことから、本研究では最初の実装としてこの拡張ディレクティブのみを定義した。

なお、Omni Compiler 自体の問題として、ディレクティブを連続して複数行に書くと、正常にコンパイルされないバグがあった。このバグは OpenACC の記述においても問題であるため、MHOAT にも関わる問題である。よって、本年度の研究ではこの点について修正した。さらに、上

```
void fpga() {
#pragma accomn ondevice(FPGA)
{
#pragma acc data copy(a、 b)
{
#pragma acc kernels loop independent
    for(i=0; i < N; i++)
        a[i] = a[i] + b[i];
}
}
}
```

図 6.46: 拡張ディレクティブの使用例

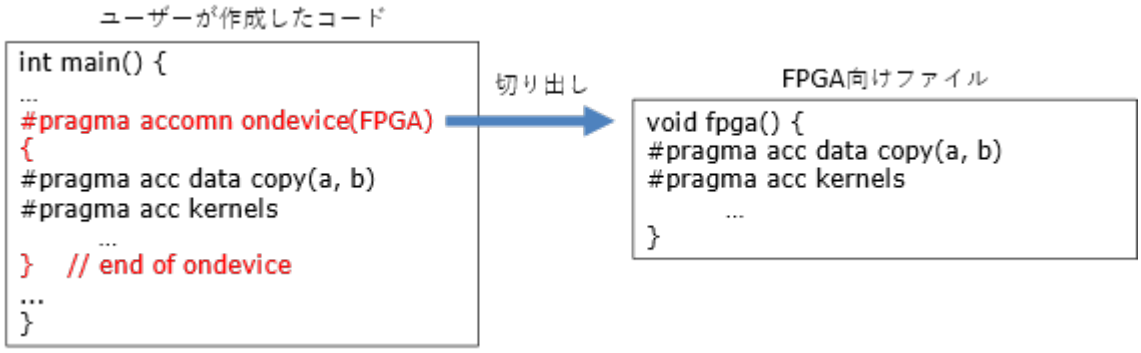


図 6.47: 追加する機能のイメージ

記の修正と合わせて、同じ Omni Compiler にて実装されており、理化学研究所にて開発しているメタプログラミング規格 METAX について、既存の Fortran 向け実装を参考に C 向けのフロントエンドの実装を並行して行った。

現在、完成している MHOAT の実装はここまでであるが、本年度では、デバイス毎に変数スコープが異なってしまう問題の解決を図った。ユーザーに関わる機能の改善点は、main 関数に拡張ディレクティブを記述できるようにすることと、一つの関数内で複数の拡張ディレクティブの記述を可能にすることである。これにより、デバイス毎に変数のスコープが異なってしまう問題を解決する。それに対する MHOAT の実装は、図 4 のように拡張ディレクティブの適用範囲を自動的に関数化し、GPU もしくは FPGA 向けのどちらか適切な方のファイルに出力するようにする。本機能はまだ実装途中であるが、今後の研究で完成させ、GPU・FPGA 協調プログラミングの記述性向上に貢献する。

6.2.3.2 評価

実験では、本研究によってどの程度、GPU・FPGA 協調プログラミングにおけるプログラミングコストが改善するかを確認するため、MHOAT を用いた OpenACC と CUDA + OpenCL でそれぞれプログラムを記述し、記述量と実行時間を比較した。昨年度は、OpenACC のプログラムは MHOAT の開発前だったため、MHOAT の出力と同様のプログラムを手書きして実験したが、今年度は新たに MHOAT によるコンパイル結果を評価した。プログラムの内容は、それぞれのア

Hardware	
CPU	Intel Xeon Gold 6126 (12C / 2.6GHz) x2
Host Memory	DDR4-2666 16GiB x12
GPU	NVIDIA Tesla V100 (32GiB HBM2 PCIe 3.0 x16) x4
FPGA	Intel Stratix 10 GX 2800 (BittWare 520N PCIe Gen3 x16) x2
Software	
OS	CentOS 7
GPU + Host Compiler	PGI Compiler 19.10
FPGA Compiler	OpenARC V0.21
OpenCL Compiler	Intel FPGA SDK for OpenCL 19.4

表 6.7: Cygnus の実行環境

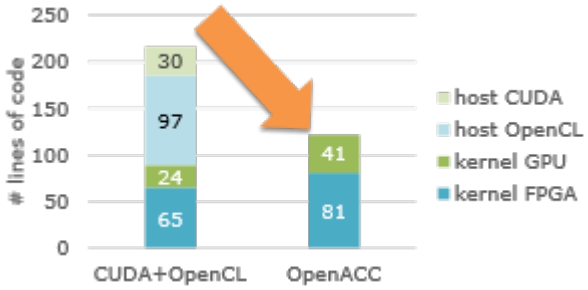


図 6.48: 行数の比較

クセラレータが活用できる演算を想定し、GPU 上で行列積、及びその結果にベクトルを乗算した後、その結果を用いて FPGA 上で CG 法を行う toy program とした。この実験は筑波大学計算科学研究センターのスーパーコンピュータ、Cygnus の 1 ノード上で、GPU と FPGA をそれぞれ 1 台ずつ用いて行った。実行環境を表 1 に示す。

記述量の比較では行数と文字数を比較した。図 12 はプログラムの記述に要する行数を比較したもので、MHOAT を用いた OpenACC 記述の場合、CUDA+OpenCL に比べ約 44%を削減できることがわかった。

また、図 13 は同じ比較を文字数ベースで行なったもので、同じく MHOAT による OpenACC 記述の場合、CUDA+OpenCL に比べ約 64%を削減できることがわかった。以上により、本研究の成果によって、ユーザーが GPU・FPGA 協調プログラミングを行う際の負担を大幅に減らせることが明らかとなった。

一方で、実行時間は OpenACC の方が CUDA + OpenCL に対し、GPU では 1.7 倍、FPGA では 1.07 倍となった。FPGA についてはほぼ同等のパフォーマンスになったことから、FPGA では性能を落とすことなく、OpenACC による簡潔な記述を行うことができる事がわかる。一方で、GPU の方はそれと比較して差が大きい事がわかった。原因として、CUDA では shared memory を使っているのに対し、OpenACC の GPU 側（PGI compiler）のコンパイル結果では使われて

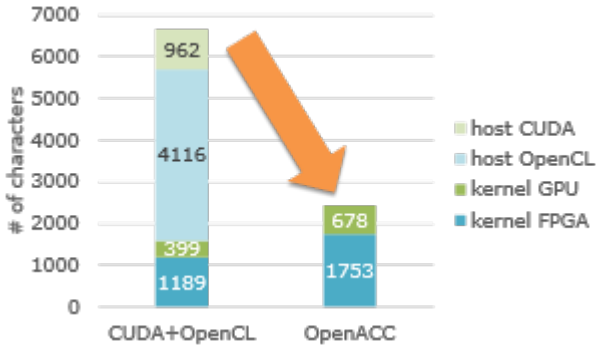


図 6.49: 文字数の比較

いなかったため、その差ではないかと考えられる。CUDA のコードでも shared memory 非使用で実行した場合、性能は大きく低下した。しかし、OpenACC による最適化はコンパイラに大きく依存するため、OpenACC の記述の仕方、特に並列度の指定による性能の変化についてはより詳しく解析する必要がある。本成果は、The 13th International Symposium on High-level Parallel Programming and Applications (HLPP 2020) にて発表した [2]。また、本研究に関して昨年度発表した情報処理学会 HPC 研究会報告により、今年度の CS 領域奨励賞を受賞した [7]。

6.2.4 FPGA 間高性能通信システムに関する研究

近年、高性能計算 (HPC) の分野で Field Programmable Gate Array (FPGA) が注目されている。従来、FPGA 開発の困難さが FPGA を HPC に適用する際の障壁であったが、高位合成 (HLS: High Level Synthesis) の技術によって解消されつつある。最新の FPGA ボードは最大で 100Gbps × 4 と、非常に強力な外部通信ポートを持つ。我々は、この FPGA が持つ高性能な通信機構に注目しており、本研究では高位合成が持つユーザフレンドリーな開発性能と通信機構を組み合わせて用いる。

6.2.4.1 OpenCL 向け FPGA 間通信フレームワーク：CIRCUS

HPC アプリケーションでは、Message Passing Interface (MPI) が高速通信網を利用するための通信ライブラリとして一般的に用いられている。しかしながら、FPGA の演算アーキテクチャは CPU と全く異なっているため、我々は MPI の Application Programming Interface (API) を直接 FPGA 向けに移植することは、良い戦略だと考えていない。パイプラインは FPGA における基本的な処理構造であり、OpenCL コンパイラはコード中にあるループからパイプラインを構築する。したがって、演算パイプラインと通信パイプラインを接続し、すべてをパイプラインにして利用することが最適であると考えている。

パイプライン通信を導入する利点は、通信と演算が完全にオーバーラップされることである。従来手法では、アプリケーションの中で明示的に通信と演算のオーバーラップを記述しなければならなかった。例えば、演算を止めないために、MPI_Isend や MPI_Irecv といった non-blocking な通信 API を使用しなければならず、また、通信に用いるデータの寿命や、通信とデータの依存関係を考慮にいれてプログラムを記述しなければならない。

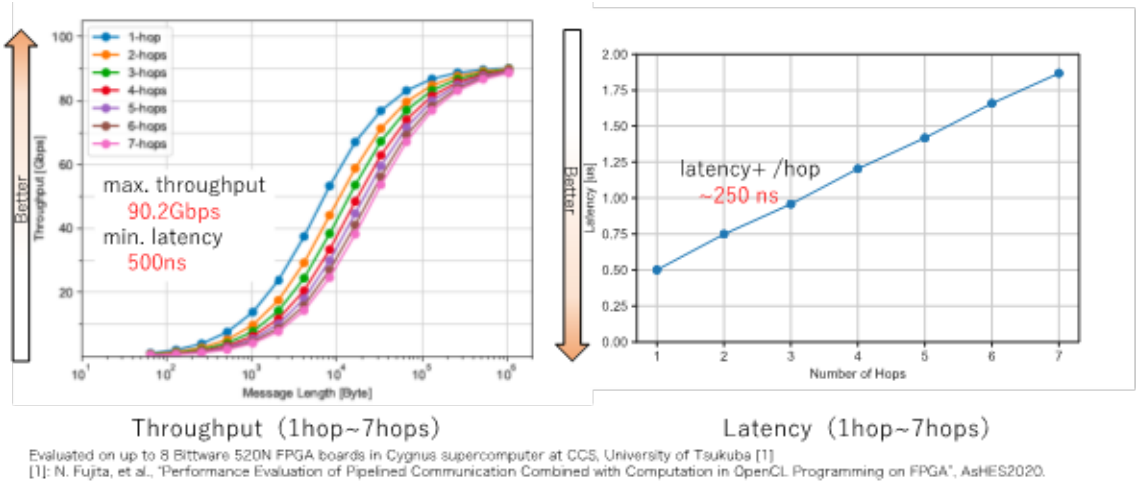


図 6.50: CIRCUS の通信性能評価

一方、パイプライン通信では、通信と一体となったパイプラインを自然に得られ、演算と通信をクロックサイクルレベルの粒度でオーバーラップできる。我々は、再構成可能なハードウェアが持つこの特性は、GPU といった他のアクセラレータにはないものであり、利点であると考えている。

本研究の目的は、FPGA を用いた並列システムを構築し、FPGA が持つ強力な 100Gbps 通信を OpenCL から扱えることを示すことである。そして、それを実現するために、我々は Communication Integrated Reconfigurable CompUting System (CIRCUS) システムを提唱している。

図 14 に CIRCUS の通信性能評価を示す。この性能評価は筑波大学計算科学研究センター（CCS）が運用しているスーパーコンピューター Cygnus 上で行い、最大で 8 枚の Bittware 520N FPGA Board を用いた。520N は 100Gbps の通信ポートを 4 つ持ち、Cygnus では 8x8 の 2D トーラスネットワークが構築されている。CIRCUS の通信帯域は最大で 90.2Gbps、通信レイテンシは最小で 500ns が得られた。また、2D トーラスネットワーク上で非隣接のノードに通信する際のレイテンシの増分は約 250ns/hop ということがわかった。

6.2.4.2 宇宙輻射輸送コードの並列 FPGA 化

筑波大学計算科学研究センターでは、宇宙輻射輸送シミュレーションコード ARGOT (Accelerated Radiative transfer on grids using Oct-Tree) の開発を行っている。ARGOT は輻射輸送問題を解く際に 2 つにアルゴリズムを組み合せており、点光源からの輻射輸送を ARGOT アルゴリズム、空間的に広がった光源からの輻射輸送を ART (Authentic Radiation Transfer) アルゴリズムを用いて解く。ART 法は ARGOT プログラムの中で 90%以上の計算時間を占める重要なアルゴリズムであり、本研究では ART 法の部分に注目し FPGA 向けに最適化を行う。ART 法では問題空間を 3 次元のメッシュに分割し、その中でレイトレーシングを行うことで輻射輸送の計算を行う。図 15 に示すように、レイは境界から発射され、それぞれのレイが平行に直進し、反射や屈折はしない。

ART 法はレイトレーシングを用いているため、あるレイに関する計算は進路に応じて順序通りに計算しなければならないが、異なるレイの間には計算の依存関係がなく並列に計算できる。ART 法のボトルネックはレイがメッシュを通過するたびに指数関数を含む計算（式 1）しなければならないことや、メッシュデータに対するメモリアクセスパターンがレイの方向によって様々 (数百〜

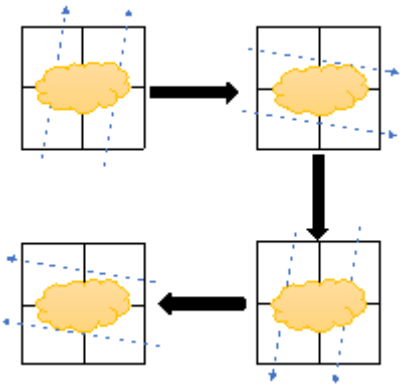


図 6.51: ART 法の ray の動き

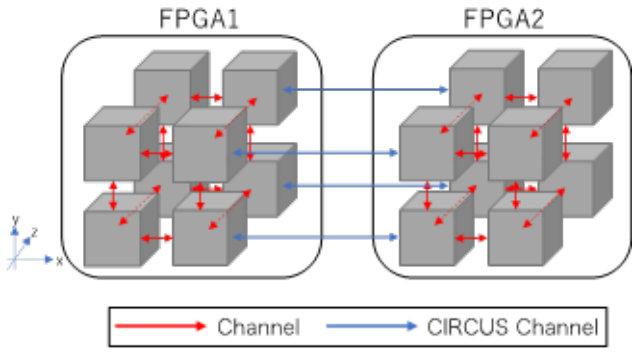


図 6.52: CIRCUS を使った ART の実装

数千パターンになりうる) になることである。 FPGA はオンチップの内蔵メモリを持ち、低レイテンシ・高バンド幅にランダムアクセスが可能である。それに加えて、FPGA であれば ART 法に最適化したメモリアクセス回路をハードウェアに組み込めるため、ART 法は FPGA での実装に適したアルゴリズムであると考えている。

$$I_\nu^{out}(\hat{\boldsymbol{n}}) = I_\nu^{in}(\hat{\boldsymbol{n}})e^{-\Delta\tau_\nu} + S_\nu(1 - e^{-\Delta\tau_\nu}) \qquad (1)$$

数式 1: ART 法カーネルの式

図 16 にどのようにして ART 法を FPGA に実装しているのかを示す。図にある立方体は Processing Element (PE) であり、ART 法の計算が実装された OpenCL カーネルである。同じ FPGA 内は、OpenCL の Intel FPGA 向け拡張である Channel を用いて接続されており、レイデータをやり取りして計算を行う。複数の FPGA を用いる時は、FPGA 間の通信に CIRCUS を用いて擬似的に巨大な FPGA があるように振る舞う。

NVIDIA V100 GPU と、Bittware 520N FPGA Board を用いた性能評価の結果を図 17 に示す。これは ARGOT プログラムから作成したマイクロベンチマークコードを用いて、323 サイズの問題を解いた時の性能である。また、Weak Scaling の条件で最大 4 ノードまで用いる。この時、 1 ノードあたり 1 アクセラレータ（GPU or FPGA）という条件で実行する。FPGA 実装は 1、 2、 4 ノード実行時にそれぞれ 5.70、 8.41、 10.6 倍、GPU 実装よりも高速という結果が得られた。ま

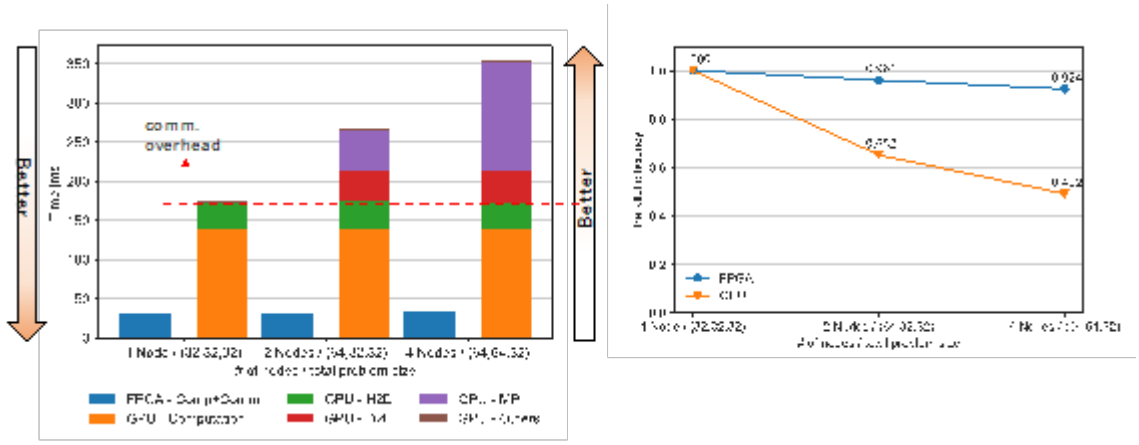


図 6.53: ART on FPGA の性能評価

た、４ノード実行時に 0.924 という並列化効率が得られ、CIRCUS による低遅延広帯域なパイプライン通信によって高い性能が得られたことがわかる。

今後の課題としては、より大規模な問題を用いた性能評価を行うことが挙げられる。我々がターゲットとしている問題サイズは 10243 であり、これは 1 ノードのメモリで扱える問題サイズでは到底ない。Cygnus 環境では、最大で 64 枚の FPGA が扱えるため、それらをすべて用いた並列計算を行いたいと考えている。ただし、現在の FPGA 版の実装では、その条件で計算を行うために必要な機能が不足しているため、不足している機能の実装を進めなければならない。

6.3 京都大学

「ポスト京の高並列SIMD機構およびプロセッサアーキテクチャに関する研究」

6.3.1 概要

本研究の目的は、フラグシップ2020プロジェクトの下で理化学研究所計算科学研究機構が開発したスーパーコンピュータ富岳について、そのプロセッサアーキテクチャ、特に高並列SIMD機構に関する評価を行い、プロセッサおよびコンパイラの研究開発へのフィードバックを行うことにある。本報告書では、2020年に行った富士通製の富岳プロセッサA64FX用コンパイラの3つの版の評価を中心に述べる。

6.3.2 富士通製コンパイラの評価

本研究項目では、PIC法によるプラズマシミュレーションコード[41]のカーネルループ5種について、以下に示すSVEアーキテクチャ対応の富士通製コンパイラの各バージョンにより生成されるコードの品質を評価した。

F-2003 富士通コンパイラ 4.1.0 2020.02.26版。

F-2006 富士通コンパイラ 4.2.0 2020.06.30版。

F-2011 富士通コンパイラ 4.3.0a 2020.11.16版。

なお、全ての版についてwhileループのベクトル化を有効とするために、オプション`-Ksimd_uncounted_loop`を付与している。

比較評価の対象として、以下のコンパイラによるコードも参照した。

F-1912 富士通コンパイラ 4.1.0 2019.12.19版。

HC 筆者によるSVE対応の手動コンパイル。

Intel-KNL Xeon Phi/KNL (AVX-512) [18]をターゲットとしたIntel Composer v18.0.0。

6.3.2.1 `particle_push()`のカーネルループ

図6.54に、関数`particle_push()`のカーネルループのソースコードを示す。このループは、右辺・左辺配列の単純な連続アクセスと各要素に対するスカラ乗算から成り、そのベクトル化は容易であると考えられる。実際Intel-KNLでは最内ループに対してベクトル化と2-way loop unrollingが実施されている[43]。

富士通コンパイラについては2019年度報告[46]に示したように、F-1912で図6.55に概念的に示す概ね最適なコードが生成されている。特に4-way software pipeliningが施された最内main loopは、以下に示す必要最小限の命令から構成されている。

```
const double (*ef)[esz][esy][esx] =
  ((const double(*)[3][esz][esy][esx])Efld)[mu];
const double (*bf)[esz][esy][esx] =
  ((const double(*)[3][esz][esy][esx])Bfld)[mu];
double (*const restrict et)[esz][esy][esx] = (double(*)[esz][esy][esx])Etemp;
double (*const restrict bt)[esz][esy][esx] = (double(*)[esz][esy][esx])Btemp;
const int xb = -1,  xe = UBound[mu][0] + 1,  xe2 = xe + 1;
...
for (int z=zb; z<ze2; z++)
  for (int y=yb; y<ye2; y++)
    for (int x=xb; x<xe2; x++) {
      et[0][z][y][x] = ef[0][z][y][x] * qmr;
      et[1][z][y][x] = ef[1][z][y][x] * qmr;
      et[2][z][y][x] = ef[2][z][y][x] * qmr;
      bt[0][z][y][x] = bf[0][z][y][x] * qmr;
      bt[1][z][y][x] = bf[1][z][y][x] * qmr;
      bt[2][z][y][x] = bf[2][z][y][x] * qmr;
    }
```

図 6.54: `particle_push()` のカーネルループ

- `ef[d][z][y][x]` などの load のための `ld1d {zt.d}, pg/z, [xn,xm,lsl #3]` (`xn` $= f$`[d][z][y]`, $f \in \{$`ef`, `bf`$\}$, `d` $\in \{0,1,2\}$, `xm` $=$ `x` $+ m$, $m \in \{0,8,16,24\}$, $24 = 6 \times 4$ 個)
- `ef[d][z][y][x]*qmr` などのための `fmul` ($24 = 6 \times 4$ 個)
- `et[d][z][y][x]` などの store のための `st1d {zt.d}, pg/z, [xn,xm,lsl #3]` (`xn` $= f$`[d][z][y]`, $f \in \{$`et`, `bt`$\}$, `d` $\in \{0,1,2\}$, `xm` $=$ `x` $+ m$, $m \in \{0,8,16,24\}$, $24 = 6 \times 4$ 個)
- `x`, `x+8`, `x+16` を保持するレジスタの increment と `x+24` の計算のための `add` (4 個)
- ループ制御のための `sub`, `cmp`, `bge` (3 個)[1]

また、main loop でのベクトルレジスタ消費は `ef[d][z][y][x]` などの保持のための 24 個 (6×4) と `qmr` のための 1 個、整数レジスタ消費は `ef0zy=ef[0][z][y]` などのための 12 個と `x0`, `x1`, `x2`, `x3`, `n` のための 5 個であり、いずれも必要最小限である。

一方 main loop 以外の部分については、以下の問題がある。

1. `x` ループで不変である `ef0zy` など 12 個の配列ベースのレジスタ割付が、main loop とそれ以外では異なっており、software pipeline の prologue で `mov` によりコピーされている。このコピーは明らかに無駄であり、また 12 個の余分なレジスタ消費がなければ、remainder loop および `y` ループでの合計 4 個の spill と、`z` ループでの 14 個の spill の内の 8 個が解消できる。

[1] 厳密には、ループ終了条件 `n<7` を `x0>xb+(xn-7)*8` に置換すれば `sub` が不要になる。

```
int off0010=esx*8, off0100=esy*esx*8, off1000=esz*esy*esx
int off00y0=off0010*yb, off0z00=off0100*zb;
int off0zy0=off0z00+off0y00, off1zy0=off0zy0+off1000, off2zy0=off1zy0+off1000;
void *efv=(void*)ef, *etv=(void*)et, *bfv=(void*)bf, *btv=(void*)bt;
void *ef0zyb=efv+off0zy0, *ef1zyb=efv+off1zy0, *ef0zyb=efv+off2zy0;
void *et0zyb=etv+off0zy0, *et1zyb=etv+off1zy0, *et0zyb=etv+off2zy0;
void *bf0zyb=bfv+off0zy0, *bf1zyb=bfv+off1zy0, *bf0zyb=bfv+off2zy0;
void *bt0zyb=btv+off0zy0, *bt1zyb=btv+off1zy0, *bt0zyb=btv+off2zy0;
if(zb<ze2) for(z=zb,zn=ze-zb;zn!=0;z++,zn--){
  void *ef0zy=ef0zyb, *ef1zy=ef1zyb, *ef2zy=ef2zyb;
  void *et0zy=et0zyb, *et1zy=et1zyb, *et2zy=et2zyb;
  void *bf0zy=bf0zyb, *bf1zy=bf1zyb, *bf2zy=bf2zyb;
  void *bt0zy=bt0zyb, *bt1zy=bt1zyb, *bt2zy=bt2zyb;
  if(yb<ye2) for(y=yb,yn=ye-yb;yn!=0;y++,yn--){
    if(xe2<xb){
      int xn=xe2-xb, n=xn>>3, x0=xb;
      if(n>=7){
        int x1=x0+8, x2=x0+16;
        for(;n>=3;n-=4){ /* 4-way software pipelined main loop body */
          int x3=x2+8; // x0+24
          *((double*)et0zy+x0)<0:7>=*((double*)ef0zyx+x0)<0:7>*qmr;
          // by ld1d+fmul+st1d, repeat for et[1],et[2],bt[0],bt[1],bt[2]
          x0=x2+16; // x0+=32
          *((double*)et0zy+x1)<0:7>=*((double*)ef0zyx+x1)<0:7>*qmr;
          // by ld1d+fmul+st1d, repeat for et[1],et[2],bt[0],bt[1],bt[2]
          x1=x2+24; // x1+=32
          *((double*)et0zy+x2)<0:7>=*((double*)ef0zyx+x2)<0:7>*qmr;
          // by ld1d+fmul+st1d, repeat for et[1],et[2],bt[0],bt[1],bt[2]
          x2+=32;
          *((double*)et0zy+x3)<0:7>=*((double*)ef0zyx+x3)<0:7>*qmr;
          // by ld1d+fmul+st1d, repeat for et[1],et[2],bt[0],bt[1],bt[2]
        }
      }
      for(;n!=0;n--){ /* body for n=1, 2, ..., or 6 */ }
      int m=xn-((xn>>3)<<3)
      for(;m!=0;m--){ /* vectorized remainder */ }
    }
    ef0zy+=off0010; ef1zy+=off0010; ef2zy+=off0010;
    // repeat for et, bf, bt
  }
  ef0zyb+=off0100; ef1zyb+=off0100; ef2zyb+=off0100;
  // repeat for et, bf, bt
}
```

図 6.55: F-1912 が生成する particle_push ループの概念的コード

2. software pipelined loop 以外の 2 個の x ループ、すなわち software pipelining の remainder とベクトル化の remainder では、`et[0][z][y][x]=ef[0][z][y][x]*qmr` などのためのコードが

   `ld1d`(1), `fmul`(1), `st1d`(1), `ld1d`(2), `fmul`(2), `st1d`(2), ..., `ld1d`(6), `fmul`(6), `st1d`(6)

   となっており、遅延隠蔽のためのスケジューリングが全く施されていない。すなわち 31 個のベクトルレジスタが利用可能な状況であるにも関わらず、6 個を消費して

   `ld1d`(1), ..., `ld1d`(6), `fmul`(1), ... `fmul`(6), `st1d`(1), ..., `st1d`(6)

   というコードを得る単純なスケジューリングが施されていない。

3. z ループ末尾の `ef0zyb+=off0100` など 12 個の加算で共通に参照される `off0100=esy*esx*8` に対して、`ldp` および `ldr` による spill-in が 11 回行われている。この要因はおそらく、`ef0zyb` などの spill-in/out を伴う加算を、単純な

   `ldr`(1), `add`(1), `str`(1), `ldr`(2), `add`(2), `str`(2), ...

   という繰り返しではなく、

   ..., `ldr`(2), `str`(1), `add`(2), `ldr`(3), `str`(2), ...

   のように interleave するスケジューリングを施したコードとしつつ、使用するレジスタ数を最小化するために `add` で `off0100` を保持するレジスタを破壊してしまうことにあると思われる。このようなスケジューリング自体が有効であるとしても、そのために spill-in を繰り返す意義は不明で、かつ多数の整数レジスタが利用可能な状況で spill-in が必要なレジスタ割付をしてしまうことも不可解である。

F-2003 以降のコードは F-1912 のものと本質的に等価であり、上記の問題点は解決されおらず [47]、(1) についてはレジスタ割付に関する将来課題とされている [17]。

6.3.2.2 `ppush()` のカーネルループ 1

関数 `ppush()` の最初のカーネルループは、図 6.56 に示すように[2]、ある単位立方体セル内に存在する全ての荷電粒子に対するローレンツ力を計算し、それに基づいて各粒子の速度ベクトルを更新するループである。粒子 p の位置ベクトル $\boldsymbol{x}(p)$ および速度ベクトル $\boldsymbol{v}(p)$ の x, y, z 成分は、それぞれ SOA (structure of array) 型の 1 次元配列の p に対応する要素 $\boldsymbol{x}(p) = (\mathtt{px}[p], \mathtt{py}[p], \mathtt{pz}[p])$ および $\boldsymbol{v}(p) = (\mathtt{vx}[p], \mathtt{vy}[p], \mathtt{vz}[p])$ として与えられ、またあるセル $c = [x, x{+}1) \times [y, y{+}1) \times [z, z{+}1)$ 内の粒子、すなわち $\boldsymbol{x}(p) \in c$ を満たす粒子 p は、$\mathtt{head} \le p < \mathtt{tail}$ なるインデックスを持つ。粒子 p に作用するローレンツ力は、$\boldsymbol{x}(p)$、$\boldsymbol{v}(p)$、およびセル内の全粒子に共通する質量・電荷によって補正された電磁場のセル頂点の値 $\boldsymbol{E}(x{+}\Delta_x, y{+}\Delta_y, z{+}\Delta_z)$ と $\boldsymbol{B}(x{+}\Delta_x, y{+}\Delta_y, z{+}\Delta_z)$ $(\Delta_x, \Delta_y, \Delta_z \in \{0, 1\})$ により定まる。

このループの特徴は、ローレンツ力の計算に使用するセルの基準座標 $(x, y, z) = (\mathtt{x0}, \mathtt{y0}, \mathtt{z0})$ と、48 個 $(48 = 2 \times 8 \times 3)$ の電磁場ベクトル要素 $\boldsymbol{E}(x{+}\Delta_x, y{+}\Delta_y, z{+}\Delta_z)_d = \mathtt{e}d\Delta_x\Delta_y\Delta_z$ および

[2] 図にはマクロ `Push_Part()` を展開したものを示している。

```
for (int p=head; p<tail; p++) {
  const double px0 = px[p],  py0 = py[p],  pz0 = pz[p];
  const double dx0 = px0-x0,  dy0 = py0-y0,  dz0 = pz0-z0;
  const double xy1 = dx0*dy0,  xz1 = dx0*dz0,  yz1 = dy0*dz0;
  const double zz2 = 1.0-dz0,  xz2 = dx0*zz2,  yz2 = dy0*zz2;
  const double v3 = xy1*dz0,  v2 = xz1-v3;
  const double v4 = yz1-v3,   v1 = dz0-xz1-v4;
  const double v7 = xy1*zz2,  v6 = xz2-v7;
  const double v8 = yz2-v7,   v5 = zz2-xz2-v8;
  const double eex = ex100*v1 + ex101*v2 + ex111*v3 + ex110*v4 +
    ex000*v5 + ex001*v6 + ex011*v7 + ex010*v8;
  const double eey = ey100*v1 + ey101*v2 + ey111*v3 + ey110*v4 +
    ey000*v5 + ey001*v6 + ey011*v7 + ey010*v8;
  const double eez = ez100*v1 + ez101*v2 + ez111*v3 + ez110*v4 +
    ez000*v5 + ez001*v6 + ez011*v7 + ez010*v8;
  const double bbx = bx100*v1 + bx101*v2 + bx111*v3 + bx110*v4 +
    bx000*v5 + bx001*v6 + bx011*v7 + bx010*v8;
  const double bby = by100*v1 + by101*v2 + by111*v3 + by110*v4 +
    by000*v5 + by001*v6 + by011*v7 + by010*v8;
  const double bbz = bz100*v1 + bz101*v2 + bz111*v3 + bz110*v4 +
    bz000*v5 + bz001*v6 + bz011*v7 + bz010*v8;
  const double brs = 2.0/(1.0+bbx*bbx+bby*bby+bbz*bbz);
  double vx0 = vx[p] + eex,  vy0 = vy[p] + eey,  vz0 = vz[p] + eez;
  const double vx1 = vx0 + vy0*bbz - vz0*bby;
  const double vy1 = vy0 + vz0*bbx - vx0*bbz;
  const double vz1 = vz0 + vx0*bby - vy0*bbx;
  vx0 = vx0 + brs*(vy1*bbz-vz1*bby);
  vy0 = vy0 + brs*(vz1*bbx-vx1*bbz);
  vz0 = vz0 + brs*(vx1*bby-vy1*bbx);
  vx[p] = vx0 + eex;  vy[p] = vy0 + eey;  vz[p] = vz0 + eez
}
```

図 6.56: ppush() のカーネルループ 1

$\boldsymbol{B}(x+\Delta_x, y+\Delta_y, z+\Delta_z)_d = \mathtt{b}d\Delta_x\Delta_y\Delta_z$ ($d \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}$, $\Delta_x, \Delta_y, \Delta_z \in \{0,1\}$) が、総計 51 個のループ不変スカラ変数として与えられることにある。これらのスカラ変数を用いることで、ループは完全に SIMD ベクトル化可能であるが、これらを全てベクトルレジスタに収めることはできないため、ループ内でのスカラ変数の参照方法がコンパイラによるコード生成のポイントの一つとなる。

以下このループについて、いくつかの観点からコード生成の良否を評価する。なおこのループについては、F-1912 と F-2003 のコードは本質的に等価であるため、F-2003 の評価は省略する。

6.3.2.2.1 命令数の評価

表 6.8 は、各コンパイラが生成したコード中に出現する命令の数を、以下の 7 つのタイプに分類して示したものである。

load `px[p]` などの配列要素の load (vec,array)、メモリに割り付けられたスカラ変数の内、あらかじめベクトル長分に展開したものに対する load (vec,spill) と、実行時に broadcast して展開する load (bcast,spill)、およびスカラレジスタへ spill-in する load (scalar,spill)。なお Intel-KNL については、これらの分類に相当するメモリオペランド演算命令を含めた数をカッコ内に示す。

store `px[p]` などの配列要素の store (vec,array)、およびスカラレジスタを spill-out する store (scalar,spill)。

move ベクトルレジスタ間のデータ移動 (V←V)、スカラレジスタからベクトルレジスタへの broadcast (V←S)、SVE 固有の即値の broadcast (imm)、および SVE 固有の `movprfx`。

add/mul 全てのオペランドがベクトルレジスタであるような加減乗算 (reg)、SVE 固有の即値オペランドを持つもの (imm)、および KNL に固有のメモリオペランドを持つもの (mem)。

fma 乗算項破壊型 (×) および加算項破壊型 (+) の FMA のうち、全てのオペランドがベクトルレジスタであるもの (reg) と、KNL に固有のメモリオペランドを持つもの (mem)。

div `brs=2.0/(1.0+...)` のための除算を、逆数との乗算で行うための一連の命令。なお SVE にも AVX-512 にもベクトル除算命令が存在するが、どのコンパイラも逆数近似命令などから成る命令列を生成している。

loop ループ制御および配列要素インデックス計算のためのスカラ命令。

なお ISA の違いにより、命令数の単純な総計 (raw) での比較が難しいため、以下の補正を加えた値 (norm) により比較することとする。

- Intel-KNL のメモリオペランド演算命令に対して、同数の load を加算する。
- 除算に関連する命令は全て除外する。

この補正値と、オペランド数を増やすための `movprfx` を補正値から除外した値 (norm−mp) のいずれについても、富士通コンパイラの各版の命令数は、HC には及ばないものの Intel-KNL よりも有意に小さい値となっている。ただし F-1912 からの改善は `movprfx` の 1 命令減少に留まっている。

表 6.8: ppush() カーネルループ 1 の命令数比較

inst. type		F-1912	F-2006	F-2011	HC	Intel-KNL
load	(vec,array)	6	6	6	6	3 (6)
	(vec,spill)	0	0	0	0	0(38)
	(bcast,spill)	15	15	15	11	1 (1)
	(scalar,spill)	1	1	1	0	0
store	(vec,array)	3	3	3	3	3
	(scalar,spill)	0	1	1	0	0
move	(V←V)	0	0	0	0	6
	(V←S)	21	21	21	22	
	(imm)	1	1	1	0	
	(movprfx)	9	8	8	7	
add/mul	(reg)	23	23	23	23	31
	(imm)	1	1	1	3	
	(mem)					26
fma	(×,reg)	23	11	11	3	12
	(×,mem)					4
	(+,reg)	40	52	52	57	21
	(+,mem)					11
div		9	9	9	8	8
loop		3	3	3	3	3
total	(raw)	155	154	154	148	129
	(norm)	146	145	145	140	162
	(norm−mp)	137	137	137	133	162

表 6.9: ppush() カーネルループ 1 のレジスタ割付

compiler	vector register				scalar. reg.	memory				imm. const
	global	const	local	unused	global	global (×8)	global (×1)	scalar	const	
F-1912	15	0	16	1	21	0	15	1	0	1[*1,2]
F-2006	15	0	16	1	21	0	15	1	0	1[*1,2]
F-2011	15	0	16	1	21	0	15	1	0	1[*1,2]
HC	18	0	14	0	22	0	11	0	0	0[*3]
Intel-KNL	16	2	14	0	0	38[*4]	0	0	1[*5]	0

[*1] 定数 2.0 の乗算を加算で代替。
[*2] fdup で生成する 1.0 とは別に演算命令即値も使用。
[*3] 1.0, 2.0 は演算命令即値でのみ使用。
[*4] 2 回参照する配列要素 3 個を含む。
[*5] 除算用の冗長な定数 1.0。broadcast して展開。

6.3.2.2.2 レジスタ割付

前述のように本節のカーネルループには、51 個のループ不変スカラ変数（3 個のセル基準座標と、48 個の電磁場ベクトル要素）があり、これら全てを 32 個のベクトルレジスタに割り付けることは不可能である。またループボディで行われる計算も比較的複雑であるため、それに必要なループ内ローカル変数や一時変数も多数存在することから、register pressure は更に厳しいものとなり、表 6.9 に示すように、どのコンパイラでも多数の変数をベクトルレジスタ以外の領域に割り付けている。なお表の global はループ不変スカラ変数、scalar はスカラレジスタのメモリへの spill、const はループ内に出現する浮動小数点定数 (1.0, 2.0)、local はループ内ローカル変数／一時変数が、それぞれ割り付けられたレジスタやメモリスロットの個数を意味する。なおメモリに割り付けられたループ不変スカラ変数については、8 重展開されたものを (×8)、展開せずに取得時に broadcast するものを (×1) と標記している。また unused は変数等が割り付けられていないレジスタの個数を、imm. const は命令即値で与えられる定数の個数を、それぞれ意味する。

富士通コンパイラの評価

表 6.9 から明らかなように、富士通コンパイラの各版でレジスタに割り付けることができたループ不変スカラ変数は 15 個であり、2020 年度での改善は見られない。具体的な課題と改善状況を以下に示す。

1. 強い register pressure のために変数等の spill が生じる状況であるにも関わらず、変数等が何も割り付けられていないベクトルレジスタが存在する。

2. 無意味に life-time が長いレジスタが存在するなど、必要以上に複数の式評価を interleave しており、spill の回避・削減を優先する命令スケジューリングとなっていない。この結果、表 6.9 で local に分類されるベクトルレジスタは 16 個であり、HC の 14 個よりも有意に多くなっている。

3. spill したループ不変変数の多くをスカラレジスタに割り付けている。しかし表 6.10 に示す

表 6.10: ppush() カーネルループ 1 のスカラレジスタ割付

用途	F-1912	F-2006	F-2011	HC
frame base	2	2	2	1
loop counter/index	2	2	2	2
array base	6[*1]	6[*1]	6[*1]	6
loop invariant	21	21	21	22

[*1] 1 個は spill-in。

表 6.11: ppush() カーネルループ 1 の eex 等の計算での FMA 命令

FMA	F-1912	F-2006	F-2011
fmla	35	41	41
fmad	7	1	1

ように、スカラレジスタの使用状況に以下のような問題があり、この割付戦略が十分に機能していない。

- ループ中では参照されない stack frame base x29 を使用していない。ただしこの問題は、コード規約によるものと思われる。
- 占有的に割り付けられた配列ベースに対して、不必要な spill-in が行われている。

なおコード規約で x29 が使用できないとすると、各版での割付は不必要な spill-in を除いて最適である。

4. 48 個のループ不変スカラ変数 e$d\Delta_x\Delta_y\Delta_z$ と b$d\Delta_x\Delta_y\Delta_z$ (d $\in \{$x, y, z$\}$, $\Delta_x, \Delta_y, \Delta_z \in \{0, 1\}$) は、以下の ex$\Delta_x\Delta_y\Delta_z$ の例のような式でのみ参照される。

```
eex = ex100*v1 + ex101*v2 + ex111*v3 + ex110*v4 +
      ex000*v5 + ex001*v6 + ex011*v7 + ex010*v8;
```

この式を評価するための自然な命令列は、最終的に eex を保持すべきレジスタを destination とする ex100*v1 のための fmul と、それに引き続き各項の乗算・加算を行う 7 個の fmla から構成される。しかし表 6.11 に示すように、F-1912 では乗算項レジスタを破壊して部分和を格納する fmad が 7 個使用され、fmad の前後で部分和を保持するレジスタが 2 個必要となり、(1) で述べた表 6.9 の local に分類されるレジスタ数を増やす要因となっていた。一方 F-2006/F-2011 での fmad は 1 個であり、かつ乗算項 v8 の最終参照でのみ使用されているため、レジスタ数に関する悪影響は完全に排除されている。

なお上記の問題 (1)–(3) の解決は、将来課題とされている [17]。

6.3.2.2.3 除算

富士通コンパイラの各版では、除算 x/y の計算を、$u = \texttt{frecpe}(y)$ で得られる $1/y$ の初期近似値 u を元に、6 次収束漸化式を用いて

$$x \cdot u \cdot \sum_{i=0}^{6} \varepsilon^i = (x \cdot u) + ((x \cdot u) + (x \cdot u)(\varepsilon + \varepsilon^2))(\varepsilon + \varepsilon^4) = x(1 - \varepsilon^7)/y \approx x/y$$

により行っている（ただし $\varepsilon = 1 - u \cdot y$）。

2019 年度報告 [46] で述べたように、F-1912 では $x = 2.0$ であることを利用しつつ、4 個のレジスタを消費する以下のコードを生成しており、F-2006/F-2011 でも同様である。

```
// z1 = y,  z2 = 1.0
frecpe  z3.d, z1.d              // z3 = u = frecpe(y)
fmls    z2.d, p0/m, z1.d, z3.d  // z2 = ε = 1 − u · y
fadd    z3.d, z3.d, z3.d        // z3 = x · u
movprfx z1.d, p0/z, z2.d        // z1 = ε
fmla    z1.d, p0/m, z2.d, z2.d  // z1 = ε + ε^2
fmul    z0.d, z2.d, z2.d        // z0 = ε^2
fmad    z1.d, p0/m, z3.d, z3.d  // z1 = t = (x · u) + (x · u)(ε + ε^2)
fmla    z2.d, p0/m, z0.d, z0.d  // z2 = ε + ε^4
fmla    z3.d, p0/m, z1.d, z2.d  // z3 = (x · u) + t · (ε + ε^4) = x/y
```

しかし、2018 年度報告 [45] で示したように、式の評価順序を変更することにより、以下に示すように上記のコードと等価で消費レジスタ数が 3 個となるコードを生成することができる。

```
// z1 = y,  z2 = 1.0
frecpe  z0.d, z1.d              // z0 = u = frecpe(y)
fmls    z2.d, p0/m, z1.d, z0.d  // z2 = ε = 1 − u · y
fadd    z0.d, z0.d, z0.d        // z0 = x · u
fmul    z1.d, z2.d, z2.d        // z1 = ε^2
fmad    z1.d, p0/m, z1.d, z2.d  // z1 = ε + ε^4
fmla    z2.d, p0/m, z2.d, z2.d  // z2 = ε + ε^2
fmad    z2.d, p0/m, z0.d, z0.d  // z2 = t = (x · u) + (x · u)(e + e^2)
fmad    z1.d, p0/m, z2.d, z0.d  // z1 = (x · u) + t · (e + e^4) = x/y
```

両者の差異は、中間結果の定義・参照距離を拡大することを優先するか、使用レジスタ数を最小化することを優先するかにあるが、out-of-order 実行では前者のメリットは一般に小さく、少なくとも強い register pressure の下では後者を優先すべきと考えられる。したがってこの問題は 6.3.2.2.2 節の項目 (2) と同様に、過度の interleave によるレジスタの過剰使用の一種と考えられる。

なお、この問題の解決は将来課題とされており [17]、F-2011 で導入された消費レジスタ数が少ないが critical path が長いアルゴリズムの選択オプションは、本質的な解決にはなっていない。

6.3.2.2.4 Remainder Loop

富士通コンパイラのいずれの版においても、本節のループは 8 重 unrolling してベクトル化された main loop と、predicating によってベクトル化された remainder loop から構成されている。2018/2019 年度報告 [45, 46] で指摘したように、main loop と remainder loop ではレジスタ割付や

命令順序が異なり、かつ remainder loop では main loop にはない変数の spill-in/out や定数の再生成が生じていた。表 6.12 に示すように、この問題は 2020 年度の各版でも解消されておらず、むしろ悪化している。なお表中の不変変数と定数に付記した `zi`, `xi`, mem は、変数・定数がそれぞれ（main loop では）ベクトルレジスタ、スカラレジスタ、メモリに割り付けられていることを示し、dup は `dup` や `fdup` による 8 重展開を伴うことを意味する。

6.3.2.3 `ppush()` のカーネルループ 2

関数 `ppush()` の 2 番目のカーネルループは、図 6.57 に示すように[3]、ある単位立方体セル内に存在する全ての荷電粒子 p について、第 1 カーネルループにより更新された速度ベクトル $\boldsymbol{v}(p) = (\mathtt{vx}[p], \mathtt{vy}[p], \mathtt{vz}[p])$ に基づき、位置ベクトル $\boldsymbol{x}(p) = (\mathtt{px}[p], \mathtt{py}[p], \mathtt{pz}[p])$ の更新と、セルの各辺に定義される電流密度ベクトル $\boldsymbol{J}(x+\Delta_x, y+\Delta_y, z+\Delta_z)$ $(\Delta_x, \Delta_y, \Delta_z \in \{0, 1\})$ の積算を行う。また更新前後の位置ベクトルにより定まる粒子のセル間移動方向 $\lfloor \boldsymbol{x}(p) + \boldsymbol{v}(p) \rfloor - \lfloor \boldsymbol{x}(p) \rfloor \in \{-1, 0, 1\}^3$ をコード化した d_p $(-13 \leq d_p \leq 13)$ を、配列 `mdir`$[p]$ に書き込む操作も行う。

このループの特徴は、電流密度の計算に使用するセルの基準座標 $(x, y, z) = (\mathtt{x0}, \mathtt{y0}, \mathtt{z0}) = \lfloor \boldsymbol{x}(p) \rfloor$ と、粒子が移動後に属するセルの基準座標 $\lfloor \boldsymbol{x}(p) + \boldsymbol{v}(p) \rfloor$ を計算するための座標 $(\mathtt{xbm2}, \mathtt{ybm2}, \mathtt{zbm2})$ が、総計 6 個のループ不変スカラ変数として与えられ、また電流密度の積算が 12 個のスカラ変数 $\mathtt{j}d\Delta_x\Delta_y\Delta_z = \boldsymbol{J}(x+\Delta_x, y+\Delta_y, z+\Delta_z)_d$ $(d \in \{x, y, z\}, \Delta_d = 0)$ に対して行 われることに ある。これらのスカラ変数を用いることで、ループは完全に SIMD ベクトル化可能であるが、これらを全てベクトルレジスタに収めることができるか否かが、コード生成のポイントの一つとなる。

以下このループについて、6.3.2.2 節と同様のものを含め、いくつかの観点からコード生成の良否を評価する。なおこのループについては、F-1912 と F-2003 のコードは本質的に等価であるため、F-2003 の評価は省略する。

6.3.2.3.1 命令数の評価

表 6.13 は、各コンパイラが生成したコード中に出現する命令の数を、6.3.2.2.1 節と同様の分類に従って示したものである。ただし本節のカーネルループで出現しない分類項目は除外し、また以下の分類項目を追加している。

add/mul ベクトルレジスタ間の比較演算と max 演算を、このカテゴリに追加。

cvt 浮動小数点数と整数の間での相互の変換。

表 6.12: `ppush()` カーネルループ 1 の remainder loop の冗長 spill-in/out

			F-1912	F-2006	F-2011
ループ不変変数	`zi`	in+dup	1	1	1
	`zi`	in	2	2	2
	`xi`	out+in	13	15	14
	mem	out+in	5	6	5
配列ベース		in	1	1	1

[3]図にはマクロ `Scat_Curr()` と `Jvec_Inc()` を展開したものを示している。

```
for (int p=head,i=0; p<tail; p++,i++) {
  const double px0 = px[p],  py0 = py[p],  pz0 = pz[p];
  const double px1 = px[p] = px0 + vx[p];
  const double py1 = py[p] = py0 + vy[p];
  const double pz1 = pz[p] = pz0 + vz[p];
  const int xx = px1 - xbm2,  yy = py1 - ybm2,  zz = pz1 - zbm2;
  const double x1 = xx + xbm2,  y1 = yy + ybm2,  z1 = zz + zbm2;
  mdir[i] = (z1-z0)*9.0 + (y1-y0)*3.0 + (x1-x0);
  const double xr = (x0==x1) ? (px0+px1)*0.5 : ((x0<x1) ? x1 : x0);
  const double yr = (y0==y1) ? (py0+py1)*0.5 : ((y0<y1) ? y1 : y0);
  const double zr = (z0==z1) ? (pz0+pz1)*0.5 : ((z0<z1) ? z1 : z0);
  const double xd0 = (px0+xr)*0.5 - x0;
  const double yd0 = (py0+yr)*0.5 - y0;
  const double zd0 = (pz0+zr)*0.5 - z0;
  const double vx0w4 = yd0*zd0,  vy0w4 = zd0*xd0,  vz0w4 = xd0*yd0;
  const double vx0w2 = yd0-vx0w4,  vx0w3 = zd0-vx0w4;
  const double vy0w2 = zd0-vy0w4,  vy0w3 = xd0-vy0w4;
  const double vz0w2 = xd0-vz0w4,  vz0w3 = yd0-vz0w4;
  const double vx0w1 = 1.0-yd0-vx0w3;
  const double vy0w1 = 1.0-zd0-vy0w3;
  const double vz0w1 = 1.0-xd0-vz0w3;
  const double vvx0 = +(xr-px0);
  const double vvy0 = +(yr-py0);
  const double vvz0 = +(zr-pz0);
  jx000 += vvx0*vx0w1;  jx010 += vvx0*vx0w2;
  jx100 += vvx0*vx0w3;  jx110 += vvx0*vx0w4;
  jy000 += vvy0*vy0w1;  jy100 += vvy0*vy0w2;
  jy001 += vvy0*vy0w3;  jy101 += vvy0*vy0w4;
  jz000 += vvz0*vz0w1;  jz001 += vvz0*vz0w2;
  jz010 += vvz0*vz0w3;  jz011 += vvz0*vz0w4;
}
```

図 6.57: ppush() のカーネルループ 2

表 6.13: ppush() カーネルループ 2 の命令数比較

inst. type		F-1912	F-2006	F-2011	HC	Intel-KNL
load	(vec,array)	6	6	6	6	3(9)
	(vec,spill)	0	0	0	0	0(1)
store	(vec,array)	4	4	4	4	4
move	(V←V)	0	0	0	0	7
	(imm)	3	3	3	1	
	(movprfx)	6	6	6	3	
add/mul	(reg)	41	41	41	42	33
	(imm)	6	6	6	3	
	(mem)					3
fma	(×,reg)	4	3	3	4	9
	(×,mem)					4
	(+,reg)	12	13	13	13	13
	(+,mem)					0
cvt		6	6	6	6	6
loop		4	4	3	3	3
total	(raw)	92	92	91	85	85
	(norm)	92	92	91	85	92
	(norm−mp)	86	86	85	82	92

命令数の総計に対して 6.3.2.2.1 節と同様の補正を行った結果を比較すると、富士通コンパイラ各版の命令数は Intel-KNL よりも少なく、F-2011 で 1 命令減少の改善があったものの、依然として HC には劣る結果となっている。この要因は、6.3.2.3.3 節で述べるレジスタ割付品質の差、それに部分的に関連する加減乗算と FMA の選択の差、および配列インデックス制御の差によるものである。

6.3.2.3.2 ループ制御と配列要素アクセス

本節のカーネルループでは、固有の配列である mdir[] の更新が存在するが、富士通コンパイラ以外では

```
int n = tail - head;
for (int pp=0; pp<n; p++) { ... }
```

への変換と、それに基づき mdir[] を含む配列ベースの設定が行われるため[4]、表 6.13 に示したようにループ制御のためのスカラ命令数は 3 である。またスカラレジスタ消費量は、pp と n、および 6 個の配列ベースの、総計 8 個である。

一方、富士通コンパイラの F-2006 までの各版では、mdir[i] (i = p − head) の i を別のインデックスとしているため余分にスカラレジスタを消費し、かつこのレジスタの increment するために余分なスカラ命令が必要となっている。この問題の改善は F-2011 で解消し、ループインデックスの共通化が行われるようになった [17]。

[4] i=pp であるため、mdir[] に関するベースは mdir 自身である。

表 6.14: ppush() カーネルループ 2 のベクトルレジスタ割付

	vector register				memory		imm
compiler	global	const	local	unused	global	const	const
F-1912	18	0	14	0	0	0	3[*1,2]
F-2006	18	0	14	0	0	0	3[*1,2]
F-2011	18	0	14	0	0	0	3[*1,2]
HC	18	2	12	0	0	0	1[*1]
Intel-KNL	18	4[*3]	10	0	3[*4]	1	0

[*1] 1.0 は演算命令即値でのみ使用。
[*2] fdup で生成する 0.5 とは別に演算命令即値も使用。
[*3] 式変形の結果生じる陽には現れない定数 2.0 を含む。
[*4] 2 回参照する配列要素 3 個を含む。

6.3.2.2.3.3 ベクトルレジスタ割付

前述のように本節のカーネルループには、6 個のループ不変スカラ変数（3 個のセル基準座標と、3 個の floor 計算用座標）と、電流密度積算用の 12 個の縮約スカラ変数があり、ループボディで行われる比較的複雑な計算に必要なループ内ローカル変数や一時変数も多いため、register pressure は厳しいものとなっている。実際、表 6.14 に示すように、どのコンパイラでも全ての変数・定数をベクトルレジスタに固定的に割り付けることができず、変数・定数に対するメモリアクセスや定数生成命令が生じるコードとなっている。なお表の各列の意味は、6.3.2.2.2 節で示した表 6.9 のものと同じである。またループボディに出現する定数は、0.5, 1.0, 3.0, 9.0 の 4 種である。、

富士通コンパイラの評価

表 6.14 に示すように、富士通コンパイラの各版では、local に分類されるレジスタの個数が 2 個多い。その結果 HC ではレジスタに割り付けている定数 9.0 と 0.5 を fdup で生成しており、表 6.13 に示した全命令数が HC よりも多い要因の一つとなっている。またこの事実とは直接関係しないが、9.0 が出現する式

```
(z1-z0)*9.0 + (y1-y0)*3.0 + (x1-x0);
```

を、富士通コンパイラの各版では；

```
((((z1-z0)*9.0) + (y1-y0)*3.0) + (x1-x0));
```

のように評価するため、fsub を 3 個、fmul を 1 個、fmad または fmla を 1 個、fadd を 1 個の、合計 6 命令を要している（カッコ内の式が 1 命令に相当）。一方 HC では；

```
((((z1-z0)*9.0 + x1) + (y1-y0)*3.0) - x0);
```

の用に順序を変えて評価することで、fsub を 3 個、fmad を 1 個、fmla を 1 個の、合計 5 命令に削減している。この違いも、全命令数が HC よりも多い要因の一つとなっている。

6.3.2.2.3.4 Remainder Loop

6.3.2.2.4 節で述べたように、main loop と remainder loop ではレジスタ割付や命令順序が異なり、表 6.15 に示す main loop では生じていない変数の spill-in や定数の再生成が生じている。

表 6.15: ppush() カーネルループ 2 の remainder loop の冗長 spill-in

			F-1912	F-2006	F-2011
ループ不変変数	zi	in+dup	1	1	1
	zi	in	5	5	5
配列ベース		in	1	1	1

```
for (int p=head,i=0; p<tail; p++,i++) {
  const double px0 = px[p],  py0 = py[p],  pz0 = pz[p];
  const double px1 = px[p] = px0 - vx[p];
  const double py1 = py[p] = py0 - vy[p];
  const double pz1 = pz[p] = pz0 - vz[p];
  const int xx = px1 - xbm1,  yy = py1 - ybm1,  zz = pz1 - zbm1;
  const double x1 = xx + xbm1,  y1 = yy + ybm1,  z1 = zz + zbm1;
  const double xr = (x0==x1) ? (px0+px1)*0.5 : ((x0<x1) ? x1 : x0);
  const double yr = (y0==y1) ? (py0+py1)*0.5 : ((y0<y1) ? y1 : y0);
  const double zr = (z0==z1) ? (pz0+pz1)*0.5 : ((z0<z1) ? z1 : z0);
  const double xd0 = (px0+xr)*0.5 - x0;
  const double yd0 = (py0+yr)*0.5 - y0;
  const double zd0 = (pz0+zr)*0.5 - z0;
  const double vx0w4 = yd0*zd0,  vy0w4 = zd0*xd0,  vz0w4 = xd0*yd0;
  const double vx0w2 = yd0-vx0w4,  vx0w3 = zd0-vx0w4;
  const double vy0w2 = zd0-vy0w4,  vy0w3 = xd0-vy0w4;
  const double vz0w2 = xd0-vz0w4,  vz0w3 = yd0-vz0w4;
  const double vx0w1 = 1.0-yd0-vx0w3;
  const double vy0w1 = 1.0-zd0-vy0w3;
  const double vz0w1 = 1.0-xd0-vz0w3;
  const double vvx0 = -(xr-px0);
  const double vvy0 = -(yr-py0);
  const double vvz0 = -(zr-pz0);
  jx000 += vvx0*vx0w1;  jx010 += vvx0*vx0w2;
  jx100 += vvx0*vx0w3;  jx110 += vvx0*vx0w4;
  jy000 += vvy0*vy0w1;  jy100 += vvy0*vy0w2;
  jy001 += vvy0*vy0w3;  jy101 += vvy0*vy0w4;
  jz000 += vvz0*vz0w1;  jz001 += vvz0*vz0w2;
  jz010 += vvz0*vz0w3;  jz011 += vvz0*vz0w4;
}
```

図 6.58: cscat() のカーネルループ

表 6.16: cscat() カーネルループの命令数比較

inst. type		F-1912	F-2006	F-2011	HC	Intel-KNL
load	(vec,array)	6	6	6	6	3(9)
move	(V←V)	0	0	0	0	7
	(imm)	1	1	1	0	
	(movprfx)	6	6	6	3	
add/mul	(reg)	36	36	36	39	30
	(imm)	6	6	6	3	
	(mem)					3
fma	(×,reg)	3	3	3	3	9
	(×,mem)					3
	(+,reg)	12	12	12	12	12
cvt		6	6	6	6	6
loop		3	3	3	3	3
total	(raw)	79	79	79	75	76
	(norm)	79	79	79	75	82
	(norm−mp)	73	73	73	72	82

6.3.2.4 cscat() のカーネルループ

関数 cscat() のカーネルループは ppush() の第 2 カーネルループと類似しており、図 6.58 に示すように[5]、ある単位立方体セル内に存在する全ての荷電粒子 p について、セルの各辺に定義される電流密度ベクトル $\boldsymbol{J}(x+\Delta_x, y+\Delta_y, z+\Delta_z)$ $(\Delta_x, \Delta_y, \Delta_z \in \{0,1\})$ の積算を行う。ppush() の第 2 カーネルループと同様に、電流密度の計算に使用するセルの基準座標 $(x,y,z) = (\mathtt{x0}, \mathtt{y0}, \mathtt{z0}) = \lfloor \boldsymbol{x}(p) \rfloor$ と、粒子が移動前に属していたセルの基準座標 $\lfloor \boldsymbol{x}(p) - \boldsymbol{v}(p) \rfloor$ を計算するための座標 $(\mathtt{xbm1}, \mathtt{ybm1}, \mathtt{zbm1})$ が、総計 6 個のループ不変スカラ変数として与えられ、また電流密度の積算が 12 個のスカラ変数 $\mathtt{j}d\Delta_x\Delta_y\Delta_z = \boldsymbol{J}(x+\Delta_x, y+\Delta_y, z+\Delta_z)_d$ $(d \in \{x,y,z\}, \Delta_d = 0)$ に対して行われることが特徴である。

以下このループについて、6.3.2.3 節との違いを中心に、コード生成の良否を評価する。なおこのループについては、F-1912 と F-2003 のコードは本質的に等価であるため、F-2003 の評価は省略する。

6.3.2.4.1 命令数の評価

表 6.16 は、各コンパイラが生成したコード中に出現する命令の数を、6.3.2.3.1 節と同様の分類に従って示したものである。富士通コンパイラの各版の命令数は HC に次いで少なく、両者の差は 6.3.2.4.2 節で述べる定数 0.5 を生成する fdup の有無のみであり、レジスタ割付の改善によって HC と同等の命令数を達成できるものと思われる。

6.3.2.4.2 ベクトルレジスタ割付

本節のカーネルループは、類似する ppush() の第 2 カーネルループには存在する比較的複雑な式

[5]図にはマクロ Scat_Curr() と Jvec_Inc() を展開したものを示している。

表 6.17: cscat() カーネルループのベクトルレジスタ割付

	vector register				memory	imm
compiler	global	const	local	unused	global	const
F-1912	18	0	13	1	0	1[*1,2]
F-2006	18	0	13	1	0	1[*1,2]
F-2011	18	0	13	1	0	1[*1,2]
HC	18	1	12	1	0	0[*1]
Intel-KNL	18	3[*3]	10	1	3[*4]	0

[*1] 1.0 は演算命令即値でのみ使用。
[*2] fdup で生成する 0.5 とは別に演算命令即値も使用。
[*3] 式変形の結果生じる陽には現れない定数 2.0 を含む。
[*4] 2 回参照する配列要素 3 個を含む。

表 6.18: cscat() カーネルループの remainder loop の冗長 spill-in

			F-1912	F-2006	F-2011
ループ不変変数	zi	in+dup	1	1	1
	zi	in	3	2	2

```
mdir[i] = (z1-z0)*9.0 + (y1-y0)*3.0 + (x1-x0);
```

がなく、この式にのみ出現する定数 3.0 と 9.0 もないため、register pressure がやや小さい。その結果、富士通コンパイラの各版では HC と同様にメモリへの spill が完全に排除されている。ただし、ループ内で使用していないレジスタが 1 個あるにも関わらず定数 0.5 のための fdup が生成されており、6.3.2.4.1 節で述べた命令数が HC よりも 1 個多い要因となっている。

6.3.2.4.3 Remainder Loop

6.3.2.2.4 節で述べたように、main loop と remainder loop ではレジスタ割付や命令順序が異なり、表 6.18 に示す main loop では生じていない変数の spill-in や定数の再生成が生じている。

6.3.2.5 pmove() のカーネルループ

関数 pmove() のカーネルループは、以下に示す極めて簡単なループである。

```
while (mdir[j]==0.0)  j++;
```

上記の配列 mdir には、ppush() の第 2 カーネルループが計算した各粒子のセル間移動方向コード d ($-13 \leq d \leq 13$) が double で格納されており、上記のループは $d = 0$ の粒子、すなわち元のセルに留まる粒子をスキップするためのものである。多くの場合このような粒子が大半を占めているため、このスキップ操作の効率が pmove() の性能を大きく左右する。

F-1912 でこのループのベクトル化機能が導入されたが、以下の不正なコードが生成されていた。

```
// x20=mdir, w21=j
```

```
        setffr                                  // ffr=true
        ptrue   p1.d, ALL                       // p1=true
.L1152:
        sxtw    x0, w21                         // x0=j
        ldff1d  {z0.d}, p1/z, [x20, x0, lsl #3] // z0<i>=mdir[j+i]
        rdffr   p0.b                            // p0=ffr
        and     p0.b, p1/z, p0.b, p1.b          // p0=p0&p1=ffr (redundant)
        fcmne   p0.d, p0/z, z0.d, #0.0          // p0<i>=z0<i>!=0.0 && ffr<i>
        brkb    p2.b, p1/z, p0.b                // k=p0==0>?8:argmin(p0<i>)
                                                // p2<i>=(i<k)
        sqincp  x21, p2.d, w21                  // j+=k
        ptest   p2, p1.b                        // C=p2==0
        cset    w0, cc                          // w0=C?0:1
        uxtb    w0, w0                          // redundant zero extension
        cbnz    w0, .L1152                      // while(p2!=0)
```

すなわち上記のコードの “ptest p2, p1.d” は誤りであり、オペランドを反転した “ptest p1, p2.d” としなければならない。この誤りは F-2003 で修正され、F-2006/F2011 でも本質的に等価なコードが生成されているが、cset, uxtb, cbnz によるループ継続は明らかに冗長であり、これらは b.last .L1152 に置換することができる。さらに ptest も冗長であり、brkb を brkbs に置換すれば不要となる。

6.3.2.6 スカラコードの品質

本報告では、ベクトル化ループあるいはベクトル化可能な最内ループに対するコードを評価・解析の対象としているが、富士通コンパイラでは対象外の部分についても不可解な低品質コードが生成されている。たとえば ppush() と cscat() には共通して、ベクトル化対象の最内ループでスカラ変数に積算した電流密度を配列に加算する、以下のコードが存在している。

```
jv[0][z  ][y  ][x  ] += jx000 * qhalf;
jv[0][z  ][y+1][x  ] += jx010 * qhalf;
jv[0][z+1][y  ][x  ] += jx100 * qhalf;
jv[0][z+1][y+1][x  ] += jx110 * qhalf;
jv[1][z  ][y  ][x  ] += jy000 * qhalf;
jv[1][z+1][y  ][x  ] += jy100 * qhalf;
jv[1][z  ][y  ][x+1] += jy001 * qhalf;
jv[1][z+1][y  ][x+1] += jy101 * qhalf;
jv[2][z  ][y  ][x  ] += jz000 * qhalf;
jv[2][z  ][y  ][x+1] += jz001 * qhalf;
jv[2][z  ][y+1][x  ] += jz010 * qhalf;
jv[2][z  ][y+1][x+1] += jz011 * qhalf;
```

2018 年度報告 [45] で述べたように、上記のコードを適切にコンパイルした場合、各代入式に対して必須の ldr, fmadd, str を除く命令の数は、表 6.19 の ideal 欄に示すものとなる。

しかし富士通コンパイラの各版では非常に非効率なコード生成が行われており、表に示すように多数の命令が加えられている。

表 6.19: jv[][][][] 更新のための命令数

ppush()	ideal	F-1912	F-2006	F-2011
ldr xi|wi	9	14	14	14
ldr di	1	2	12	5
add	1	1	1	1
sxtw	0	1	1	1
mov	0	1	1	1
total	11	19	29	22
cscat()				
add	1	12	12	12
lsl	0	2	2	2
total	1	14	14	14

表 6.20: 問題点と改善状況

問題点	F-1912	F-2003	F-2006	F-2011
particle_push() のベクトル化	△	⇐	⇐	⇐
配列のインデックス共通化	×	⇐	◯	⇐
ベクトルレジスタ不使用	×	⇐	⇐	⇐
過大な式評価 interleave	×	⇐	⇐	⇐
スカラレジスタへの spill	×	⇐	⇐	⇐
不適切な FMA 選択	×	⇐	◯	⇐
除算命令列での過大なレジスタ消費	×	⇐	⇐	⇐
remainder loop での冗長 spill-in/out	×	⇐	⇐	⇐
pmove() のベクトル化	×	△	⇐	⇐
jv[][][][] 更新コードの非効率性	×	⇐	⇐	⇐

6.3.2.7 まとめ

6.3.2.1–6.3.2.6 節で指摘した問題点は、以前の報告 [47] の中でも指摘しており、いくつかの問題点は表 6.20 に示すように既に改善が実施されている。なお表中の記号の意味は、以下のとおりである。

- × : 2019 年度報告書 [46] で指摘した F-1912 の問題点。
- ⇐ : 前の版に対する改善がなかった問題点。
- △ : 前の版に対して部分的な改善が見られた問題点。
- ◯ : （ほぼ）解決された問題点。

表に示すように、富士通コンパイラの最新版 F-2011 には依然として以下の課題があり、早急な改善が強く求められる。

1. particle_push() のカーネルループのベクトル化はできているが、main loop 以外のコードが非効率である。
2. 強い register pressure のために変数等の spill が生じる状況であるにも関わらず、変数等が何も割り付けられていないベクトルレジスタが存在することがある。
3. 無意味に life-time が長いレジスタが存在するなど、必要以上に複数の式評価を interleave しており、spill の回避・削減を優先する命令スケジューリングとなっていない。
4. スカラレジスタに占有的に割り付けられた配列ベースを spill-in する。
5. 除算のための命令列でのベクトルレジスタの消費数が過大である。
6. remainder loop のベクトル化コードが main loop のコードとは異なり、冗長な spill-in/out が生じている。
7. pmove() のカーネルループ（while ループ）のベクトル化コードに明白な冗長性がある。
8. ppush() と cscat() に共通する配列 jv[][][][] の更新操作など、比較的単純なスカラコードが著しく非効率である。

6.3.3 まとめ

本研究では、富岳用の富士通コンパイラについて、開発段階の各版の評価を行い、開発者へのフィードバックを行った。富岳の開発完了に伴い、今後の改善・改版は富士通の自発的な品質改善に委ねられるが、未解決課題のフォローと他のカーネルでの問題点のやその解決法の提案を行うことが、今後も必要である。

共同研究 成果・参考文献

[1] Arima, E., Classification-Based Unified Cache Replacement via Partitioned Victim Address History, Proceedings of 2020 23rd Euromicro Conference on Digital System Design (DSD), pp.101-108, Aug. 2020

[2] Kodama, Y., Odajima, T., Arima, E., Sato, M., Evaluation of Power Management Control on the Supercomputer Fugaku, Proceedings of 2020 IEEE International Conference on Cluster Computing (CLUSTER), EEHPC workshop volume, pp. 484-493, Sep. 2020

[3] Nakajima, K. Iwashita, T., Yashiro, H., Shimokawabe, T., Matsuba, H., Nagao, H., Ogita, T., Katagiri, T., h3-Open-BDEC: Innovative Software Platform for Scientific Computing in the Exascale Era, ISC High Performance 2020, (Project Poster), June 2020

[4] Nakajima, K., Gerofi, B., Ishikawa, Y., Horikoshi, M., Efficient Parallel Multigrid Solver on Intel Xeon Phi Cluster, ACM Proceedings of IXPUG Workshop in conjunction with HPC Asia 2021, 2021

[5] 有間 英志, 児玉 祐悦, 小田嶋 哲哉, 辻 美和子, 佐藤 三久, 次世代 HPC システムのためのプロセッサアーキテクチャ評価環境と電力性能予測, 研究報告ハイパフォーマンスコンピューティング (HPC), 2020-HPC-178, xx, pp. 1 - 9, 2021

[6] E. Arima, Y. Kodama, T. Odajima, M. Tsuji and M. Sato, "Power/Performance/Area Evaluations for Next-Generation HPC Processors using the A64FX Chip," 2021 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS), 2021, pp. 1-6, doi: 10.1109/COOLCHIPS52128.2021.9410320.

[7] 児玉 祐悦, 小田嶋 哲哉, 有間 英志, 佐藤 三久, スーパーコンピュータ「富岳」における電力制御の評価, 研究報告ハイパフォーマンスコンピューティング (HPC), 2020-HPC-175, 25, pp. 1 - 8, 2020

[8] 中島研吾，坂本龍一，星野哲也，有間英志，塙敏博，近藤正章，低精度演算とアプリケーション性能，情報処理学会研究報告（2020-HPC-174-5），2020

[9] 中島研吾，岩下武史，八代尚，下川辺隆史，松葉浩也，長尾大道，荻田武史，片桐孝洋,（計算＋データ＋学習）融合によるエクサスケール時代の革新的シミュレーション手法，第 25 回計算工学講演会（オンライン，2020 年 6 月），2020

[10] 中島研吾，荻田武史，塙敏博，河合直聡，伊田明弘，星野哲也，低精度・混合精度演算による高性能・高信頼性疎行列ソルバー，情報処理学会研究報告（2020-HPC-175-2），2020

[11] 中島研吾，メニィクラスタ向け並列多重格子法，日本応用数理学会 2020 年度年会（オンライン，2020 年 9 月）

[12] 中島研吾，メニィクラスタ向け並列多重格子法,，情報処理学会研究報告（2020-HPC-176-6），2020

[13] 中島研吾，Balazs Gerofi，石川裕，堀越将司，メニィコアクラスタ向け並列多重格子法の最適化，日本応用数理学会「行列・固有値問題の解法とその応用」研究部会 第 30 回研究会 (2020 年 12 月 7 日，オンライン)2020

[14] ARM Limited, *Scalable Vector Extension: Architecture Overview (Version 00rel2.1)*, Apr. 2017.

[15] ARM Limited, *SVE Instructions (Version 00rel2.1)*. Apr. 2017.

[16] ARM Limited, *ARM Architecture Reference Manual: ARMV8, for ARMv8-A Architecture Profile*. Jan. 2016.

[17] 富士通株式会社, part.c におけるコンパイラ改善について, Feb. 2021.

[18] Intel Corporation, *Intel Architecture Instruction Set Extensions Programming Reference*. Aug. 2015.

[19] Arima, E., Schulz, M., Pattern-Aware Staging for Hybrid Memory Systems, Proceedings of ISC High Performance 2020, pp.474–495, 2020

[20] Arima, E., Hanawa, T., Trinitis, C., Schulz, M., Footprint-Aware Power Capping for Hybrid Memory Based Systems, Proceedings of ISC High Performance 2020, pp.347–369, 2020

[21] A64FX Microarchitecture Manual: `https://github.com/fujitsu/A64FX`

[22] RIKEN Simulator: `https://github.com/RIKEN-RCCS/riken_simulator`

[23] Mucci, P. J., Browne, S., Deane, C., and Ho, G., PAPI: A Portable Interface to Hardware Performance Counters, Proceedings of the department of defense HPCMP users group conference, vol. 710, 1999

[24] Guler, A., and Jha, N. K., McPAT-Monolithic: An Area/Power/Timing Architecture Modeling Framework for 3-D Hybrid Monolithic Multicore Systems, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 10, pp. 2146–2156, 2020

[25] Allan, G., Choosing between DDR4 and HBM in memory-intensive applications, `http://www.techdesign-forum.com/practice/technique/choosing-between-ddr4-and-hbm-in-memory-intensive-applications/`, 2018

[26] Pawlowski, J. T., Prospects for Memory, Work- shop on Memory-Centric High-Performance Computing (MCHPC), 2019

[27] Oakforest-PACS スーパーコンピュータシステム（最先端共同 HPC 基盤施設）: `http://jcahpc.jp/ofp/ofp_intro.html`

[28] Nakajima, K., Optimization of Serial and Parallel Communications for Parallel Geometric Multigrid Method, Proceedings of the 20th IEEE International Conference for Parallel and Dis-tributed Systems (ICPADS 2014) 25-32 (2014)

[29] Monakov, A., Lokhmotov, A., Avetisyan, A., Automatically tuning sparse matrix-vector multiplication for GPU architectures, Lecture Notes in Computer Science 5952, 112-125, 2010

[30] Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units, SIAM Journal on Scientific Computing 36-5, C401-C423, 2014

[31] 細川洋輝, 奥田洋司, 階層的メモリ構造を考慮した反復法ソルバーの混合精度計算，第 24 回日本計算工学会講演会（大宮，2019 年 5 月 29 日），2019

[32] Gerofi, B, Riesen, R., Takagi, M., Boku, T., Nakajima, K., Ishikawa, Y., Wisniewski, R.W., Performance and Scalability of Lightweight Multi-kernel Based Operating Systems, IEEE Proceedings of IPDPS 2018, 116-125, 2018

[33] Nakajima, K., Gerofi, B., Ishikawa, Y., Horikoshi, M., Parallel Multigrid Methods on Many-core Clusters with IHK/McKernel, IEEE Proceedings of 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 2019) in conjunction with SC19, Denver, CO, 2019

[34] Norihisa Fujita, Ryohei Kobayashi, Yoshiki Yamaguchi, Tomohiro Ueno, Kentaro Sano, Taisuke Boku, "Performance Evaluation of Pipelined Communication Combined with Computation in OpenCL Programming on FPGA", Proc. of AsHES2020 (in IPDPS2020), Jun. 2020.

[35] Ryuta Tsunashima, Ryohei Kobayashi, Norihisa Fujita, Taisuke Boku, Seyong Lee, Jeffrey S. Vetter, Hitoshi Murai, Masahiro Nakao, Mitsuhisa Sato, "OpenACC unified programming environment for GPU and FPGA multi-hybrid acceleration", Proc. in HLPP2020, Jul. 2020.

[36] 藤田典久, 小林諒平, 山口佳樹, 上野知洋, 佐野健太郎, 朴泰祐, "OpenCL プログラミングを用いた並列 FPGA 処理システムの性能評価", 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 13, No.3, pp. 13-28, 2020 年 11 月.

[37] Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, Taisuke Boku, Kohji Yoshikawa, Makito Abe, Masayuki Umemura, "Multi-Hybrid Accelerated Simulation by GPU and FPGA on Radiative Transfer Simulation in Astrophysics", 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 13, No.3, 17 pages, 2020 年 11 月.

[38] 藤田典久, 小林諒平, 山口佳樹, 朴泰祐, 吉川耕司, 安部牧人, 梅村雅之, "Stratix 10 FPGA を用いた ray-tracing 法による輻射輸送計算の高速化", 情報処理学会研究報告ハイパフォーマンスコンピューティング (HPC), 2020-HPC-175(8), pp.1-10, 2020 年 7 月 30 日.

[39] 柏野隆太, 小林諒平, 藤田典久, 朴泰祐, "OpenCL 対応 FPGA 間光リンク接続フレームワーク CIRCUS と SMI の性能評価", 情報処理学会研究報告ハイパフォーマンスコンピューティング (HPC), 2020-HPC-175(16), pp.1-8, 2020 年 7 月 31 日.

[40] 2020 年度コンピュータサイエンス領域奨励賞受賞者詳細 https://www.ipsj.or.jp/award/cs-awardee-2020.html 理化学研究所・筑波大学・令和 2 年度共同研究報告

[41] Hiroshi Nakashima, Yoshiki Summura, Keisuke Kikura, and Yohei Miyake. Large Scale Manycore-Aware PIC Simulation with Efficient Particle Binning. In *Proc. Intl. Parallel and Distributed Processing Symp.*, pp. 202–212, May 2017.

[42] 中島 浩, ポスト京の高並列 SIMD 機構およびプロセッサアーキテクチャに関する研究（2015 年度成果報告書）. 13 pages, Mar. 2016.

[43] 中島 浩, ポスト京の高並列 SIMD 機構およびプロセッサアーキテクチャに関する研究（2016 年度成果報告書）. 43 pages, Mar. 2017.

[44] 中島 浩, ポスト京の高並列 SIMD 機構およびプロセッサアーキテクチャに関する研究（2017 年度成果報告書）. 28 pages, Mar. 2018.

[45] 中島 浩, ポスト京の高並列 SIMD 機構およびプロセッサアーキテクチャに関する研究（2018 年度成果報告書）. 30 pages, Mar. 2019.

[46] 中島 浩, ポスト京の高並列 SIMD 機構およびプロセッサアーキテクチャに関する研究（2019 年度成果報告書）. 27 pages, Mar. 2020.

[47] 中島 浩, 富士通 SVE 対応クロスコンパイラ (2020.11.16 版) による part.c のコード生成の評価. C01-201216-R-2, 7 pages, Dec. 2020.