「富岳」セミナー
中級編第一部
講習会資料

登録施設利用促進機関 / 文科省委託事業「HPCIの運営」代表機関
一般財団法人 高度情報科学技術研究機構（RIST）【著】

国立研究開発法人理化学研究所 計算科学研究センター（R-CCS）

本講の目的

- 「富岳」でプログラムを効率的に走らせるためには、チューニングによりプログラムの高速化を目指す必要があります。そのためには、性能情報を採取し、自分のプログラムがどのような特性を持っているかを知ることが重要となります。
- 本講は、チューニング未経験の人を対象に、性能向上策実施の補助となるプログラム性能情報の採取方法や最適化メッセージの見方、重要な最適化手法の概略、および最適化手法の適用例等の基本事項を説明します。

本講の開催予定

本コースは今後開催予定も含めて三部構成を予定しています。

- 第一部（今回開催）：プロファイラ・重要な最適化手法概略編

  第一部ではCPU単体性能最適化の中で特に重要な最適化手法であるSIMD化、ソフトウェアパイプライニング、プリフェッチを取り上げます。各手法について、概念の説明、最適化メッセージ（コンパイルリスト）の見方、プロファイラを用いて効果を確認する方法を説明します。

- 第二部（今後開催予定）：MPI・LLIO編

  第二部ではMPI並列性能情報（MPI統計情報）の採取方法、及び「富岳」に固有なMPIの事項を取り上げる予定です。

- 第三部（今後開催予定）：Fortranでの各種最適化手法編

  第一部で取り上げなかったCPU単体性能の各種最適化手法を取り上げる予定です。

第一部（今回開催）の内容

- 「富岳」の概略

  CPUとメモリ等、Tofuの概略、計算ラックの構成概略を説明します。

- プロファイラ

  性能情報採取ツールである基本プロファイラ、詳細プロファイラ、およびCPU性能解析レポートの使用方法を説明します。

- 重要な最適化手法

  第一部（本講義）ではCPU単体性能最適化の中で特に重要な最適化手法であるSIMD化、ソフトウェアパイプライニング、プリフェッチを取り上げます（それ以外の各種最適化手法は今後開催予定の第三部で取り上げる予定です）。概念の説明、最適化メッセージ（コンパイルリスト）の見方、プロファイラを用いて効果を確認する方法を具体的なループ例にもとづいて説明します。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
- 【付録】参考文献

内容

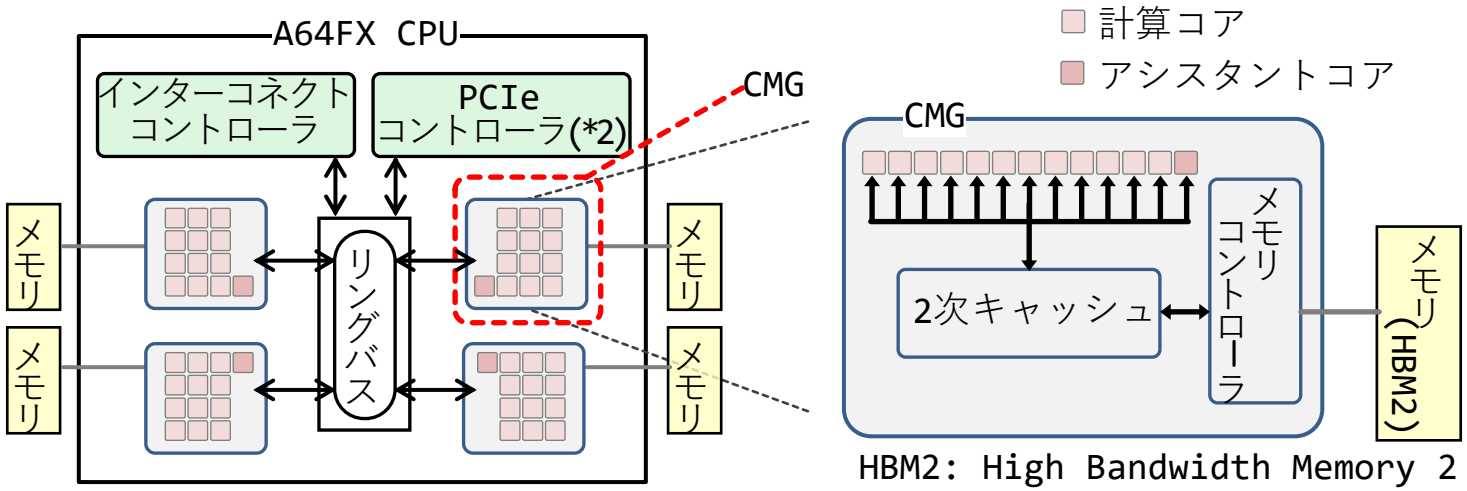
- 「富岳」の概略
  - CPUとメモリ等
  - Tofuの概略
  - 計算ラックの構成
- プロファイラ
- 重要な最適化手法
- 【付録】参考文献

■ CPU（ノード）の構成

CPU内の構成を下図に示す。

- ■ CPUチップ：A64FX
- ■ CPU（ノード）の総数：158,976
- ■ CPU当たりのコア数：
  - □ 計算ノード：計算コア 48 個 + アシスタントコア 2 個(*1)
  - □ 計算兼I/Oノード：計算コア 48 個 + アシスタントコア 4 個(*1)

(*1) 12個の計算コアと1個（または0個）のアシスタントコア、2次キャッシュ、およびメモリコントローラで、1個のCMG（Core Memory Group）を構成する。一つのCPU内には四つのCMGが含まれる。



(*2) PCIeコントローラは計算兼I/Oノードのみに存在する。

■ CPUの演算性能等

■ 動作周波数：

□ 計算ノード：通常モード 2.0 GHz①、ブーストモード 2.2 GHz

□ 計算兼I/Oノード：常に2.2 GHz

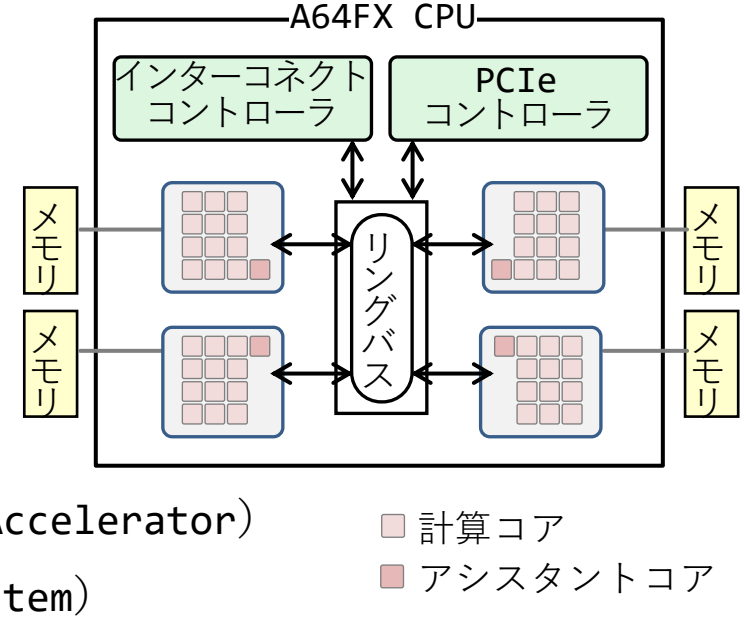
■ 浮動小数点演算器：512-bit SIMD② FMA③ × 2個④

■ ピーク演算性能：コア当たり、倍精度で 64 GFLOPS(*3)、
ノード当たり、倍精度で 3072 GFLOPS

(*3) 1秒間にコア当たり、① 2 G サイクル× ② 倍精度 8 要素 × ③ 2 積和演算 × ④ 2 個 ＝ 64 G 回の浮動小数点演算、即ち、64 GFLOPS

■ 従って、「富岳」のCPU性能を発揮させるためには、SIMDとFMAの活用が重要である。

- レジスタ
  - 浮動小数点レジスタ：コア当たり 32 個
- キャッシュ
  - 1次データキャッシュ：コア当たり 4 ウェイ（4 ウェイの合計 64 KiB）
  - 2次データキャッシュ：CMG当たり 16 ウェイ（16 ウェイの合計 8 MiB）
- メモリ
  - 容量：CPU当たり 32 GiB（CMG当たり 8 GiB）
  - 転送速度：CPU当たり 1024 GB/s
- 計算ノード間ネットワーク
  - Tofu Interconnect D
    (28 Gbps x 2 lane x 10 port)
- ストレージ
  - 階層化ストレージ：
    - 第1階層：LLIO（Lightweight Layered IO-Accelerator）
    - 第2階層：FEFS（Fujitsu Exabyte File System）
    - 第3階層：商用クラウドストレージ（要別途契約）、HPCIストレージ



内容

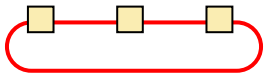
- 「富岳」の概略
  - CPUとメモリ等
  - Tofuの概略
  - 計算ラックの構成
- プロファイラ
- 重要な最適化手法
- 【付録】参考文献

■ Tofuインターコネクト、Tofu単位

- ■ 大規模並列計算では多数のノードを利用してMPIプログラムを実行し、一般に多数のノード間で大量のデータのやり取り（通信）が必要となる。そのため、大規模並列計算では一般に通信時間が大きくなるので通信性能が非常に大事である。
- ■ 通信時間は通信を行うノード間に介在するネットワークの本数（ホップ数）に依存し、ホップ数が少ないほど通信時間は短くなる。
- ■ 通信性能を向上させるためにはホップ数を最小化することが重要であり、それを実現するために、「富岳」ではTofuインターコネクトという技術を導入している。
  - ■ Tofuインターコネクトの、大きさ2×3×2のa、b、c軸（次スライドで説明）で構成される単位をTofu単位という。

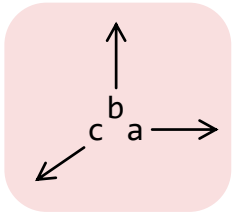
※ Tofuインターコネクト、及び「富岳」に固有なMPIの事項は第二部：MPI・LLIO編（今後開催予定）で説明することを予定していますので、第一部（本講義）ではTofuの概略のみを説明することとします。

- Tofuインターコネクトの、大きさ2×3×2のa、b、c軸で構成される単位をTofu単位という。
  - 一つのTofu単位には、ノード（図の □ の部分）が12個含まれる。
  - 各Tofu単位内の：
    - 各ノードの座標はa、b、c軸で表される。
    - 各ノード間の結合は
      - b軸方向はトーラス結合
        （ノードがリング状に結合）。

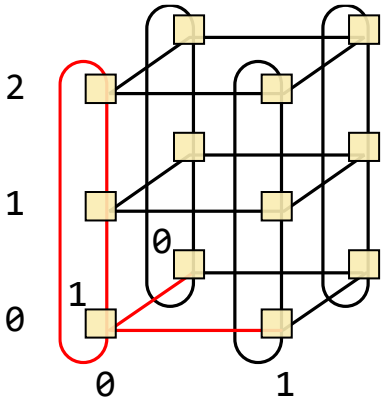


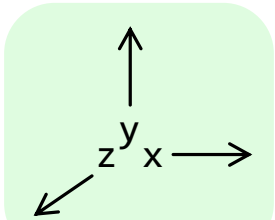
      - a、c軸方向はメッシュ結合
        （ノードが3個以上の場合、両端が結合していない）。

  - 各Tofu単位の座標はx、y、z軸で表される。



Tofu単位





- x、y、z軸について隣接するTofu単位同士では、a、b、c座標を同じくするノード同士が接続されている。
  以後、x、y、z軸およびa、b、c軸を物理座標と呼ぶ。

- 論理座標
  - MPIジョブの実行時に、ユーザはジョブスクリプトにおいて、（例えば「#PJM -L "node=2x6x4"」のように）使用したいノード数を、X、Y、Z軸の各方向の個数で指定する。
  - 使用したいノード数の指定におけるX、Y、Z軸を論理座標と呼ぶ。

※ 色々な指定例は第二部：MPI・LLIO編（今後開催予定）で紹介する予定です。

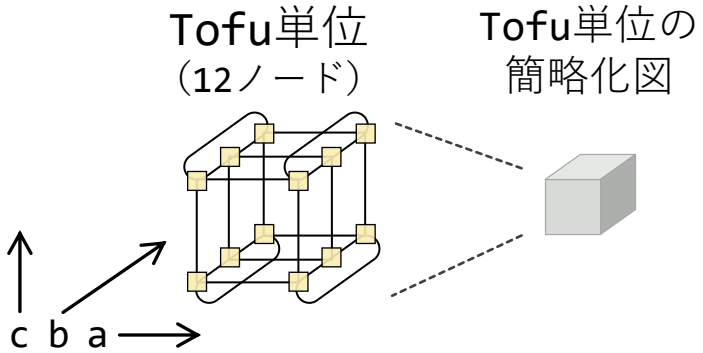
- 論理座標と物理座標の対応付け
  - 論理座標のノードは、実際にはTofuの物理座標上のノードに対応付けられる。
  - この対応付けはシステムが自動的に行うので、ユーザーは関知する必要はない。

内容

- 「富岳」の概略
  - CPUとメモリ等
  - Tofuの概略
  - 計算ラックの構成
- プロファイラ
- 重要な最適化手法
- 【付録】参考文献

- チューニングにおけるノード形状やI/Oの重要性
  - 実行時のノード数やノード形状を決めるにあたっては計算ラックの構成を踏まえて性能に悪影響が出にくい値にするのが良い。
  - 計算ラック内には計算に加えてI/Oを担当するノードというものがあり、I/O性能の改善が求められる場合には計算ラック内でのI/Oノードの位置を意識してノード形状を決める必要がある。
  - I/Oノードのうち、ストレージI/Oノード（SIO）はI/O性能向上を目的とするLLIOの構成要素である。
  - I/Oを分散する場合にも計算ラックの構成を踏まえて検討すると良い。

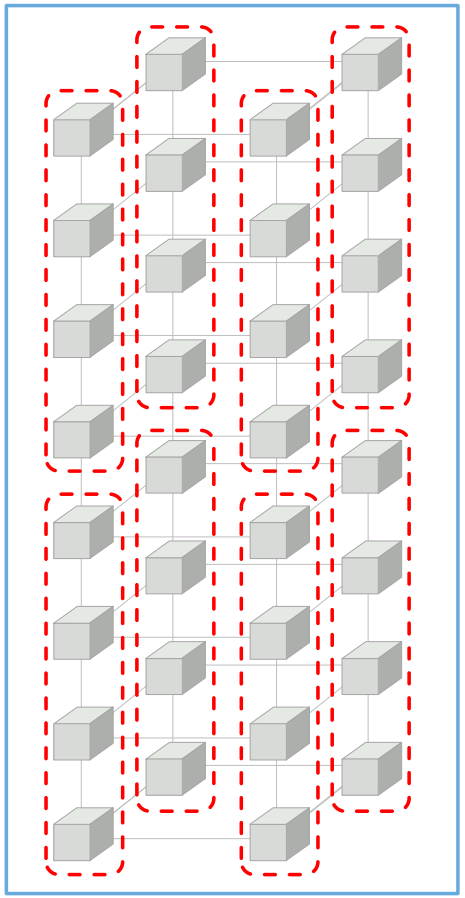
※ ストレージI/Oノード等の位置、LLIOの初歩的な使用例等は第二部：MPI・LLIO編（今後開催予定）で説明することを予定していますので、第一部（本講義）では計算ラック内でのTofu単位の構成の概略のみを紹介することとします。



■ 一つのシェルフにTofu単位が4個含まれる。
- 1個のシェルフに48ノード入る。

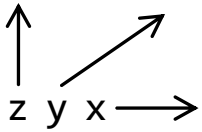
■ 8個のシェルフが一つの計算ラックに収められている。
- 一つの計算ラックに384ノード入る(*1)。

(*1) ただし、36個の計算ラックには192ノード入る。



■ 384ノードのラック数：396、
192ノードのラック数：36、
よって、ノードの総数は、158,976となる。

■ 384ノード以下を使用する場合はジョブスクリプトにおいて「#PJM -L "rscgrp=small"」を、385ノード以上を使用する場合は「#PJM -L "rscgrp=large"」を指定する。
385ノード以上のジョブはシェルフ単位で
ノードが割り当てられる。



内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
- 【付録】参考文献

プロファイラ

「富岳」では、プログラムの高速化を目的として、

(1) プログラム内で計算時間のかかっている部分の調査

(2) 並列ジョブのプロセス間やスレッド間のロードバランスが均等かどうかの調査

(3) 各種性能特性（GFLOPS値、ピーク性能比、メモリスループットなど）の調査

(4) CPUに関するより詳細な情報（実行時間内訳、ビジー率、キャッシュミスなど）の調査

を行うため、下記のプロファイラが提供されている。

- **基本プロファイラ**
  サンプリング解析でプログラム全体または一部の統計情報の計測および出力を行う。
- **詳細プロファイラ（テキスト形式）**
  ハードウェアカウンタによりプログラムの指定した区間の実行性能情報の計測および出力を行う。
- **CPU性能解析レポート**
  ハードウェアカウンタで複数の実行により計測した多数のCPU性能解析情報を集約し、表と付随するグラフを用いて可視化する。

各プロファイラは、Fortran/C/C++の、逐次、スレッド並列、MPI並列、スレッド並列＋MPI並列のいずれのプログラムに対しても適用可能です。

■ 各プロファイラの使い分けの指針

- 手始めとして下記(1)を調査したい場合は、基本プロファイラを用いてプログラムの広い範囲にわたって計測するのが良い。

- 特定のループに注目している状態で、下記(2)や(3)を調査したい場合には詳細プロファイラを用い、より詳しく(4)を調査したい場合にはCPU性能解析レポートを用いるのが良い。

(1) プログラム内で計算時間のかかっている部分の調査
(2) 並列ジョブのプロセス間やスレッド間のロードバランスが均等かどうかの調査
(3) 各種性能特性（GFLOPS値、ピーク性能比、メモリスループットなど）の調査
(4) CPUに関するより詳細な情報（実行時間内訳、ビジー率、キャッシュミスなど）の調査

本章では各プロファイラの使用方法の概要を説明します。

内容

■ Fortran言語のコンパイルコマンド

■ ログインノードでのコンパイルコマンド

■ MPIなしの場合

コンパイルコマンド	最適化オプション	最適化の推奨オプション	自動スレッド並列有効化オプション	OpenMP有効化オプション	スレッド並列化リンク時オプション
frtpx	-O(*1)	-Kfast	-Kparallel	-Kopenmp	-Nlibomp(*2)または -Nfjomplib(*3)

■ MPIありの場合

コンパイルコマンド	最適化オプション	最適化の推奨オプション	自動スレッド並列有効化オプション	OpenMP有効化オプション	スレッド並列化リンク時オプション
mpifrtpx	-O(*1)	-Kfast	-Kparallel	-Kopenmp	-Nlibomp(*2)または -Nfjomplib(*3)

■ 計算ノードでのコンパイルコマンドは、frtpxをfrtに、mpifrtpxをmpifrtに、それぞれ置き換えたもの。

(*1) 後の節「【補足】-Oオプションを指定しない場合／指定した場合のデフォルト」も参照して下さい。
(*2) -Nlibomp：並列処理にLLVM OpenMPライブラリを使用する（デフォルトで有効）。
(*3) -Nfjomplib：並列処理に富士通OpenMPライブラリを使用する。

■ C言語/C++言語 tradモードのコンパイルコマンド

■ ログインノードでのコンパイルコマンド

■ MPIなしの場合

言語	コンパイルコマンド	最適化オプション	最適化の推奨オプション	自動スレッド並列有効化オプション	OpenMP有効化オプション	スレッド並列化リンク時オプション
C	fccpx	-O(*1)	-Kfast	-Kparallel	-Kopenmp	-Nlibomp(*2)または -Nfjomplib(*3)
C++	FCCpx					

■ MPIありの場合

言語	コンパイルコマンド	最適化オプション	最適化の推奨オプション	自動スレッド並列有効化オプション	OpenMP有効化オプション	スレッド並列化リンク時オプション
C	mpifccpx	-O(*1)	-Kfast	-Kparallel	-Kopenmp	-Nlibomp(*2)または -Nfjomplib(*3)
C++	mpiFCCpx					

■ 計算ノードでのコンパイルコマンドは、fccpxをfccに、mpifccpxをmpifccに、FCCpxをFCCに、mpiFCCpxをmpiFCCに、それぞれ置き換えたもの。

(*1) 後の節「【補足】-Oオプションを指定しない場合／指定した場合のデフォルト」も参照して下さい。
(*2) -Nlibomp：並列処理にLLVM OpenMPライブラリを使用する（デフォルトで有効）。
(*3) -Nfjomplib：並列処理に富士通OpenMPライブラリを使用する。

■ C言語/C++言語 clangモードのコンパイルコマンド

■ clangモードの注意点

- clangモードと指定するために、コンパイルおよびリンク時に-Nclangを付ける。
- -Nclangオプションが有効な場合、-Nfjomplibは無効になり、-Nlibompが有効になる。
- clangモードでは自動並列化は対応していない。-Kparallelを付けるとコンパイル時にエラーとなる。

■ ログインノードでのコンパイルコマンド

■ MPIなしの場合

言語	コンパイルコマンド	clangモード有効化の必須オプション	最適化オプション	最適化の推奨オプション	OpenMP有効化オプション
C	fccpx	-Nclang	-O(*1)	-Ofast	-fopenmp
C++	FCCpx				

■ MPIありの場合

言語	コンパイルコマンド	clangモード有効化の必須オプション	最適化オプション	最適化の推奨オプション	OpenMP有効化オプション
C	mpifccpx	-Nclang	-O(*1)	-Ofast	-fopenmp
C++	mpiFCCpx				

- 計算ノードでのコンパイルコマンドは、fccpxをfccに、mpifccpxをmpifccに、FCCpxをFCCに、mpiFCCpxをmpiFCCに、それぞれ置き換えたもの。

(*1) 後の節「【補足】-Oオプションを指定しない場合／指定した場合のデフォルト」も参照して下さい。

■ 【補足】-Oオプションを指定しない場合／指定した場合のデフォルト

- コンパイラの最適化の-Oオプションを指定しない場合／指定した場合のデフォルトは以下のようになっている。

	-Oオプションを 何も指定しない場合	-Oのみを 指定した場合
Fortran	-O2	-O3
C/C++ tradモード	-O2 (*1)	-O2
C/C++ clangモード	-O2 (*1)	-O2

(*1) ■ 「C言語使用手引書」および「C++言語使用手引書」における説明では、従来通り -O0 がデフォルトとの記載となっているので、注意して下さい。

■ C/C++コンパイラにおいて-Oオプションを何も指定しない場合の動作を従来と同じ動作（-O0）としたい時は、コマンドラインオプションまたはコンパイラの環境変数に "-O0" を指定して下さい。

■ コンパイラの環境変数による指定例

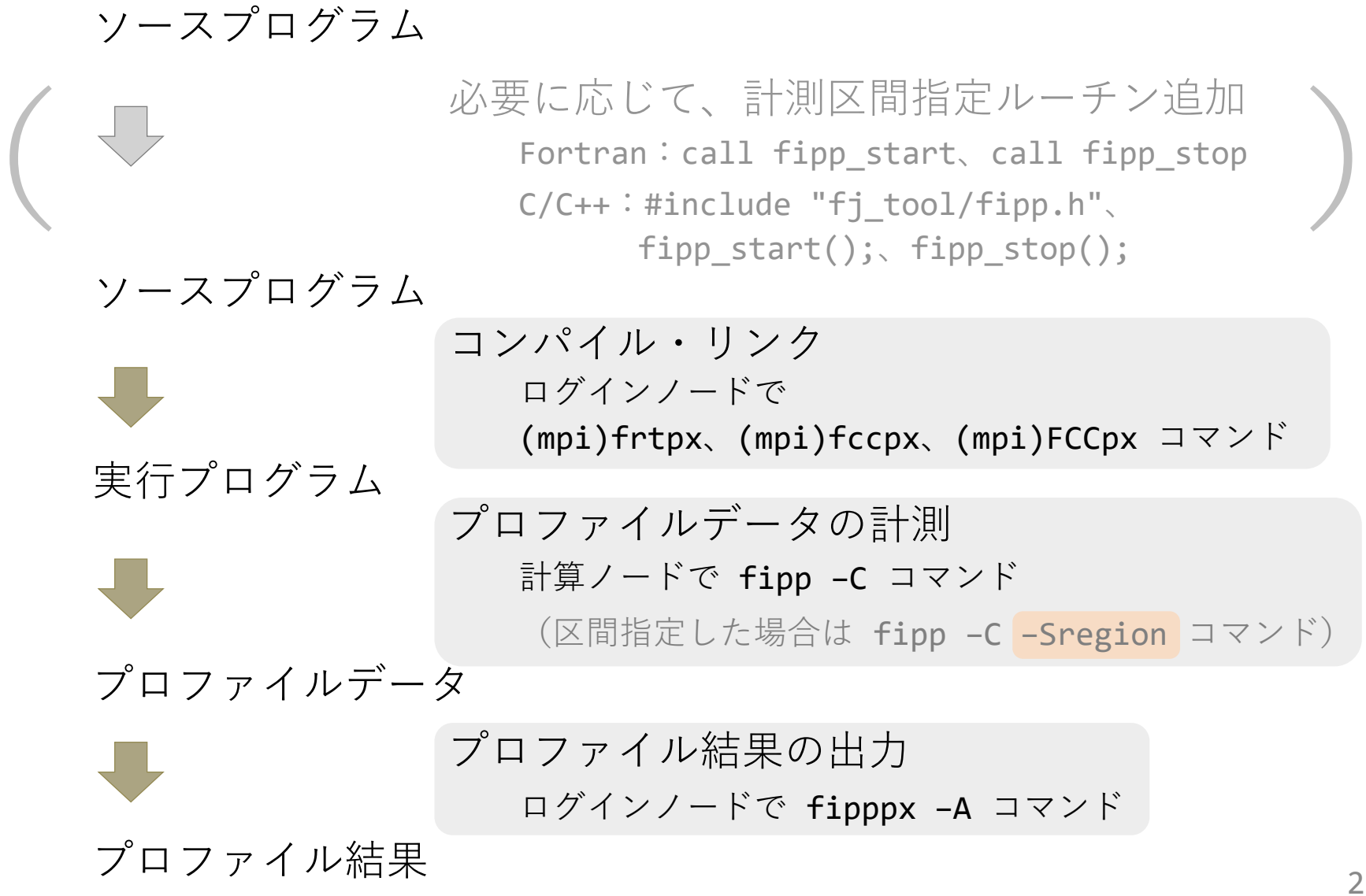
- ログインノードでの指定例
  C言語の場合：export fccpx_ENV="-O0"
  C++言語の場合：export FCCpx_ENV="-O0"
- 計算ノードでの指定例
  C言語の場合：export fcc_ENV="-O0"
  C++言語の場合：export FCC_ENV="-O0"

内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略

    本節では、基本プロファイラ、詳細プロファイラ、CPU性能解析レポートの使用手順の概略を示します。具体的な使用方法は、サンプル・プログラムやサンプル・ジョブスクリプトを用いて、その後の節で示します。
  - 基本プロファイラの使用方法
  - 詳細プロファイラ（テキスト形式）の使用方法
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

■ 基本プロファイラの使用手順概略



■ 詳細プロファイラの使用手順概略

ソースプログラム

計測区間指定ルーチン追加

Fortran：call fapp_start(引数)、call fapp_stop(引数)

C/C++：#include "fj_tool/fapp.h"、
fapp_start(引数);、fapp_stop(引数);

ソースプログラム

コンパイル・リンク

ログインノードで

(mpi)frtpx、(mpi)fccpx、(mpi)FCCpx コマンド

実行プログラム

プロファイルデータの計測

計算ノードで fapp -C コマンド

プロファイルデータ

プロファイル結果の出力

ログインノードで fapppx -A コマンド

プロファイル結果

■ CPU性能解析レポートの使用手順概略

ソースプログラム

ソースプログラム

実行プログラム

プロファイルデータ

プロファイル結果
（csv形式）

CPU性能解析レポート
（Excel形式）

計測区間指定ルーチン追加

Fortran：call fapp_start(引数)、call fapp_stop(引数)

C/C++：#include "fj_tool/fapp.h"、
fapp_start(引数);、fapp_stop(引数);

コンパイル・リンク

ログインノードで
(mpi)frtpx、(mpi)fccpx、(mpi)FCCpx コマンド

プロファイルデータの計測

計算ノードで `fapp -C` コマンド(*1)

(*1) fappコマンドのオプションを変えて複数回の計測が必要。

プロファイル結果の出力

ログインノードで `fapppx -A -tcsv` コマンド(*2)

(*2) 上記(*1)に対応した複数回の実行が必要。

CPU性能解析レポートの作成

Excelによりcpu_pa_report.xlsmを起動

内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略
  - 基本プロファイラの使用方法
  - 詳細プロファイラ（テキスト形式）の使用方法
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

内容

■ サンプルプログラム

- スレッド並列＋プロセス並列の以下のプログラムを例としてプロファイルを行う。
- メインルーチン直下には配列Xをゼロクリアするループと
  サブルーチンSUB1をコールするループが存在する。
- CALL FIPP_STARTとCALL FIPP_STOPを使用して測定区間を上記の処理範囲に限定する。

```
MODULE PARA
  INCLUDE 'mpif.h'
  INTEGER::JSTA,JEND
END MODULE PARA
```

```
PROGRAM MAIN
  USE PARA
  ...
  IF (MYRANK==0) THEN
    JSTA=1; JEND=N/2
  ELSE
    JSTA=N/2+1; JEND=N
  ENDIF
  CALL FIPP_START
  DO J=1,N
    DO I=1,N
      X(I,J) = 0D0
    ENDDO
  ENDDO
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL FIPP_STOP
  ...
END PROGRAM MAIN
```

```
SUBROUTINE SUB1(X,N)
  USE PARA
  DOUBLE PRECISION::X(N,N)
!$OMP PARALLEL
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = 1D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J)=X(I,J)+1D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE SUB1
```

■ コンパイル・リンク

- 前スライドのFortranサンプルプログラム（スレッド並列＋プロセス並列）を、以下のようにコンパイル・リンクする。

```
mpifrtpx -Kfast,parallel,openmp -o a.exe  a.F90
```

- プロファイラ用のライブラリをリンクするためにはリンク時に-Nfjprofの指定が必要であるが、-Nfjprofはデフォルトで有効である。

- 参考
  - C/C++ tradモードの場合、リンク時に-Nfjprofを指定する（デフォルトで有効）。
  - C/C++ clangモードの場合、リンク時に-ffj-fjprofを指定する（デフォルトで有効）。

■ ジョブスクリプト [1/2]

- 本例では、（①②により）プロセス数：2、（③④により）プロセス当たりのスレッド数：2で、ジョブを投入する。
- ⑤でプログラムの実行が開始し、[1]のfippコマンドによってプロファイルが行われる。
  - [2]の「-C」の指定は必須である。
  - 測定区間を限定して計測するため[3]の「-Sregion」を追加する。
- プロファイラは、本例では[4]で指定した10ミリ秒ごとに、その時点でプログラム内のどの行が実行しているかを調べる。これをサンプリングと呼ぶ。
  - -iオプション省略時は-i 100オプションが有効になる。
  - -iオプションの引数には10～3,600,000の範囲の整数値を指定する。

```
#!/bin/bash
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=20:00"
#PJM -L "node=1"                          ①
#PJM --mpi "max-proc-per-node=2"          ②
#PJM --mpi "rank-map-bychip"
#PJM -S
export PLE_MPI_STD_EMPTYFILE=off
export PARALLEL=2                         ③
export OMP_NUM_THREADS=2                  ④
fipp -C -Sregion -Icpupa,call -Puserfunc -i10 -d PD mpiexec -stdout stdout（中略）./a.exe ⑤
 [1] [2]   [3]                              [4]
```

■ ジョブスクリプト [2/2]

- ジョブが終了すると、プロセスごとに、プログラム内の各行ごとに合計サンプリング回数（以下コストと呼ぶ）のデータが得られる。
- データは、[5]の「-d」で指定したディレクトリ（本例ではPD）の下のファイルに入る。
  - 指定したディレクトリが存在しない場合、新規にディレクトリが作成される。
  - 指定したディレクトリが存在する場合、そのディレクトリは空でなければならない。

⚠ fippコマンドを使用する場合、環境変数FLIB_FASTOMPにTRUEを指定して下さい（デフォルトはTRUE）。

```
#!/bin/bash
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=20:00"
#PJM -L "node=1"                          ①
#PJM --mpi "max-proc-per-node=2"          ②
#PJM --mpi "rank-map-bychip"
#PJM -S
export PLE_MPI_STD_EMPTYFILE=off
export PARALLEL=2                         ③
export OMP_NUM_THREADS=2                  ④
fipp -C -Sregion -Icpupa,call -Puserfunc -i10 -d PD mpiexec -stdout stdout（中略）./a.exe ⑤
```

[1] [2] [3] [5]

■ 測定区間を限定する方法についての補足

■ プログラム内の特定の部分のみをサンプリングしたい場合、

- Fortranの場合は、
  - 測定したい区間の前後を⑥ `CALL FIPP_START` と⑦ `CALL FIPP_STOP` で囲み(*1)、

    (*1) 複数の箇所を指定することが可能。その場合、指定した全ての計測区間の結果が合算される。

- C/C++（tradモード、clangモード）の場合は、
  - ⑧のインクルード（`"fj_tool/fipp.h"`）を指定し、
  - 測定したい区間の前後を⑨ `fipp_start();` と⑩ `fipp_stop();` で囲み(*1)、
- ⑪でfippコマンドのオプションに `-Sregion` を追加する。

Fortranの例

```
  CALL FIPP_START          ⑥
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL FIPP_STOP           ⑦
```

C/C++の例

```
#include "fj_tool/fipp.h"  ⑧
  ...
  fipp_start();            ⑨
  for(LL=0; LL<50; LL++){
    SUB1(X,N);
  }
  fipp_stop();             ⑩
```

```
fipp -C -Sregion -Icpupa,call -Puserfunc -i10 -d PD  mpiexec ./a.exe   ⑪
```

内容

■ プロファイル結果の出力 [1/2]

- 本例でのディレクトリPD下に作成されたファイルはバイナリファイルなのでそのままでは読めない。
- ジョブ終了後、ログインノードで[6]のfipppxコマンドを実行する(*1)。
  - [7]の「-A」の指定は必須である。
  - [8]の「-d」の引数には、fippコマンド実行時に指定したディレクトリを指定する。

(*1) ログインノードで[6][7]の「fipppx -A」の代わりに、計算ノードで「fipp -A」と指定して実行することもできる。

```
fipppx -A -pall -Icpupa,balance,call,src[:path] -d PD -o fipp_total.txt    ⑫
  [6]  [7]                                          [8]
```

- [8]で指定したディレクトリ（本例ではPD）にあるファイルが、読める形式（以下、報告書と呼ぶ）に変換され、ファイル（本例ではfipp_total.txt）に書き出される。
- 次スライドへ続く

■ プロファイル結果の出力 [2/2]

■ 【補足】前ページの以下の再掲図における(*2)～(*6)のオプションの機能概略を記す。

```
fipppx -A -pall -Icpupa,balance,call,src[:path] -d PD -o fipp_total.txt    ⑫
  [6]  [7]  (*2)   (*3)  (*4)    (*5)   (*6)        [8]
```

(*2) -pオプションはプロファイル結果に出力するプロセスを指定する。-pallを指定した場合、全プロセスの情報を読み込み、出力する。-pオプションの引数に指定できるその他の引数については「プロファイラ使用手引書」を参照して下さい。

(*3) CPU動作状況を出力する（後のスライドで詳細を説明します）。

(*4) コスト情報にコストバランス情報（並列実行単位間のコストを比較した情報）を出力する。

(*5) コールグラフ情報を出力する。

(*6) ソースコード情報と行ごとのコストを出力する。*path*にはソースコードが存在するディレクトリパスを指定する。*path*を複数指定する場合、コロン(:)で区切って指定して下さい。*path*を省略した場合、プログラム翻訳時に指定したディレクトリパスを参照する。

内容

■ プロファイル結果出力（報告書）の内容

前スライドのfipppxコマンドのオプションで出力した場合、報告書は以下の項目で構成される。

■ Fujitsu Instant Performance Profiler Version x.y.z	← プロファイルデータ計測環境情報
■ Time statistics	← 時間統計情報
□ Performance monitor event:statistics	← CPU動作状況 • fippおよびfipppxコマンドで「-Icpupa」有効時に出力。
■ Procedures profile (Total thread cost basis)	← 手続コスト分布情報
■ Loops profile (Total thread cost basis)	← ループコスト分布情報
■ Lines profile (Total thread cost basis)	← 行コスト分布情報
□ Call graph	← コールグラフ情報 • fippおよびfipppxコマンドで「-Icall」有効時に出力。
□ Sources profile	← ソースコード情報 • fipppxコマンドで「-Isrc[:パス]」有効時に出力。

■ プロファイル結果出力（報告書）の内容

前スライドのfipppxコマンドのオプションで出力した場合、報告書は以下の項目で構成される。

■ Fujitsu Instant Performance Profiler Version x.y.z	← プロファイルデータ計測環境情報
■ Time statistics	← 時間統計情報
□ Performance monitor event:statistics	← CPU動作状況 • fippおよびfipppxコマンドで「-Icpupa」有効時に出力。
■ Procedures profile (Total thread cost basis)	← 手続コスト分布情報
■ Loops profile (Total thread cost basis)	← ループコスト分布情報
■ Lines profile (Total thread cost basis)	← 行コスト分布情報
□ Call graph	← コールグラフ情報 • fippおよびfipppxコマンドで「-Icall」有効時に出力。
□ Sources profile	← ソースコード情報 • fipppxコマンドで「-Isrc[:パス]」有効時に出力。

■ Performance monitor event:statistics（CPU動作状況） [1/2]

- ■ 本項目はfippおよびfipppxコマンドで「-Icpupa」有効時に出力される。
- ■ 全プロセス、プロセスごと、スレッドごとに、以下の各種性能特性が表示される。

各種性能特性	意味	備考
Execution time(s)	計測対象区間の命令実行に要した時間（秒）。	
GFLOPS	1秒あたりに実行された浮動小数点演算数。	GFLOPS値はすべてActive elementとして算出しています(*1)。そのため、Inactive elementの多いプログラムでは本来のGFLOPS値より高い値を出力します。
Floating-point peak ratio(%)	浮動小数点演算性能の理論値に対する実測値の比率（%）。	浮動小数点演算性能の理論値は倍精度演算として算出しています。そのため、単精度や半精度の演算の場合、実際の割合の2倍や4倍の値を出力します。
Mem throughput(GB/s)	メモリースループット（GB/s）。	
Mem throughput peak ratio(%)	メモリースループットの理論値に対する実測値の比率（%）。	

(*1) IF文を含むループのSIMD化においてIF文内の計算はIF文の条件の真偽によらず演算数にカウントされる。

次のスライドへ続く

■ Performance monitor event:statistics（CPU動作状況） [2/2]

前のスライドからの続き

各種性能特性	意味	備考
Effective instruction	実行された命令の総数。	実行された命令の総数には、MOVPRFX命令(*2)を含みません。
Floating-point operation	実行された浮動小数点演算の総数。	
SIMD inst. rate(%)	実行された命令の総数に対するSIMD命令数の比率（%）。	
SVE operation rate(%)	実行された浮動小数点演算の総数に対するSVE演算数の比率（%）。	
IPC	1サイクルあたりに実行された命令数。	
GIPS	1秒あたりに実行された命令数。	

(*2) 積和演算命令の前置命令として、和をとるレジスタの内容を代入先レジスタへコピーする命令。

■ プロファイル結果出力（報告書）の内容（再掲）

前スライドのfipppxコマンドのオプションで出力した場合、報告書は以下の項目で構成される。

- ■ Fujitsu Instant Performance Profiler Version x.y.z ← プロファイルデータ計測環境情報
- ■ Time statistics ← 時間統計情報
- □ Performance monitor event:statistics ← CPU動作状況
  - fippおよびfipppxコマンドで「-Icpupa」有効時に出力。
- ■ Procedures profile (Total thread cost basis) ← 手続コスト分布情報
- ■ Loops profile (Total thread cost basis) ← ループコスト分布情報
- ■ Lines profile (Total thread cost basis) ← 行コスト分布情報
- □ Call graph ← コールグラフ情報
  - fippおよびfipppxコマンドで「-Icall」有効時に出力。
- □ Sources profile ← ソースコード情報
  - fipppxコマンドで「-Isrc[:パス]」有効時に出力。

■ コスト分布情報

■ Procedures profile
(Total thread cost basis)

← 手続コスト分布情報

■ Loops profile
(Total thread cost basis)

← ループコスト分布情報

■ Lines profile
(Total thread cost basis)

← 行コスト分布情報

以下では、Lines profileについて説明する。Procedures profileと
Loops profileは同様なので省略する。

■ Lines profile（行コスト分布情報） [1/5]

■ 本項目は以下の各項目で構成されている。「Application」、「Process n」、及び「Thread m」を情報集計レベルと呼ぶ。

■ Lines profile (Total thread cost basis)

■ Application - lines ← 全プロセスについて、行ごとのコスト分布情報
- ● Cost、Barrier等
- ○ MPI(*2)

■ Process *n* - lines ← プロセス*n*について、行ごとのコスト分布情報
- ● Cost、Barrier等
- ○ MPI(*2)

■ Thread *m* - lines ← プロセス*n*のスレッド*m*について、行ごとのコスト分布情報
- ● Cost、Barrier等
- ○ MPI(*2)

■ 「Process *n*」の他の「Thread」について以下同様。

■ 他の「Process」について以下同様。

(*2) MPIコスト
- fippおよびfipppxコマンドで「-Impi」有効時に出力。

■ Lines profile（行コスト分布情報） [1/5]

■ 本項目は以下の各項目で構成されている。「Application」、「Process n」、及び「Thread m」を情報集計レベルと呼ぶ。

■ Lines profile (Total thread cost basis)

■ Application - lines ← 全プロセスについて、行ごとのコスト分布情報
- ● Cost、Barrier等
- ○ MPI(*2)

■ Process *n* - lines ← プロセス*n*について、行ごとのコスト分布情報
- ● Cost、Barrier等
- ○ MPI(*2)

■ Thread *m* - lines ← プロセス*n*のスレッド*m*について、行ごとのコスト分布情報
- ● Cost、Barrier等
- ○ MPI(*2)

■ 「Process *n*」の他の「Thread」について以下同様。

■ 他の「Process」について以下同様。

(*2) MPIコスト
- fippおよびfipppxコマンドで「-Impi」有効時に出力。

■ Lines profile（行コスト分布情報） [2/5]

■ 本例のサンプルプログラムとサンプルジョブスクリプトでは、全プロセスについて、以下のような表示となる（プロセス0および1は省略）。

※ 紙面の関係で、以下の図では実際の報告書の出力を加工している。

```
Lines profile (Total thread cost basis)
Application - lines
  Application and Process outputs the total value of the cost of each thread.
  Procedure outputs the total value of the line cost of each thread.
```

Cost	%	Operation (s)	Barrier	%	Line	
214	100.0000	2.1518	32	14.9533	--	Application
84	39.2523	0.8444	0	0.0000	56	sub1._OMP_1_
64	29.9065	0.6438	0	0.0000	49	sub1._OMP_1_
31	14.4860	0.3116	0	0.0000	57	sub1._OMP_1_
25	11.6822	0.2513	25	11.6822	59	sub1._OMP_1_
8	3.7383	0.0806	7	3.2710	52	sub1._OMP_1_
2	0.9346	0.0201	0	0.0000	25	main._PRL_1_

← 情報集計レベル

↑ 手続のコスト　↑ 演算時間(s)　↑ スレッド間同期待ちコスト (*1)　↑ 行番号　↑ 生成手続名（次スライドで説明します。）

(*1) スレッド並列プログラム以外の場合、本項目は「--」固定です。

■ Lines profile（行コスト分布情報） [3/5]

- スレッド並列プログラムの場合、並列化された部分を生成手続の情報として出力する。
- 生成手続名には、手続名の後ろに生成手続の種別に応じた識別子が付加される。

Lines profile (Total thread cost basis)
Application - lines

Cost	%	Operation (s)	Barrier	%	Line	
214	100.0000	2.1518	32	14.9533	--	Application
84	39.2523	0.8444	0	0.0000	56	sub1._OMP_1_
						（中略）
2	0.9346	0.0201	0	0.0000	25	main._PRL_1_

↑ 生成手続名

言語種別	生成手続の種別	生成手続名
Fortran	自動並列手続	手続名._PRL_数字_
	OpenMP並列手続	手続名._OMP_数字_
	TASK構文	手続名._TSK_数字_
C/C++（tradモード）	自動並列手続	手続名._PRL_数字
	OpenMP並列手続	手続名._OMP_数字
	TASK構文	手続名._TSK_数字
C/C++（clangモード）	OpenMP並列手続	手続名.omp_outlined._debug__数字 ※「_debug__」は、翻訳時オプション-gが指定されている場合に付加します。
	TASK構文	手続名.omp_task_entry.数字

■ Lines profile（行コスト分布情報） [4/5]

- Lines profileのコスト値をソースリストに転記した結果を右図に示す。

Lines profile (Total thread cost basis)
Application - lines

Cost	Barrier	Line	
214	32	--	Application
84	0	56	sub1._OMP_1_
64	0	49	sub1._OMP_1_
31	0	57	sub1._OMP_1_
25	25	59	sub1._OMP_1_
8	7	52	sub1._OMP_1_
2	0	25	main._PRL_1_

- 48行目のDO Iのループの上限がJであるため、スレッド間にインバランスがある。インバランスによるスレッド同期待ちのコストが52行目に現れている。
  - 52行目では、この行のコスト8のうち、スレッド同期待ちのコスト（Barrier欄）が7であることを示している。55行目と59行目の関係も同様。

```
Line    Cost
   6            PROGRAM MAIN
  23              DO J=1,N
  24                DO I=1,N
  25       2          X(I,J) = 0D0
  26                ENDDO
  27              ENDDO
  28              DO LL=1,50
  29                CALL SUB1(X,N)
  30              ENDDO
```

```
Line    Cost
  40            SUBROUTINE SUB1(X,N)
  45            !$OMP PARALLEL
  46            !$OMP DO
  47              DO J=JSTA,JEND
  48                DO I=1,J
  49      64          X(I,J) = 0D0
  50                ENDDO
  51              ENDDO
  52       8    !$OMP END DO
  53            !$OMP DO
  54              DO J=JSTA,JEND
  55                DO I=1,J
  56      84          X(I,J)=X(I,J)+1D0
  57      31        ENDDO
  58              ENDDO
  59      25    !$OMP END DO
  60            !$OMP END PARALLEL
```

■ Lines profile（行コスト分布情報） [5/5]

■ Lines profileの注意事項

- 通常は、実際に計算が行われた行番号にコストが表示される。
- コンパイラが最適化を行ってプログラムの内容が変わり、元のプログラムのどの行でサンプリングされたか不明になった場合、実際に計算が行われた行ではなく、その行を含むDOループのDO文やENDDO文などの箇所にコストが表示される。
- コンパイラの最適化レベルを低くすると、最適化があまり行われずプログラムの内容が変わらないため、コストが実際に実行した行番号の所に表示される可能性が高くなる。
  - ただし最適化レベルが高い場合とコスト分布が変わってしまう可能性がある。

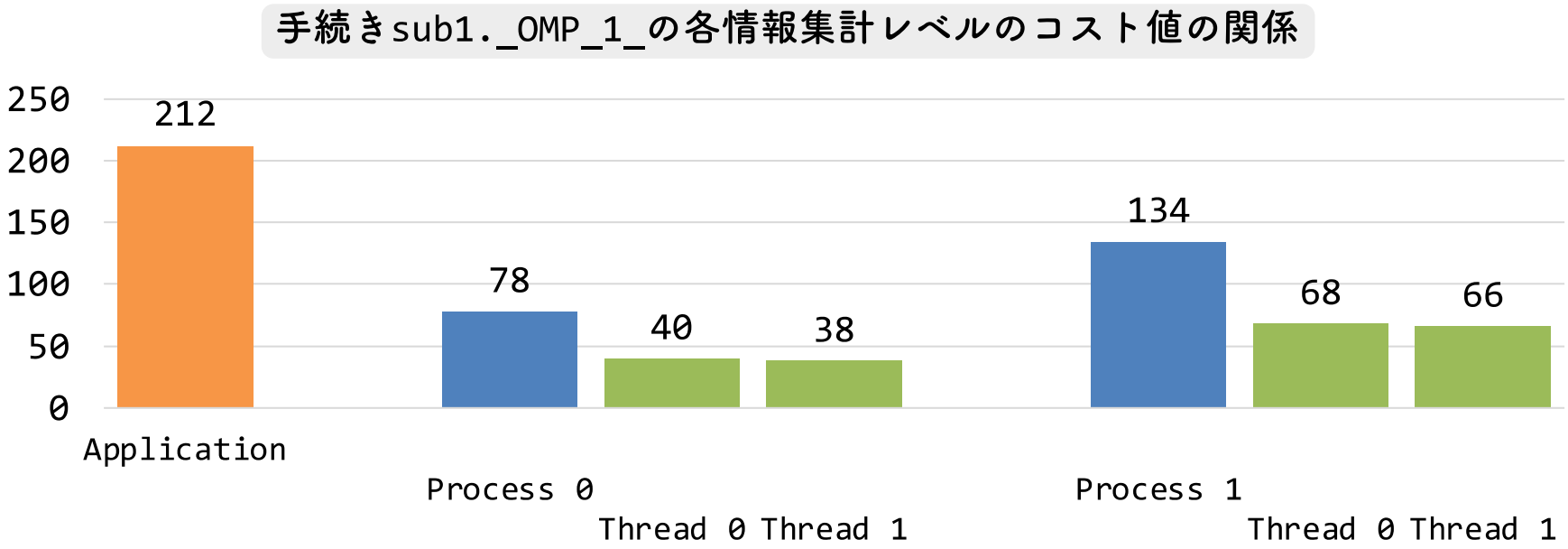
```
Line    Cost
  40            SUBROUTINE SUB1(X,N)
  45            !$OMP PARALLEL
                  ...
  53            !$OMP DO
  54              DO J=JSTA,JEND
  55                DO I=1,J
  56      84          X(I,J)=X(I,J)+1D0
  57      31        ENDDO
  58              ENDDO
                  ...
```

■ 各情報集計レベルのコスト値の関係

- **Total thread cost basis**（合計スレッドコスト基準）では、
  Lines profileの場合、各行について、
  - プロセス内の全スレッドのコストの合計がそのプロセスのコスト、
  - 全プロセスのコストの合計が「Application」のコストという関係がある。

  （Procedures profileの場合は各手続きについて、Loops profileの場合は各ループについて、同様な関係がある。）

- Processのコスト値はプロセスがどの程度の負荷を受け持っているのかを示すので、これらを比較すればプロセス間の負荷バランスの目安になる。



内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略
  - 基本プロファイラの使用方法
    - プロファイルデータの計測
    - プロファイル結果の出力
    - プロファイル結果出力（報告書）の内容
    - fipppxコマンドの-bmaxオプション
  - 詳細プロファイラ（テキスト形式）の使用方法
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

■ プロファイル結果出力の際の-bmaxオプション

■ ⑫のfipppxコマンドにおいて、[9]の「-bmax」を指定すると、[10]のファイルに出力されるコスト情報を最大スレッドコスト基準で出力する。

```
fipppx -A -bmax -pall -Icpupa,balance,call,src -d PD -o fipp_bmax.txt
```

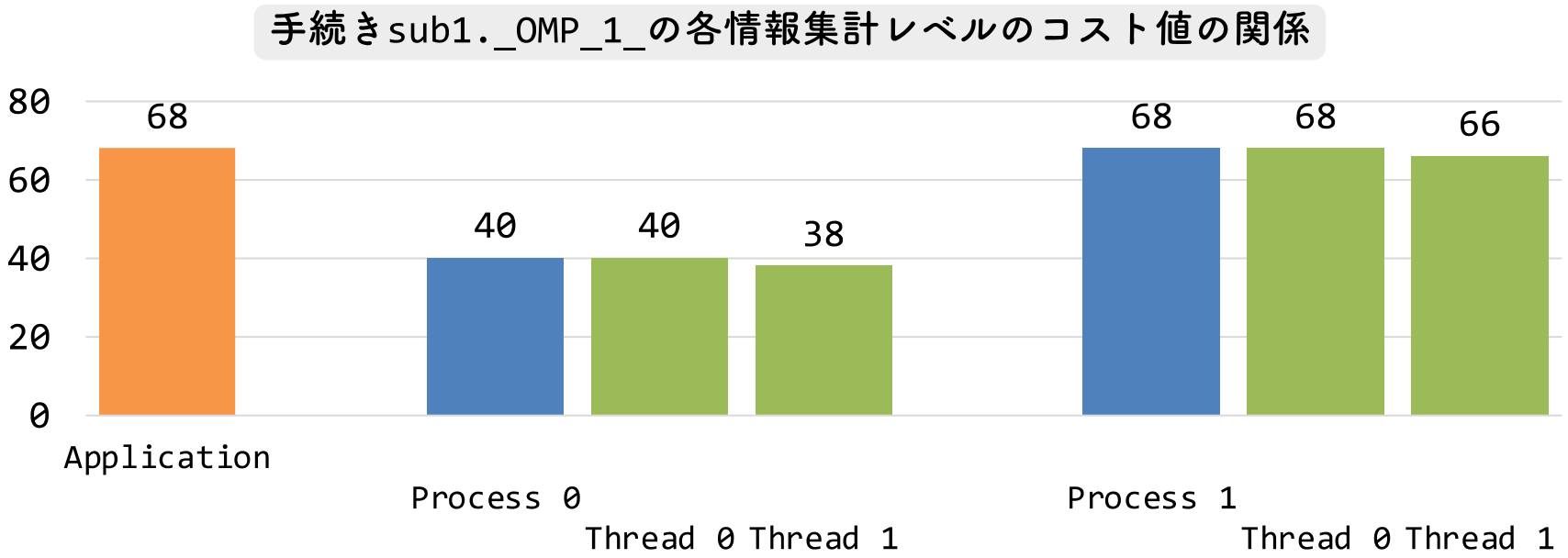
⑫

[9] [10]

■ 「-bmax」指定では、Lines profileの場合、各行について、
- プロセス内の全スレッドのコストの最大値がそのプロセスのコスト、
- プログラム内の全プロセスのコストの最大値がそのプログラムのコストとなる。

（Procedures profileの場合は各手続きについて、Loops profileの場合は各ループについて、同様な関係。）

■ この機能によって、どのプロセス・スレッドの寄与によって経過時間が決まっているかが分かりやすくなり、チューニングで注目すべき処理の目星がつく。



■ -bmaxオプションの注意事項

■ [9]の「-bmax」を指定した場合、

- ⑫で-Icpupaを指定してもPerformance monitor event:statistics（CPU動作状況）は出力されない。
- ⑫で-Icallを指定してもCall graph（コールグラフ情報）は出力されない。
- プロファイルデータ計測環境情報においてCPU周波数の値は出力されない。

```
fipppx -A –bmax -pall -Icpupa,balance,call,src -d PD -o fipp_bmax.txt      ⑫
          [9]                                                  [10]
```

■ 「-bmax」オプションは「-u」オプション(*4)と同時に指定することはできない。

(*4) 「-u」オプションはスレッド並列プログラムの生成手続名の出力方法を指定する。

- -uオプションを指定した場合、手続と生成手続のコスト情報を合算し手続名として出力する。
- -uオプションを指定しない場合、手続と生成手続を分けて出力する。

内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略
  - 基本プロファイラの使用方法
  - 詳細プロファイラ（テキスト形式）の使用方法
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略
  - 基本プロファイラの使用方法
  - 詳細プロファイラ（テキスト形式）の使用方法
    - プロファイルデータの計測
    - プロファイル結果の出力
    - プロファイル結果出力（報告書）の内容
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

■ サンプルプログラム [1/3]

- 詳細プロファイラは、プログラム内の指定した部分のプロファイルを行う。
- 以下のサンプルプログラム（スレッド並列＋プロセス並列）について、
  - 並列化したループ計算を行うSUB1と、
  - MPI通信を行うSUB2

  のプロファイルを行う場合で説明する。

```
PROGRAM MAIN
  ...
  CALL FAPP_START("SUB1",1001,0)
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL  FAPP_STOP("SUB1",1001,0)

  CALL FAPP_START("SUB2",1002,0)
  CALL SUB2(X,N)
  CALL  FAPP_STOP("SUB2",1002,0)
  ...
END PROGRAM MAIN
```

```
SUBROUTINE SUB1(X,N)
  ...
!$OMP PARALLEL
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = 0D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = X(I,J) + 1D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE SUB1
```

```
SUBROUTINE SUB2(X,N)
  ...
!$OMP PARALLEL
 DO J=1,N
!$OMP MASTER
   CALL MPI_BCAST(X(1,J),N,～)
!$OMP END MASTER
 ENDDO
!$OMP END PARALLEL
END SUBROUTINE SUB2
```

■ サンプルプログラム [2/3]

■ 測定したい部分の前と後に、それぞれ
①：CALL FAPP_START(～)、
②：CALL FAPP_STOP(～)
を指定する（引数は次スライドで説明）。

```
PROGRAM MAIN
  ...
  CALL FAPP_START("SUB1",1001,0)        ①
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL  FAPP_STOP("SUB1",1001,0)        ②

  CALL FAPP_START("SUB2",1002,0)        ①
  CALL SUB2(X,N)
  CALL  FAPP_STOP("SUB2",1002,0)        ②
  ...
END PROGRAM MAIN
```

```
SUBROUTINE SUB1(X,N)
  ...
!$OMP PARALLEL
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = 0D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = X(I,J) + 1D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE SUB1
```

```
SUBROUTINE SUB2(X,N)
  ...
!$OMP PARALLEL
 DO J=1,N
!$OMP MASTER
   CALL MPI_BCAST(X(1,J),N,～)
!$OMP END MASTER
 ENDDO
!$OMP END PARALLEL
END SUBROUTINE SUB2
```

■ サンプルプログラム [3/3]

- SUB1について、①②をDO LLのループ内のCALL SUB1の直前直後に指定することも可能であるが、オーバーヘッドを少なくするため、DO LLのループの外側に指定した。
- ①②の第1引数"SUB1"と"SUB2"、第2引数1001と1002は、報告書内の目印として使用される。
- ①②の第3引数はfappによる計測対象の起動レベルを指定する。fappコマンドの-Lオプションで使用される。通常は0を指定する。

```
PROGRAM MAIN
  ...
  CALL FAPP_START("SUB1",1001,0)   ①
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL  FAPP_STOP("SUB1",1001,0)   ②

  CALL FAPP_START("SUB2",1002,0)   ①
  CALL SUB2(X,N)
  CALL  FAPP_STOP("SUB2",1002,0)   ②
  ...
END PROGRAM MAIN
```

■ 参考：C/C++（tradモード、clangモード）の場合の指定方法

■ 1 のインクルード（"fj_tool/fapp.h"）を指定する。

■ 測定したい部分の前と後に、それぞれ

2 ：fapp_start(～);、

3 ：fapp_stop(～);

を指定する。

```
#include "fj_tool/fapp.h"                    1

int main(int argc, char *argv[])
{
  ...
  fapp_start("SUB1",1001,0);                 2
  測定したい部分
  fapp_stop("SUB1",1001,0);                  3
  ...
}
```

■ コンパイル・リンク

- 前スライドのFortranサンプルプログラム（スレッド並列＋プロセス並列）を、以下のようにコンパイル・リンクする。

```
mpifrtpx -Kfast,parallel,openmp -o a.exe  a.F90
```

- プロファイラ用のライブラリをリンクするためにはリンク時に-Nfjprofの指定が必要であるが、-Nfjprofはデフォルトで有効である。

- 参考
  - C/C++ tradモードの場合、リンク時に-Nfjprofを指定する（デフォルトで有効）。
  - C/C++ clangモードの場合、リンク時に-ffj-fjprofを指定する（デフォルトで有効）。

■ ジョブスクリプト [1/2]

- 本例では、（③④により）プロセス数：2、（⑤⑥により）プロセス当たりのスレッド数：2で、ジョブを投入する。
- ⑦でプログラムの実行が開始し、[1]のfappコマンドによってプロファイルが行われる。
  - [2]の「-C」の指定は必須である。
  - [4]の「-Hevent=」には「Statistics」を指定する（CPU性能解析情報の計測を意味する）。

```
#!/bin/bash
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=20:00"
#PJM -L "node=1"                        ③
#PJM --mpi "max-proc-per-node=2"        ④
#PJM --mpi "rank-map-bychip"
#PJM -S
export PLE_MPI_STD_EMPTYFILE=off
export PARALLEL=2                       ⑤
export OMP_NUM_THREADS=2                ⑥
fapp -C -d PD -Hevent=Statistics  mpiexec -stdout stdout -stderr stderr ./a.exe   ⑦
```

[1] [2] [3] [4]

■ ジョブスクリプト [2/2]

- ジョブが終了すると、プロセス（本例では0、1）ごとに、プロファイルデータが作成される。
- プロファイルデータは、[3]の「-d」で指定したディレクトリ（本例ではPD）の下のファイルに入る。
  - 指定したディレクトリが存在しない場合、新規にディレクトリが作成される。
  - 指定したディレクトリが存在する場合、そのディレクトリは空でなければならない。

⚠ fappコマンドを使用する場合、環境変数FLIB_FASTOMPにTRUEを指定して下さい（デフォルトはTRUE）。

```
#!/bin/bash
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=20:00"
#PJM -L "node=1"                          ③
#PJM --mpi "max-proc-per-node=2"          ④
#PJM --mpi "rank-map-bychip"
#PJM -S
export PLE_MPI_STD_EMPTYFILE=off
export PARALLEL=2                         ⑤
export OMP_NUM_THREADS=2                  ⑥
fapp -C -d PD -Hevent=Statistics  mpiexec -stdout stdout -stderr stderr ./a.exe   ⑦
 [1] [2] [3]       [4]
```

内容

■ プロファイル結果の出力

- 本例でのディレクトリPD下に作成されたファイルはバイナリファイルなのでそのままでは読めない。
- ジョブ終了後、ログインノードで[5]のfapppxコマンドを実行する(*1)。
  - [6]の「-A」の指定は必須。
  - [7]の指定は、[11]で指定する出力ファイルの形式をテキスト形式と指定する。
  - [10]の「-d」の引数には、fappコマンド実行時に指定したディレクトリを指定する。

```
fapppx -A -ttext -Icpupa,mpi -d PD -o fapp_result.txt        ⑧
 [5]  [6]  [7]     [8]  [9]  [10] [11]
```

(*1) ログインノードで[5][6]の「fapppx –A」の代わりに、計算ノードで「fapp –A」と指定して実行することもできる。

- [10]で指定したディレクトリ（本例ではPD）にあるファイルが、読める形式（以下、報告書と呼ぶ）に変換され、ファイル（本例ではfapp_result.txt）に書き出される。
- [8]と[9]は後述する。

内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略
  - 基本プロファイラの使用方法
  - 詳細プロファイラ（テキスト形式）の使用方法
    - プロファイルデータの計測
    - プロファイル結果の出力
    - プロファイル結果出力（報告書）の内容
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

■ プロファイル結果出力（報告書）の内容

報告書は以下の項目で構成されている。

```
■ Fujitsu Advanced Performance Profiler
Version x.y.z
```

← プロファイルデータ計測環境情報

```
■ Basic profile
```

← 時間統計情報

```
□ MPI profile
```

← MPI通信コスト情報
- `fapp`および`fapppx`コマンドで「`-Impi`」有効時に出力。

```
□ Performance monitor event : statistics
```

← CPU性能解析情報
- `fapp`および`fapppx`コマンドで「`-Icpupa`」有効時に出力。

■ プロファイル結果出力（報告書）の内容

報告書は以下の項目で構成されている。

```
■ Fujitsu Advanced Performance Profiler
Version x.y.z
```
← プロファイルデータ計測環境情報

```
■ Basic profile
```
← 時間統計情報

```
□ MPI profile
```
← MPI通信コスト情報
- fappおよびfapppxコマンドで「-Impi」有効時に出力。

```
□ Performance monitor event : statistics
```
← CPU性能解析情報
- fappおよびfapppxコマンドで「-Icpupa」有効時に出力。

■ MPI profile（MPI通信コスト情報）

■ 本項目はfappおよびfapppxコマンドで「-Impi」有効時に出力される。

■ 本項目は以下の各項目で構成されている。

□ MPI profile
- ■ Application ← 全プロセスについて
  - ■ 各（区間名、詳細番号）の、各MPIルーチンについて、Elapsed(s)、Wait(s)等(*1)
- ■ Process *n* ← プロセス *n* について
  - ■ 各（区間名、詳細番号）の、各MPIルーチンについて、Elapsed(s)、Wait(s)等(*1)
- ■ 他の「Process」について以下同様

(*1) 表示される項目の一覧は次スライドで示します。

■ MPI profileの「Application」欄

- Elapsed(s)欄とWait(s)欄には、各計測区間ごとに、それぞれ経過時間と待ち時間について、各MPIルーチンの、プロセス間の平均値、最大値、最小値が表示される。
- 本例ではサブルーチンSUB2内のmpi_bcast_について、経過時間や待ち時間等について、プロセス間の平均値、最大値、最小値が表示される。これにより通信時間のプロセス間インバランスの検討に役立つ。

※ 紙面の関係で、以下の図では実際の報告書の出力を加工している。

MPI profile

Application

経過時間(s) ↓　待ち時間(s) ↓　平均メッセージ長(Byte) ↓　MPIライブラリの呼出し回数 ↓

L1-L2: メッセージ長が L1 バイト以上 L2 バイト未満の場合の、MPIライブラリの呼出し回数 ↓

Kind	Elapsed(s)	Wait(s)	Byte	Call	(0-4K	4K-64K	64K-1024K	1024KByte-)	
	25.2304	0.4011	----	20000	0	0	20000	0	SUB2 1002
AVG	12.6152	0.2005	120000.0	10000.0	0.0	0.0	10000.0	0.0	mpi_bcast_
MAX	12.7846	0.3695	160000.0	10000	0	0	10000	0	
MIN	12.4458	0.0316	80000.0	10000	0	0	10000	0	

■ プロファイル結果出力（報告書）の内容

報告書は以下の項目で構成されている。

```
■ Fujitsu Advanced Performance Profiler
Version x.y.z
```
← プロファイルデータ計測環境情報

```
■ Basic profile
```
← 時間統計情報

```
□ MPI profile
```
← MPI通信コスト情報
- `fapp`および`fapppx`コマンドで「`-Impi`」有効時に出力。

```
□ Performance monitor event : statistics
```
← CPU性能解析情報
- `fapp`および`fapppx`コマンドで「`-Icpupa`」有効時に出力。

■ Performance monitor event : statistics（CPU性能解析情報） [1/3]

■ 本項目はfappおよびfapppxコマンドで「-Icpupa」有効時に出力される。

■ 本項目は以下の各項目で構成されている。

□ Performance monitor event : statistics

■ Application ← 全プロセスについて
  ■ 各（区間名、詳細番号）について、Execution time(s)、GFLOPS等(*1)

■ Process *n* ← プロセス *n* について
  ■ 各（区間名、詳細番号）について、Execution time(s)、GFLOPS等(*1)

■ Thread *m* ← プロセス *n* の各スレッドについて
  ■ 各（区間名、詳細番号）について、Execution time(s)、GFLOPS等(*1)

■ 「Process *n*」の他の「Thread」について以下同様

■ 他の「Process」について以下同様

(*1) 表示される項目の一覧は次スライドで示します。

■ Performance monitor event : statistics（CPU性能解析情報） [2/3]

■ 全プロセス、プロセスごと、スレッドごとについて、各区間ごとに以下の各種性能特性が表示される。

各種性能特性	意味	備考
Execution time(s)	計測対象区間の命令実行に要した時間（秒）。	
GFLOPS	1秒あたりに実行された浮動小数点演算数。	GFLOPS値はすべてActive elementとして算出しています(*1)。そのため、Inactive elementの多いプログラムでは本来の GFLOPS 値より高い値を出力します。
Floating-point peak ratio(%)	浮動小数点演算性能の理論値に対する実測値の比率（%）。	・浮動小数点演算性能の理論値は倍精度演算として算出しています。そのため、単精度や半精度の演算の場合、実際の割合の2倍や4倍の値を出力します。 ・fappコマンドによるプロファイルデータの計測で-Hmode=userオプションを指定した場合、この出力項目は「--」固定です。
Mem throughput(GB/s)	メモリースループット（GB/s）。	
Mem throughput peak ratio(%)	メモリースループットの理論値に対する実測値の比率（%）。	

(*1) IF文を含むループのSIMD化においてIF文内の計算はIF文の条件の真偽によらず演算数にカウントされる。

次のスライドへ続く

■ Performance monitor event : statistics（CPU性能解析情報） [3/3]

前のスライドからの続き

各種性能特性	意味	備考
Effective instruction	実行された命令の総数。	実行された命令の総数には、MOVPRFX命令(*2)を含みません。
Floating-point operation	実行された浮動小数点演算の総数。	
SIMD inst. rate(%)	実行された命令の総数に対するSIMD命令数の比率（%）。	
SVE operation rate(%)	実行された浮動小数点演算の総数に対するSVE演算数の比率（%）。	
IPC	1サイクルあたりに実行された命令数。	
GIPS	1秒あたりに実行された命令数。	

(*2) 積和演算命令の前置命令として、和をとるレジスタの内容を代入先レジスタへコピーする命令。

内容

- 「富岳」の概略
- プロファイラ
  - 【前提知識、予備知識】富士通コンパイラのコンパイルコマンド概略
  - プロファイラの使用手順概略
  - 基本プロファイラの使用方法
  - 詳細プロファイラ（テキスト形式）の使用方法
  - CPU性能解析レポートの使用方法
- 重要な最適化手法
- 【付録】参考文献

内容

■ CPU性能解析レポートの種別

- CPU性能解析レポートは、表示する情報の種類や、粒度によって、以下の四つの段階が用意されている。

レポート種別	プロファイラによる必要な計測回数	説明
単体レポート	1回	作成に必要な計測回数がもっとも少ない。実行時間、演算性能、メモリースループット、命令数についてのおおまかな情報を出力する。
簡易レポート	5回	手軽にCPU性能解析レポートを使用したい場合に推奨される。
標準レポート	11回	標準的なCPU性能解析レポート。通常の使用で推奨される。
詳細レポート	17回	もっとも詳細なCPU性能解析レポート。

○：すべての情報が出力される。
△：一部の情報が出力される。
－：情報が出力されない。

■ CPU性能解析レポートで参照可能な情報の一覧

表タイトル	表の概要	レポート種別			
		単体	簡易	標準	詳細
Information	計測環境の情報およびユーザーの指定内容を表示します。	○	○	○	○
Statistics	メモリースループット、命令数、演算数などCPUの動作状況に関係する情報を表示します。	△	○	○	○
Cycle Accounting	プログラムの実行時間の内訳を表示します。	－	△	○	○
Busy	プログラムのメモリー・キャッシュおよび演算パイプラインのビジー率に関する情報を表示します。	－	△	△	○
Cache	キャッシュミスに関する情報を表示します。	－	△	○	○
Instruction	命令ミックスに関する情報を表示します。	－	△	△	○
FLOPS	浮動小数点演算に関する情報を表示します。	－	△	○	○
Extra	ギャザー命令の内訳、および命令ミックスに含まれない命令に関する情報を表示します。	－	－	－	○
Hardware Prefetch Rate (%) (/Hardware Prefetch)	ハードウェアプリフェッチの内訳を表示します。	－	－	－	○
Data Transfer CMGs	ユーザーが指定したCMGに対する、全CMG、メモリー、Tofu、およびPCI間のスループット情報を表示します。	－	－	－	○
Power Consumption (W)	コア、L2キャッシュ、およびメモリーの消費電力を表示します。	－	－	○	○

内容

■ サンプルプログラム [1/3]

- CPU性能解析レポートは、プログラム内の指定した部分のプロファイルを行う。
- 詳細プロファイラの節で用いたサンプルプログラム（スレッド並列＋プロセス並列）のうち、
  - 並列化したループ計算を行うSUB1と、
  - MPI通信を行うSUB2

  のプロファイルを行う場合で説明する。

```
PROGRAM MAIN
  ...
  CALL FAPP_START("SUB1",1001,0)
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL  FAPP_STOP("SUB1",1001,0)

  CALL FAPP_START("SUB2",1002,0)
  CALL SUB2(X,N)
  CALL  FAPP_STOP("SUB2",1002,0)
  ...
END PROGRAM MAIN
```

```
SUBROUTINE SUB1(X,N)
  ...
!$OMP PARALLEL
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = 0D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = X(I,J) + 1D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE SUB1
```

```
SUBROUTINE SUB2(X,N)
  ...
!$OMP PARALLEL
 DO J=1,N
!$OMP MASTER
   CALL MPI_BCAST(X(1,J),N,～)
!$OMP END MASTER
 ENDDO
!$OMP END PARALLEL
END SUBROUTINE SUB2
```

■ サンプルプログラム [2/3]

■ 測定したい部分の前と後に、それぞれ
①：CALL FAPP_START(～)、
②：CALL FAPP_STOP(～)
を指定する（引数は詳細プロファイラの場合と同じ）。

```
PROGRAM MAIN
  ...
  CALL FAPP_START("SUB1",1001,0)        ①
  DO LL=1,50
    CALL SUB1(X,N)
  ENDDO
  CALL  FAPP_STOP("SUB1",1001,0)        ②

  CALL FAPP_START("SUB2",1002,0)        ①
  CALL SUB2(X,N)
  CALL  FAPP_STOP("SUB2",1002,0)        ②
  ...
END PROGRAM MAIN
```

```
SUBROUTINE SUB1(X,N)
  ...
!$OMP PARALLEL
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = 0D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = X(I,J) + 1D0
    ENDDO
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE SUB1
```

```
SUBROUTINE SUB2(X,N)
  ...
!$OMP PARALLEL
 DO J=1,N
!$OMP MASTER
   CALL MPI_BCAST(X(1,J),N,～)
!$OMP END MASTER
 ENDDO
!$OMP END PARALLEL
END SUBROUTINE SUB2
```

■ コンパイル・リンク

- 前スライドのFortranサンプルプログラム（スレッド並列＋プロセス並列）を、以下のようにコンパイル・リンクする。

```
mpifrtpx -Kfast,parallel,openmp -o a.exe  a.F90
```

- プロファイラ用のライブラリをリンクするためにはリンク時に-Nfjprofの指定が必要であるが、-Nfjprofはデフォルトで有効である。

- 参考
  - C/C++ tradモードの場合、リンク時に-Nfjprofを指定する（デフォルトで有効）。
  - C/C++ clangモードの場合、リンク時に-ffj-fjprofを指定する（デフォルトで有効）。

■ ジョブスクリプト

- 本例では、⑦により⑧のループの繰り返し数を17と指定し、⑨で-Hevent=pa1から-Hevent=pa17を指定した17回の計測を実施して「詳細レポート」を作成する。
- また、本例では（③④により）プロセス数：2、（⑤⑥により）プロセス当たりのスレッド数：12で、ジョブを投入する。
- ⑨でプログラムの実行が開始し、[1]のfappコマンドによってプロファイルが行われる。
  - [2]と[3]の「-C -Hevent=～」の指定は必須である。

```
#!/bin/bash
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=20:00"
#PJM -L "node=1"                    ③
#PJM --mpi "max-proc-per-node=2"    ④
#PJM --mpi "rank-map-bychip"
#PJM -S
export PLE_MPI_STD_EMPTYFILE=off
export PARALLEL=12                  ⑤
export OMP_NUM_THREADS=12           ⑥
LD=./a.exe
N=17                ⑦
for i in `seq 1 ${N}`; do     ⑧
  fapp -C -Hevent=pa${i} -d ./pa${i} mpiexec -stdout stdout -stderr stderr ${LD}    ⑨
done
```

[1] [2] [3] [4]

内容

■ プロファイル結果の出力

- ⑫のfapppxコマンドを実行するスクリプトを作成する。本例ではconvert_cpupa_to_csv.shと命名した。
  - [6]、[7]、[8]の指定「-A -Icpupa -tcsv」は必須である。
  - [8]の指定は、[10]で指定する出力ファイルの形式をcsv形式と指定する。
  - [9]の「-d」の引数には、fapp -Cコマンド実行時に指定したディレクトリを指定する。
  - 本例では「詳細レポート」を作成するために、⑩により⑪のループの繰り返し数を17と指定し、⑫でディレクトリ./pa1から./pa17にあるプロファイルデータを使用してpa1.csv～pa17.csvの17個のcsvファイルを出力する。
  - 計測時の-Hevent=pa1～-Hevent=pa17に対応して、[10]のcsvファイルの名前はpa1.csv～pa17.csv固定。

```
（ログインノード）$ cat ./convert_cpupa_to_csv.sh
#!/bin/bash
N=17                                                    ⑩
for i in `seq 1 ${N}`; do                               ⑪
  fapppx -A -Icpupa -tcsv -d ./pa${i} -o pa${i}.csv     ⑫
done
```

[5] [6] [7] [8] [9] [10]

- データ計測のジョブ終了後、ログインノードで上記スクリプトを⑬のように実行する。

```
（ログインノード）$ ./convert_cpupa_to_csv.sh
```
⑬

- ⑬のスクリプト実行により、pa1.csv～pa17.csvの17個のcsvファイルが出力される。

```
（ログインノード）$ ls *.csv
pa1.csv  pa3.csv  pa5.csv  pa7.csv  pa9.csv   pa11.csv  pa13.csv  pa15.csv  pa17.csv
pa2.csv  pa4.csv  pa6.csv  pa8.csv  pa10.csv  pa12.csv  pa14.csv  pa16.csv
```
⑭

内容

■ CPU性能解析レポートの作成

- 「富岳」のポータルサイトからCPU性能解析レポートファイルを以下のようにダウンロードする。
  - 以下のURLにログインする。

    マニュアル | スーパーコンピュータ「富岳」
    https://www.fugaku.r-ccs.riken.jp/docs/manuals_r01

  - このページ内の「言語マニュアル」から必要な「言語環境のバージョン」のリンクをクリックする。

    言語マニュアル
    ● x.y.z tcsds-l.m.n

  - 「言語マニュアル」ページ内の以下の「プログラミング支援ツール」から「● CPU性能解析レポートファイル」をダウンロードする（以下ではこのファイルをcpu_pa_report.xlsmと呼ぶ）。

    プログラミング支援ツール
    ● CPU性能解析レポートファイル

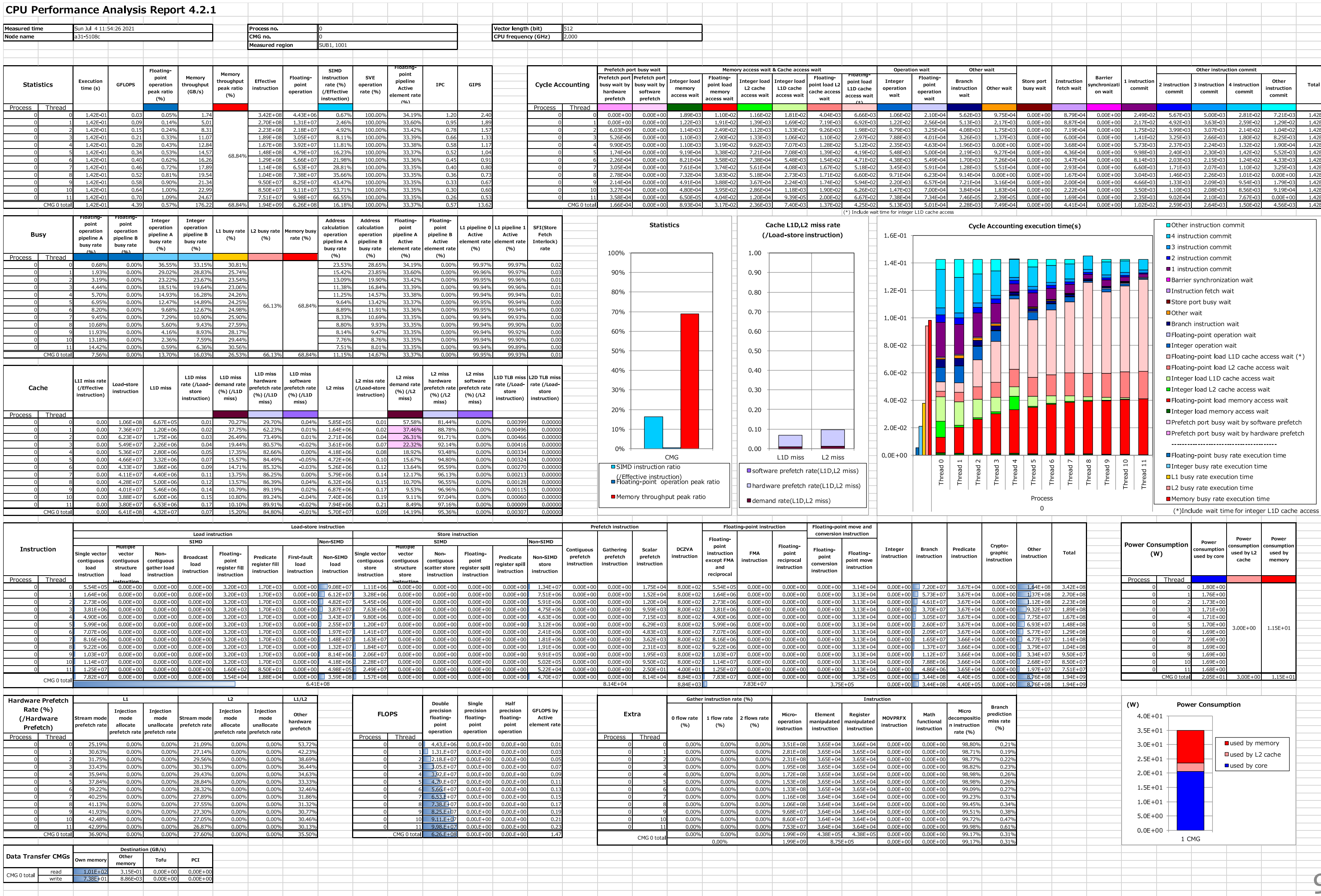
- fapppxコマンドの実行により作成したファイルpa1.csv～pa17.csvを「富岳」のログインノードから自分のPCに転送し、適当なフォルダ（例えば「prof」）の中に保存する。

- CPU性能解析レポートファイルcpu_pa_report.xlsmを「prof」にコピーする。

- cpu_pa_report.xlsmを起動する。CPU性能解析レポートに出力する内容を選択するためのダイアログが表示される。必要な情報を入力する。

内容

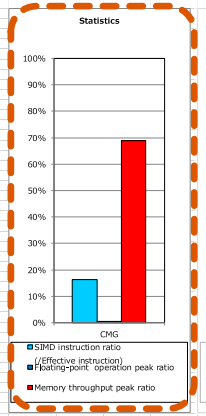
■ CPU性能解析レポートの内容概略［1/6］

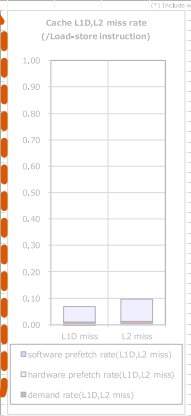


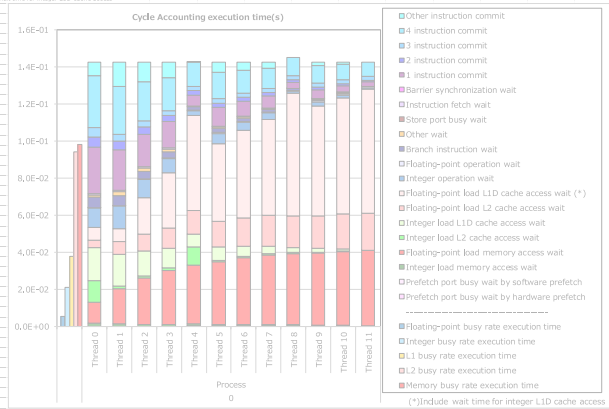
■ CPU性能解析レポートの内容概略 [2/6]

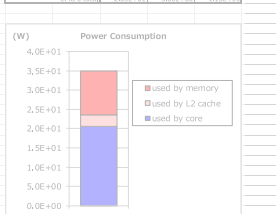
CPU Performance Analysis Report 4.2.1

Statistics：メモリースループット、命令数、演算数等CPU動作状況を表示。

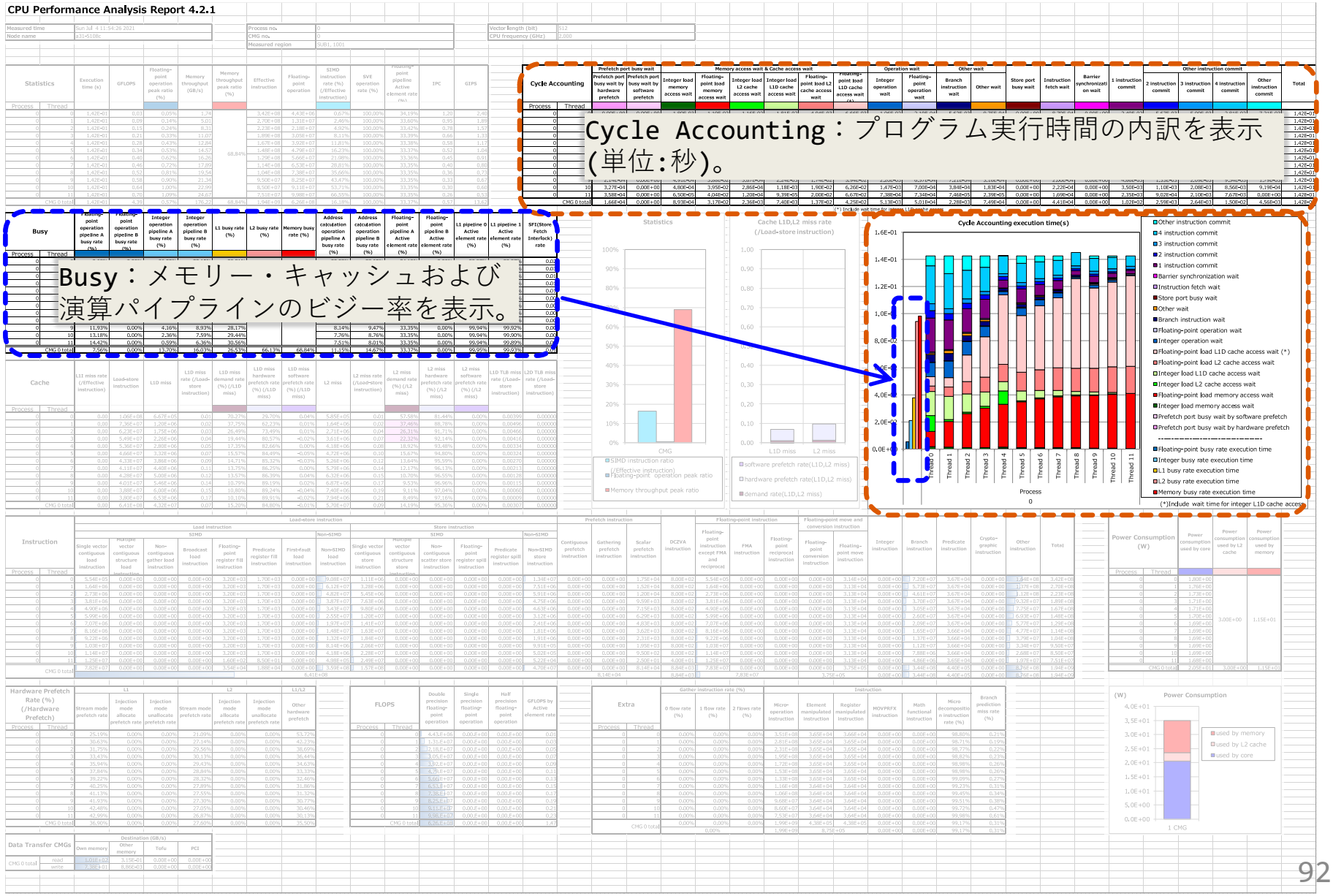






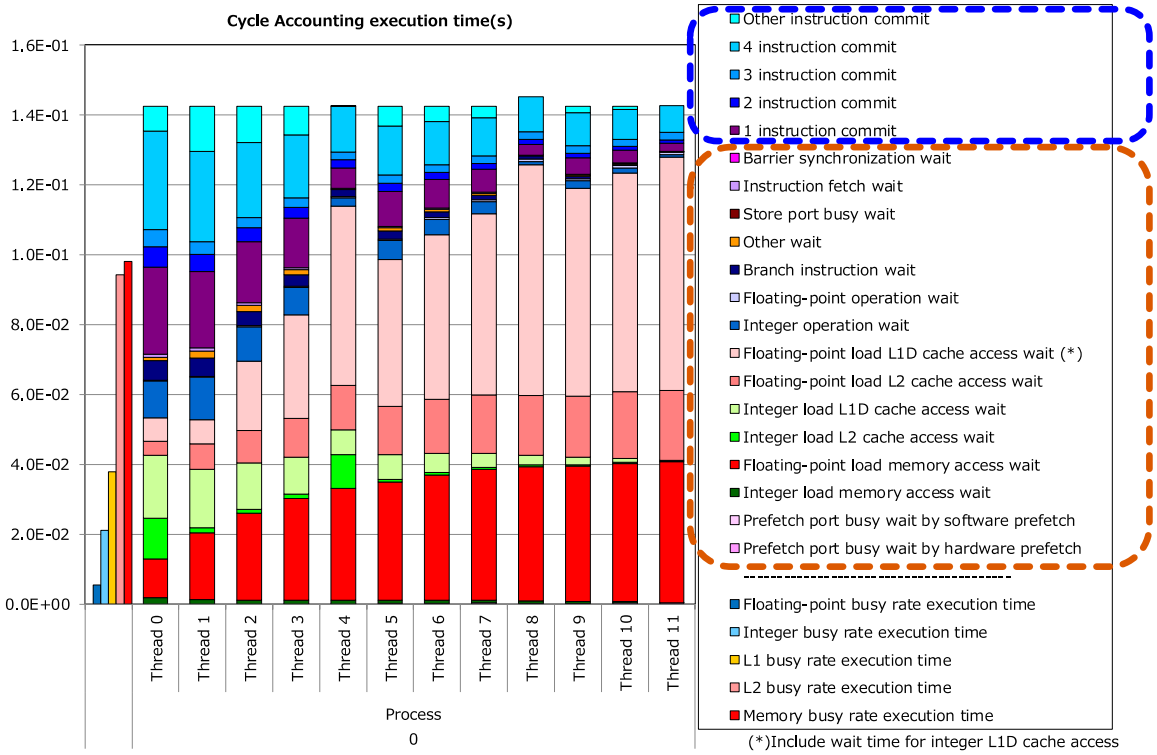


■ CPU性能解析レポートの内容概略 [3/6]



■ Cycle Accountingについて補足 [1/3]

- Cycle Accountingでは、計測対象区間の実行に要した総時間（CPUサイクル数）をCPUの動作状態で分類してグラフ化する。そのグラフからCPU内のボトルネックが把握できるので、詳細な性能分析やチューニングを行うことができる。
- グラフの主に青色系部分：1サイクルでN命令コミット（N命令実行）した時間
  - 演算が実行されている時間。
- グラフの主に赤色系部分：各種要因で命令がストールした時間（待ち時間）
  - 演算待ち、L1およびL2キャッシュアクセス待ち、メモリアクセス待ち等の時間。

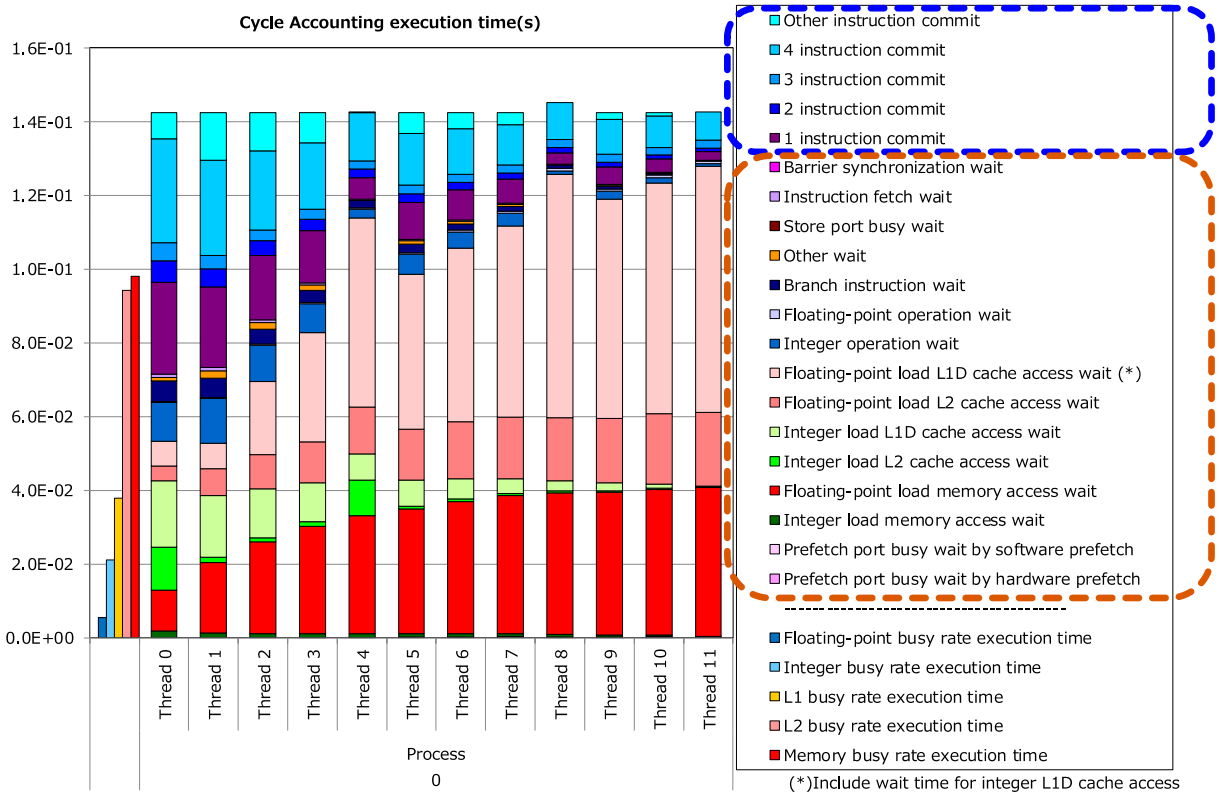


■ Cycle Accountingについて補足 [2/3]

- 本プログラム例SUB1では!$OMP DOによりJループを各スレッドにブロック分割でループスライスしている。Iループの上限はJとなっている。
- そのため、各スレッドの演算量にスレッド間インバランスがあり、それがコミット時間と待ち時間双方のインバランスとして現れている。

プログラム例（改善前）

```
SUBROUTINE SUB1(X,N)
  ...
!$OMP DO
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = X(I,J) + 1D0
    ENDDO
  ENDDO
```

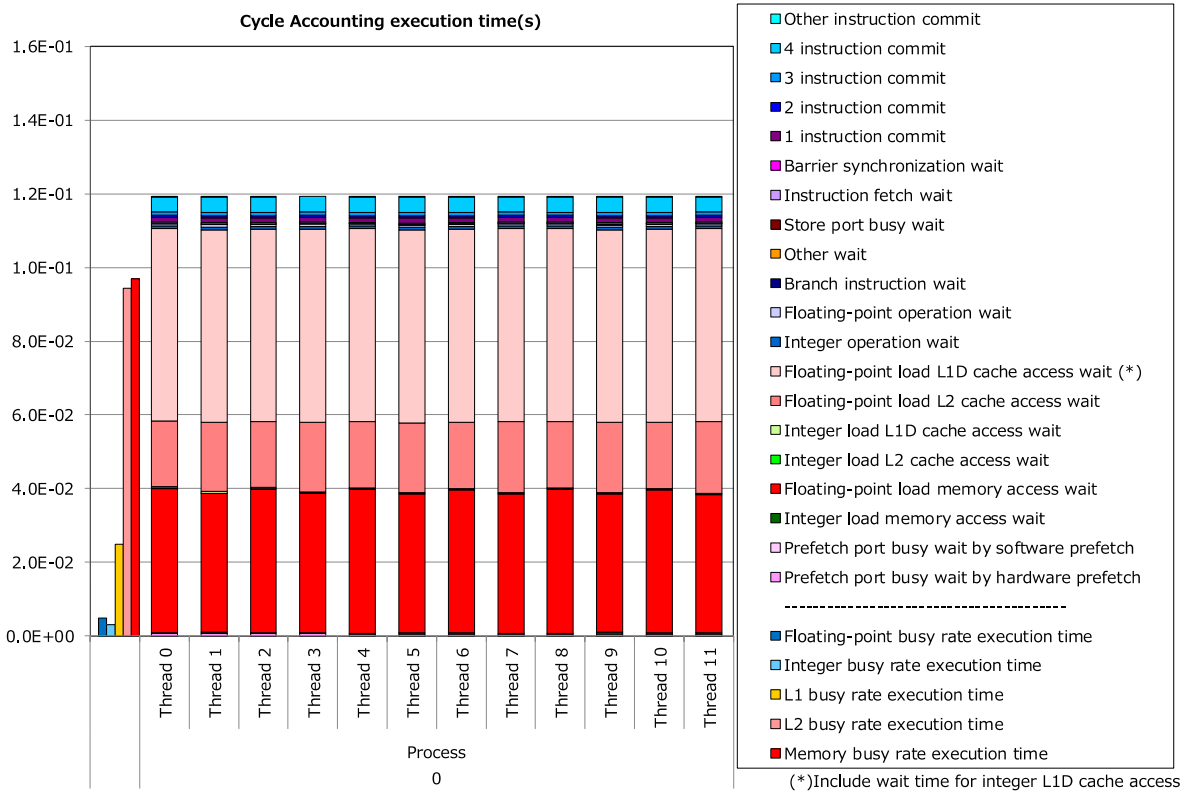


■ Cycle Accountingについて補足 [3/3]

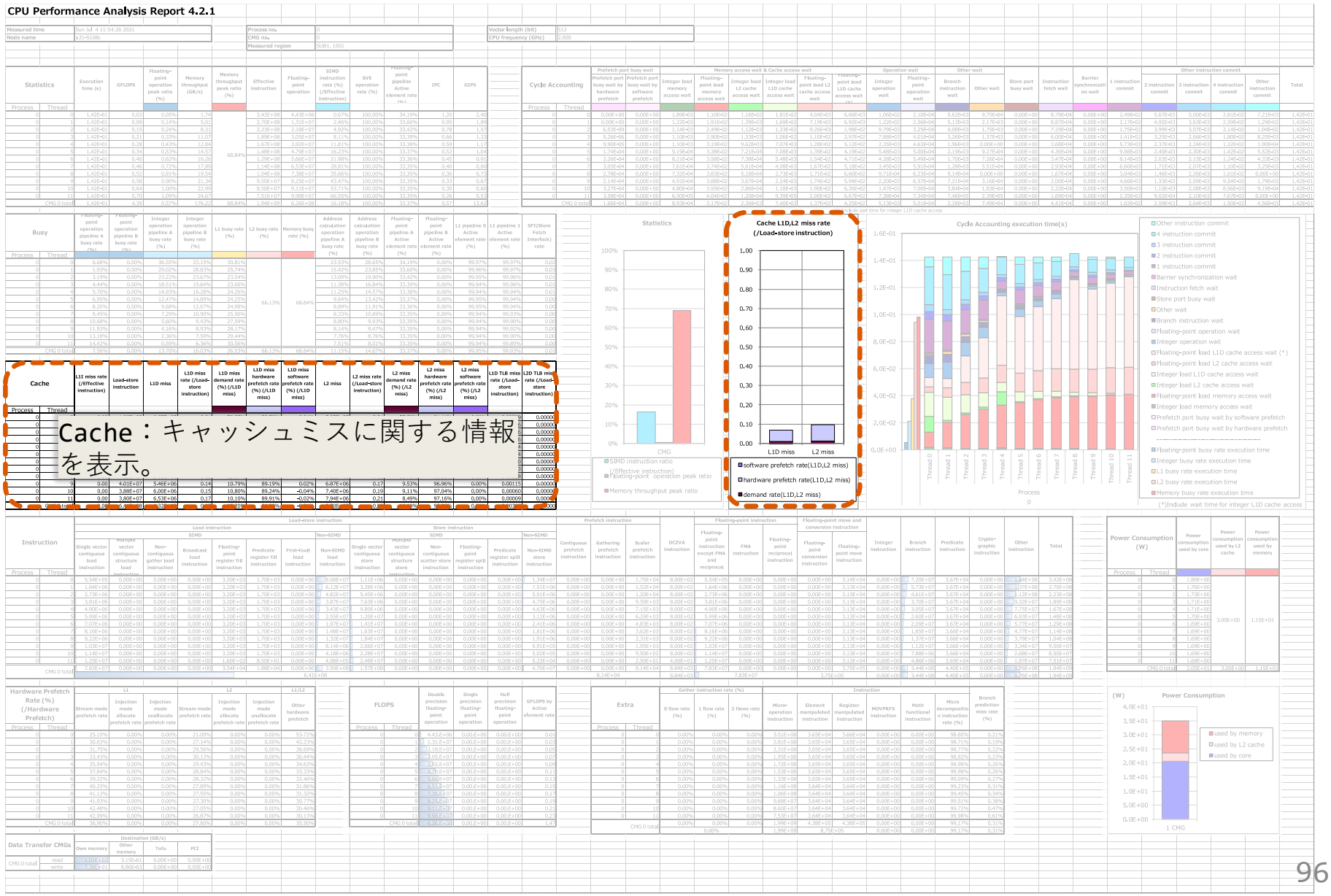
- !$OMP DOに続いてSCHEDULE(STATIC,1)を指定してサイクリック分割でループスライスすることで、スレッド間のインバランスを軽減できることがある。本改善例では以下の図のようにインバランスが大幅に軽減されていることがわかる。
- このようにCPU性能解析レポートはプログラムの問題点や改善策試行結果の視覚的な把握に役立つ。

プログラム例（改善後）

```
SUBROUTINE SUB1(X,N)
  ...
!$OMP DO SCHEDULE(STATIC,1)
  DO J=JSTA,JEND
    DO I=1,J
      X(I,J) = X(I,J) + 1D0
    ENDDO
  ENDDO
```



■ CPU性能解析レポートの内容概略 [4/6]

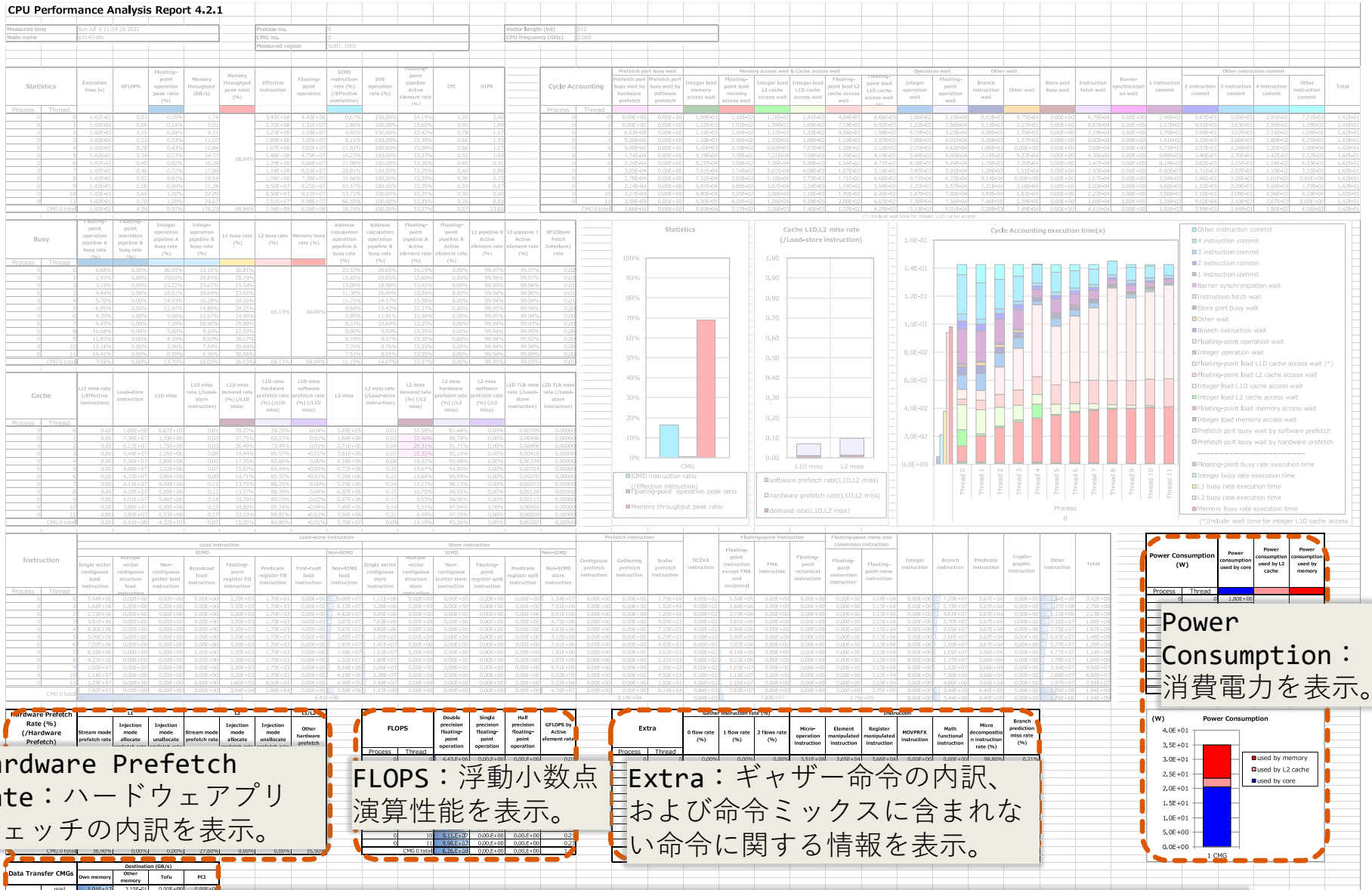


■ CPU性能解析レポートの内容概略 [5/6]

CPU Performance Analysis Report 4.2.1

Instruction：命令ミックスに関する情報を表示。

■ CPU性能解析レポートの内容概略 [6/6]



Power Consumption：消費電力を表示。

Hardware Prefetch Rate：ハードウェアプリフェッチの内訳を表示。

FLOPS：浮動小数点演算性能を表示。

Extra：ギャザー命令の内訳、および命令ミックスに含まれない命令に関する情報を表示。

Data Transfer CMGs：ユーザーが指定したCMGに対するスループット情報を表示。

内容

■ Cycle Accountingに基づくチューニングの指針

実行時間内訳	内訳の各要素が多い場合に考えられる原因	チューニングの指針	主な最適化手法
命令コミット	演算の命令数が多い。	命令数を削減する。	指示行追加(*1)によるSIMD化の促進、共通式の除去、不変式の移動
演算待ち	命令スケジューリングが悪い。	ソフトウェアパイプライニングを促進する。	指示行追加(*1)によるソフトウェアパイプライニングの促進、ループ分割
L1キャッシュアクセス待ち	レジスタのスピル・フィル（レジスタ溢れ）が多発している。	使用するレジスタ数を削減する。	ループ分割
L2キャッシュアクセス待ち	L1キャッシュミスが多発している。	L1キャッシュ利用の効率化を図る。	ループ交換、ループ融合、セクタキャッシュ、配列マージ、ループ分割
		L1キャッシュのレイテンシを隠蔽する。	L1プリフェッチ
メモリアクセス待ち	L2キャッシュミスが多発している。	L2キャッシュ利用の効率化を図る。	ループ交換、ループブロッキング、ストリップマイニング、セクタキャッシュ、ループ分割
		L2キャッシュのレイテンシを隠蔽する。	L2プリフェッチ
		データアクセス量を削減する。	高速ストア（ZFILL）

(*1) Fortranの場合、指示行!OCL NORECURRENCEや!OCL NOVREC

- 第一部（本講義）ではSIMD化、ソフトウェアパイプライニング、プリフェッチについて後の章で概念やオプションの使用方法等を説明します。
- 第三部（今後開催予定）では主な最適化手法について、コンパイルオプションや指示行の使用方法とその効果を説明する予定です。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法

  第一部（本講義）ではCPU単体性能向上のための重要な最適化手法として、SIMD化、ソフトウェアパイプライニング、プリフェッチを取り上げます。それ以外の各種最適化手法は第三部（今後開催予定）で取り上げる予定です。

  各手法ごとに、まず概念を説明します。その後、各手法ごとのコンパイルオプションやコンパイルリストの見方を説明します。その後、各手法が適用されるループ例を紹介します。

- 【付録】参考文献

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法

    本節では後の節の準備として、全ての最適化手法に共通な事項であるコンパイル時の最適化メッセージを出力するコンパイルオプションを説明します。本講義では自動スレッド並列化の最適化手法は取り上げませんが、この最適化情報も出力されるため、参考として出力される情報の見方についても説明します。
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - Fortranのコンパイルリスト出力に関連するオプション
  - C/C++ tradモードのコンパイルリスト出力に関連するオプション
  - C/C++ clangモードのコンパイルリスト出力に関連するオプション
  - 【参考情報】自動並列化のコンパイルリスト（FortranおよびC/C++ tradモードの場合）
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

■ 最適化メッセージを、コンパイルリストに表示する方法 [1/4]

■ 以下のプログラム例を、1 のように「-Nlst」を付けてコンパイルすると、a.lstに以下のコンパイルリストが作成される。

プログラム例：a.F90

```
SUBROUTINE SUB(A,N,X)
  IMPLICIT NONE
  INTEGER::N,I,J
  DOUBLE PRECISION::A(N,N),X
  DO J=1,N
    DO I=1,N
      A(I,J) = A(I,J)/X
    ENDDO
  ENDDO
  RETURN
END SUBROUTINE SUB
```

```
frtpx -Kfast -Nlst -c a.F90
```

1

コンパイルリスト：a.lst

```
External subroutine subprogram "SUB"
 (line-no.)(nest)
          1          SUBROUTINE SUB(A,N,X)
          2            IMPLICIT NONE
          3            INTEGER::N,I,J
          4            DOUBLE PRECISION::A(N,N),X
          5       1    DO J=1,N
          6       2      DO I=1,N
          7       2        A(I,J) = A(I,J)/X
          8       2      ENDDO
          9       1    ENDDO
         10            RETURN
         11          END SUBROUTINE SUB
```

行番号

DOループの
ネストのレベル。
- 一番外側：「1」
- ひとつ内側：「2」

```
Diagnostic messages: program name(SUB)
  jwd8220o-i  "a.F90", line 1: 副作用の可能性のある最適化を行いました。
  jwd8206o-i  "a.F90", line 7: 除算を逆数の乗算に変更しました。
```

■ 最適化メッセージを、コンパイルリストに表示する方法 [2/4]

■ 2 の「-Nlst=t」を指定すると、②～④が追加される。

```
frtpx -Kfast -Nlst=t  -c  a.F90
```

2

```
(line-no.)(nest)(optimize)
        1                          ②
        2
        3
        4
                                      SUBROUTINE SUB(A,N,X)
                                        IMPLICIT NONE
                                        INTEGER::N,I,J
                                        DOUBLE PRECISION::A(N,N),X
                                    <<< Loop-information Start >>>
                                    <<<  [OPTIMIZATION]                    ③
                                    <<<    COLLAPSED
                                    <<<    SIMD(VL: 8)
                                    <<<    SOFTWARE PIPELINING(（省略）)
                                    <<<    PREFETCH(HARD) Expected by compiler :
                                    <<<      A
                                    <<< Loop-information  End >>>
                                        DO J=1,N
                                    <<< Loop-information Start >>>
                                    <<<  [OPTIMIZATION]                    ④
                                    <<<    COLLAPSED
                                    <<< Loop-information  End >>>
        5       1          2v          DO I=1,N
        6       2          2v            A(I,J) = A(I,J)/X
        7       2          2v
（以下、省略）
```

SIMD化情報

多重ループの一重化

数字：アンローリングされた場合の展開数。
f：フルアンローリング。
i：インライン展開。

■ 最適化メッセージを、コンパイルリストに表示する方法 [3/4]

■ 3 の「-Koptmsg=2」を指定(*1)すると、⑤～⑧の最適化メッセージが追加される。

```
frtpx -Kfast -Koptmsg=2 -Nlst=t -c a.F90
```

3

(*1) -Kfast,optmsg=2 のように-Kオプションをまとめて指定することもできる。

```
 (line-no.)(nest)(optimize)
         1                      SUBROUTINE SUB(A,N,X)
(中略)
         5     1       2v         DO J=1,N
                                <<< Loop-information Start >>>
                                <<<  [OPTIMIZATION]
                                <<<    COLLAPSED
                                <<< Loop-information  End >>>
         6     2       2v           DO I=1,N
         7     2       2v             A(I,J) = A(I,J)/X
         8     2       2v           ENDDO
         9     1       2v         ENDDO
(以下、省略)
```

```
Diagnostic messages: program name(SUB)
   jwd8220o-i  "a.F90", line 1: 副作用の可能性のある最適化を行いました。
⑤  jwd6002s-i  "a.F90", line 5: このDOループをSIMD化しました。
⑥  jwd8204o-i  "a.F90", line 5: ループにソフトウェアパイプライニングを適用しました。
⑦  jwd8205o-i  "a.F90", line 5: ループの繰返し数が256回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
   jwd8206o-i  "a.F90", line 7: 除算を逆数の乗算に変更しました。
⑧  jwd8330o-i  "a.F90", line 7: 多重DOループをDO変数J,...,Iで1重化しました。
```

■ 最適化メッセージを、コンパイルリストに表示する方法 [4/4]

■ 以下のプログラム例について、 4 の「-Koptmsg=guide」を指定すると、-Koptmsg=2のメッセージに加えて、最適化(*2)が適用できなかった場合の阻害要因と対処方法を示すガイダンスメッセージ⑨が出力される。

プログラム例：b.F90

```
SUBROUTINE SUB(A,N)
  INTEGER::N
  DOUBLE PRECISION::A(N)
  DO I=2,N
    A(I) = A(I-1) + 1D0
  ENDDO
  RETURN
END SUBROUTINE SUB
```

```
frtpx -Kfast -Koptmsg=guide –Nlst=t -c b.F90
```

4

(*2) SIMD化、自動並列化、ソフトウェアパイプライニング、インライン展開

```
(line-no.)(nest)(optimize)
        1                       SUBROUTINE SUB(A,N)
（中略）
        4     1        2s         DO I=2,N
        5     1        2s           A(I) = A(I-1) + 1D0
        6     1        2s         ENDDO
```

```
 Diagnostic messages: program name(SUB)
   jwd8204o-i  "c.F90", line 4: ループにソフトウェアパイプライニングを適用しました。
   jwd8205o-i  "c.F90", line 4: ループの繰返し数が6回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
   jwd6202s-i  "c.F90", line 5: データの定義引用の順序が逐次実行と変わるため、このDOループはSIMD化できません。(名前:A)
⑨       [ガイダンス]
         ループをSIMD化した時のデータの定義引用の順序が逐次実行と変わらないアルゴリズムに変更する。
```

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
    - Fortranのコンパイルリスト出力に関連するオプション
    - C/C++ tradモードのコンパイルリスト出力に関連するオプション
    - C/C++ clangモードのコンパイルリスト出力に関連するオプション
    - 【参考情報】自動並列化のコンパイルリスト（FortranおよびC/C++ tradモードの場合）
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

-Nlst[={p|t}]

- コンパイルリストを「～.lst」というファイルに出力する。
- 「={p|t}」の省略時は-Nlst=pが適用される。

■ -Nlst=p

- ソースリストと統計情報を出力。
- 自動並列化、ループアンローリング、SIMD化、インライン展開等の最適化情報を出力。

■ 例

```
(line-no.)(optimize)
       23    p     2v    for (int i=0; i<N; i++){
       24    p     2v      A[i] = i;
       25    p     2v    }
       26     i            sub(A,N);
```

■ -Nlst=t

- -Nlst=pでの出力に加えて、より詳細な最適化情報を出力。
  - ■ 例：上の図のforループに対して、以下の情報が表示される。

```
<<< Loop-information Start >>>
<<<  [PARALLELIZATION]
<<<    Standard iteration count: 1000
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 160, MVE: 3, POL: S)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      A
<<< Loop-information  End >>>
```

-Koptmsg[={1|2}]

- 最適化メッセージ出力する。デフォルトは-Knooptmsgである。
- 「={1|2}」が省略された場合、-Koptmsg=1が指定されたものとみなす。

■ -Koptmsg=1

- 実行結果に副作用を生じる可能性がある最適化をしたことをメッセージ出力する。

■ 例

```
(line-no.)(optimize)
       55  pp     2v    for (int i=0; i<N; i++){
       56   p     2v      A[i] /=  X;
       57   p     2v      A[i] += A[i] + B[i] + C[i] + D[i];
       58   p     2v    }
```

```
jwd8206o-i  "b.cpp", line 56: 除算を逆数の乗算に変更しました。
jwd8209o-i  "b.cpp", line 57: 多項式の演算順序を変更しました。
```

■ -Koptmsg=2

- -Koptmsg=1オプションのメッセージに加えて、自動並列化、SIMD化、ソフトウェアパイプライニングなどの最適化メッセージを出力する。

■ 例

```
jwd5001p-i  "b.cpp", line 55: ループ制御変数'i'のループを並列化しました。
jwd6001s-i  "b.cpp", line 55: ループ制御変数'i'のループをSIMD化しました。
jwd8204o-i  "b.cpp", line 55: ループにソフトウェアパイプライニングを適用しました。
jwd8205o-i  "b.cpp", line 55: ループの繰返し数が144回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
```

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
    - Fortranのコンパイルリスト出力に関連するオプション
    - C/C++ tradモードのコンパイルリスト出力に関連するオプション
    - C/C++ clangモードのコンパイルリスト出力に関連するオプション
    - 【参考情報】自動並列化のコンパイルリスト（FortranおよびC/C++ tradモードの場合）
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

-ffj-lst[={p|t}]

- コンパイルリストを「～.lst」というファイルに出力する。
- 「={p|t}」の省略時は-ffj-lst=pが適用される。

■ -ffj-lst=p

- ソースリストを出力する。
- ループアンローリング、SIMD化、インライン展開などの最適化情報を出力する。
  - ■ 例

```
(line-no.)(optimize)
       35                #pragma loop unroll 8
       36          8v    for (int i=0; i<N; i++){
       37                  A[i] = i;
       38                }
       39     i          std::cout<<"A[0]="<<A[0]<<std::endl;
```

■ -ffj-lst=t

- -ffj-lst=pの出力に加えて、より詳細な最適化情報を出力する。
  - ■ 例：上の図のforループに対して、以下の情報が表示される。

```
<<< Loop-information Start >>>
<<<  [OPTIMIZATION]
<<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit Interleave: 1)
<<< Loop-information End >>>
```

※ コンパイル対象のファイルについて同じディレクトリ下に「～.opt.yaml」というファイルが既に存在する場合、標準エラー出力に以下のメッセージを出力し、コンパイルリストは出力されない。

```
FCCpx: 警告: ～.opt.yamlが既に存在するため、翻訳時情報は出力されません。
```

■ 【補足】C/C++ clangモードのコンパイルリスト関連のその他オプションとtradモードとの対応 [1/3]

-ffj-lst-out=*file*

- 指定されたファイル名***file***にコンパイルリストを出力する。
- -ffj-lst=pオプションも有効になる。

-ffj-src

- コンパイルリストを標準出力に出力する。
- -ffj-lstまたは-ffj-lst-out=fileオプションが同時に指定された場合、コンパイルリストはファイルに出力される。

■ 【補足】C/C++ clangモードのコンパイルリスト関連のその他オプションとtradモードとの対応 [2/3]

-Rpass=.*

- 最適化メッセージを標準エラー出力に出力する。

-Rpass-analysis=.*

- コンパイラが最適化を適用しなかった理由を標準エラー出力に出力する。

-Rpass-missed=.*

- 最適化が適用されなかったことを示すメッセージを標準エラー出力に出力する。
- ただし、出力されるメッセージには、コンパイラが最適化を適用しなかった理由は含まれない。

■ 注意事項

- tradモードでの「jwdxxxxz-y」の形式の最適化メッセージはコンパイルリストには出力されない。
- -Rpass=.*と指定すると、ループの数が多い場合、大量のメッセージが出力される。特定のメッセージだけを表示したい場合、-Rpass=\(sve-loop-vectorize\|inline\)などと指定する。

■ 【補足】C/C++ clangモードのコンパイルリスト関連のその他オプションとtradモードとの対応 [3/3]

■ tradモードのコンパイルリスト出力に関連した以下のオプションは、clangモードでは以下のオプションに置換される。

tradモードでのオプション名	clangモードでのオプション名	備考
-Koptmsg=2	-Rpass=.*	メッセージ関連
-Nlst	-ffj-lst	コンパイラ全般に関連
-Nlst=p	-ffj-lst=p	コンパイラ全般に関連
-Nlst=t	-ffj-lst=t	コンパイラ全般に関連
-Nlst_out=file	-ffj-lst-out=file	コンパイラ全般に関連
-Nsrc	-ffj-src	コンパイラ全般に関連

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
    - Fortranのコンパイルリスト出力に関連するオプション
    - C/C++ tradモードのコンパイルリスト出力に関連するオプション
    - C/C++ clangモードのコンパイルリスト出力に関連するオプション
    - 【参考情報】自動並列化のコンパイルリスト（FortranおよびC/C++ tradモードの場合）
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

■ 自動並列化されたループのコンパイルリスト ［1/5］

- ①の-Kparallelを指定すると、コンパイラは、並列性があると判断でき、かつ並列化した方が速くなると判断したループ（反復回数が多いループ）を自動並列化する。
- ②の-Koptmsg=2 -Nlst=tを指定すると、自動並列化に関する情報がコンパイルリストと最適化メッセージに表示される。

```
frtpx -Kfast,parallel,ocl,optmsg=2 -Nlst=t a.F90
```

①　②

コンパイルリスト

```
                    <<< Loop-information Start >>>
                    <<<  [PARALLELIZATION]
                    <<<    Standard iteration count: 1334
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 8)
                    <<<    SOFTWARE PIPELINING(（省略）)
                    <<<    PREFETCH(HARD) Expected by compiler :
                    <<<      A
                    <<< Loop-information  End >>>
13      1  pp   2v         DO I=1,N
14      1   p   2v           A(I) = A(I) + 1D0
15      1   p   2v         ENDDO
```

最適化メッセージ

```
jwd5001p-i  "a.F90", line 13: このDOループを並列化しました。(名前:I)
```

■ 自動並列化されたループのコンパイルリスト ［2/5］

```
                        <<< Loop-information Start >>>
                        <<<  [PARALLELIZATION]
                        <<<    Standard iteration count: 1334
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<<    PREFETCH(HARD) Expected by compiler :
              ③        <<<      A
                        <<< Loop-information  End >>>
13      1  pp   2v          DO I=1,N
14      1   p   2v            A(I) = A(I) + 1D0
15      1   p   2v          ENDDO
```

③の表示：DO文または配列演算（※）に対する自動並列化情報を示す。

※ 例：
```
DIMENSION A(100)
A = 0.0
```

- 記号の意味を以下に示す。

p	DOループまたは配列演算が並列化されたことを示します。
m	DOループまたは配列演算が並列化された部分とされなかった部分があることを示します。
s	DOループまたは配列演算が並列化されなかったことを示します。
空白	DOループまたは配列演算が並列化対象でないことを示します。

■ 自動並列化されたループのコンパイルリスト ［3/5］

```
                        <<< Loop-information Start >>>
                        <<<  [PARALLELIZATION]
                        <<<    Standard iteration count: 1334
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<<    PREFETCH(HARD) Expected by compiler :
          ④             <<<      A
                        <<< Loop-information  End >>>
13      1  pp   2v          DO I=1,N
14      1   p   2v            A(I) = A(I) + 1D0
15      1   p   2v          ENDDO
```

④の表示：並列化された範囲の先頭を示す。

■ 自動並列化されたループのコンパイルリスト ［4/5］

```
                        <<< Loop-information Start >>>
                        <<<  [PARALLELIZATION]
                        <<<    Standard iteration count: 1334
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<<    PREFETCH(HARD) Expected by compiler :
              ③         <<<      A
                        <<< Loop-information  End >>>
13      1   pp    2v          DO I=1,N
14      1    p    2v            A(I) = A(I) + 1D0
15      1    p    2v          ENDDO
             ⑤
```

⑤の表示：DO文に対して③で p、m、または s が表示された場合、DOループ中の実行文に対する自動並列化情報を示す。

- 記号の意味を以下に示す。

p	並列化可能な実行文であることを示します。
m	並列化可能な部分と不可能な部分が含まれる実行文であることを示します。
s	並列化不可能な実行文であることを示します。

■ 自動並列化されたループのコンパイルリスト ［5/5］

■ 本例のようにループに並列性がある場合、⑥、⑦が表示される。

- 計算量が少ないDOループをスレッド並列化すると、スレッド生成や同期のオーバーヘッドにより、逐次で実行するよりも遅くなる。

- ⑦は、ループ反復回数Nが
（本例では）1334回以上の場合は並列に実行し、
それより少ない場合は逐次に実行することを示す。

- 1334回はコンパイラが判断したしきい値を示す。

```
                    <<< Loop-information Start >>>
                    <<<  [PARALLELIZATION]                          ⑥
                    <<<    Standard iteration count: 1334           ⑦
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 8)
                    <<<    SOFTWARE PIPELINING(（省略）)
                    <<<    PREFETCH(HARD) Expected by compiler :
                    <<<      A
                    <<< Loop-information  End >>>
13      1  pp   2v         DO I=1,N
14      1   p   2v           A(I) = A(I) + 1D0
15      1   p   2v         ENDDO
```

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化

    本節では、SIMD化の考え方やSIMD化するためのコンパイルオプション及び指示行の使い方について、Fortranのループ例を用いて説明します。
    C/C++の場合の指示行については本節の最後の小節「【補足】FortranおよびC/C++での各種指示行について」を参照して下さい。
    - SIMD化の概念説明
    - SIMD化のコンパイルオプション
    - SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）
    - 【補足情報】C/C++ clangモードのコンパイルリストの注意事項
    - 【補足情報】関数がインライン展開された場合のコンパイルリストについて
    - SIMD化の例
    - 【補足】FortranおよびC/C++での各種指示行について
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

内容

■ SIMD化の概念説明

本小節では、SIMD化の概念について説明します。最初に、IF文を含まないDOループのSIMD化の動作イメージを説明し、次に、IF文を含むDOループのSIMD化の動作イメージを説明します。なお、説明の便宜上FortranのDOループを用いて説明しています。

本小節の内容

■ 動作説明のためのループ例と簡易モデル

■ ループ例

右の図に示すDOループを例に、SIMD化されたDOループの動作イメージを説明する。

ループ例

```
DO I=1,4
  B(I) = A(I) + 10.0
ENDDO
```

■ 簡易モデル

- SIMD命令は1命令で2要素を処理するとする(*1)。

  (*1) 実際の演算器では一つのSIMD命令で倍精度実数の場合8要素を処理できる。

- 以降のスライドでは、通常の命令に対応するSIMD命令の呼び名・表記を以下のようにする。

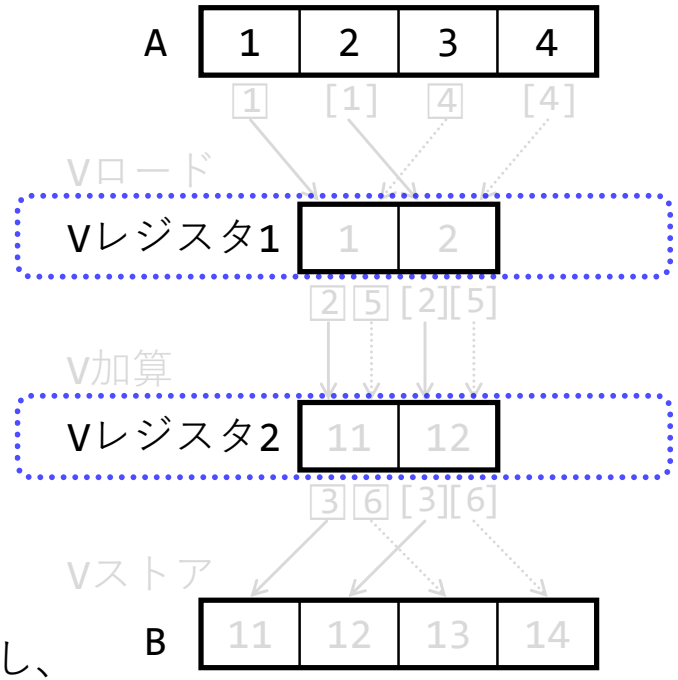
命令	対応するSIMD命令の呼び名	表記
ロード	ベクトルロード	「Vロード」
加算	ベクトル加算	「V加算」
ストア	ベクトルストア	「Vストア」

■ SIMD命令で使用するレジスタ

- SIMD命令では、一つのレジスタで2要素※を持つベクトルレジスタを使用する。

※ 簡易モデル

本例で使用する二つのベクトルレジスタを、
「Vレジスタ1」、「Vレジスタ2」と表す。



■ SIMD化されたDOループのイメージ [1/5]

- SIMD命令は2要素を一度に（同時に）処理するため、
  以下の図のループは 0 に示すように一つ飛びに反復し、
  1回目の反復でI=1とI=2の2要素を処理する。
- （次スライドへ続く）

SIMD化されたDOループのイメージ

```
DO I=1,4,2  0
```

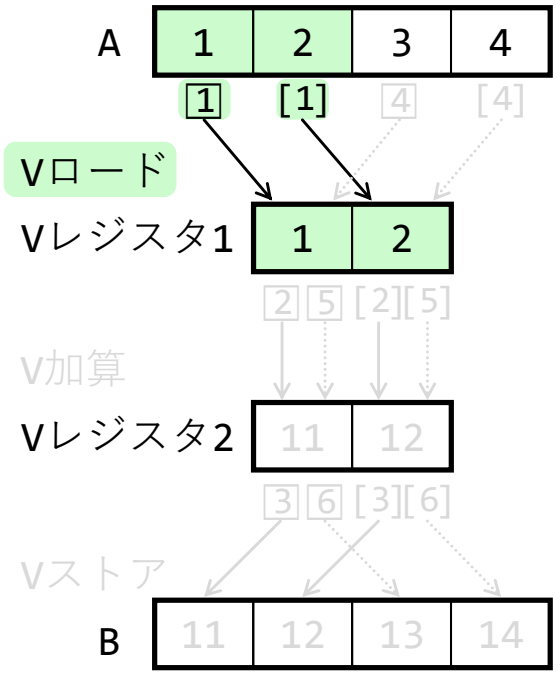
Vロード	Vレジスタ1(1)=A(I)	1	4	Vレジスタ1(2)=A(I+1) [1][4]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0	2	5	Vレジスタ2(2)=Vレジスタ1(2)+10.0 [2][5]
Vストア	B(I)=Vレジスタ2(1)	3	6	B(I+1)=Vレジスタ2(2) [3][6]

```
ENDDO
```

■ SIMD化されたDOループのイメージ [2/5]

■ 1 と [1] で、メモリ上のA(1)とA(2)の2要素に対し、一度に「Vレジスタ1」へ ベクトルロード「Vロード」を行う。

■ （次スライドへ続く）



SIMD化されたDOループのイメージ

```
DO I=1,4,2  0
```

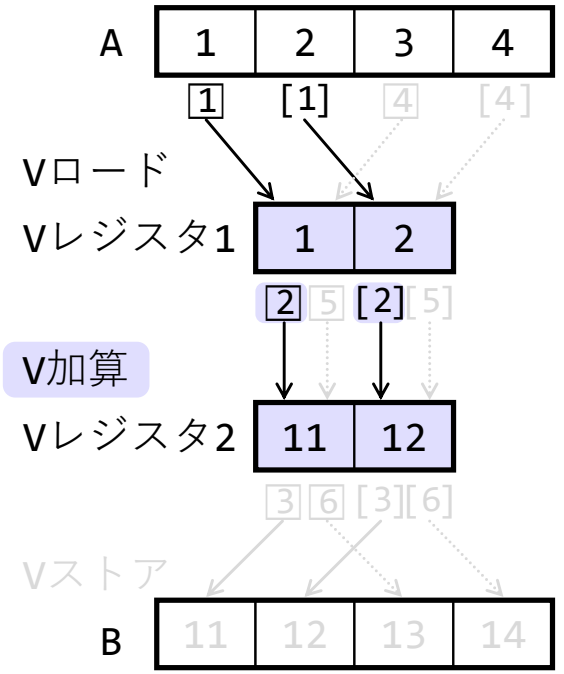
Vロード	Vレジスタ1(1)=A(I) 1 4	Vレジスタ1(2)=A(I+1) [1][4]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0 2 5	Vレジスタ2(2)=Vレジスタ1(2)+10.0 [2][5]
Vストア	B(I)=Vレジスタ2(1) 3 6	B(I+1)=Vレジスタ2(2) [3][6]

```
ENDDO
```

■ SIMD化されたDOループのイメージ [3/5]

■ 2 と [2] で、「Vレジスタ1」内の2要素に対し、一度に10.0をベクトル加算「V加算」し、結果を「Vレジスタ2」に代入する。

■ （次スライドへ続く）



SIMD化されたDOループのイメージ

DO I=1,4,2 0

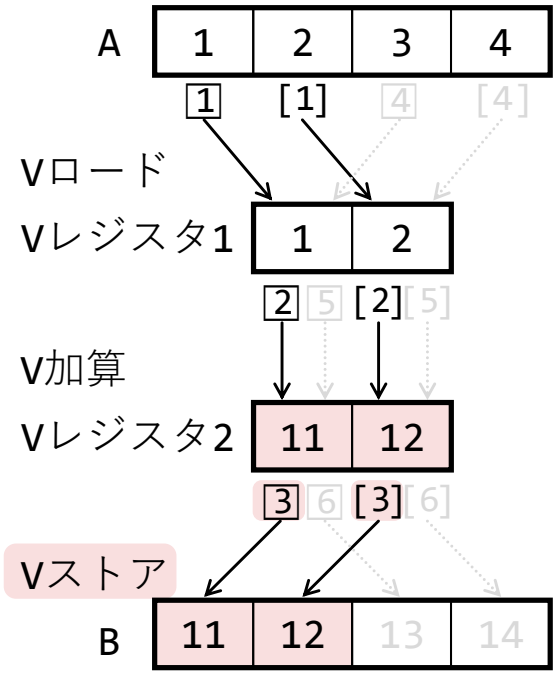
Vロード	Vレジスタ1(1)=A(I) 1 4	Vレジスタ1(2)=A(I+1) [1][4]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0 2 5	Vレジスタ2(2)=Vレジスタ1(2)+10.0 [2][5]
Vストア	B(I)=Vレジスタ2(1) 3 6	B(I+1)=Vレジスタ2(2) [3][6]

ENDDO

■ SIMD化されたDOループのイメージ [4/5]

■ 3 と [3] で、「Vレジスタ2」の2要素に対し、一度にメモリ上のB(1)、B(2)へ ベクトルストア「Vストア」を行う。

■ （次スライドへ続く）



SIMD化されたDOループのイメージ

DO I=1,4,2 0

Vロード	Vレジスタ1(1)=A(I) 1 4	Vレジスタ1(2)=A(I+1) [1][4]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0 2 5	Vレジスタ2(2)=Vレジスタ1(2)+10.0 [2][5]
Vストア	B(I)=Vレジスタ2(1) 3 6	B(I+1)=Vレジスタ2(2) [3][6]

ENDDO

■ SIMD化されたDOループのイメージ [5/5]

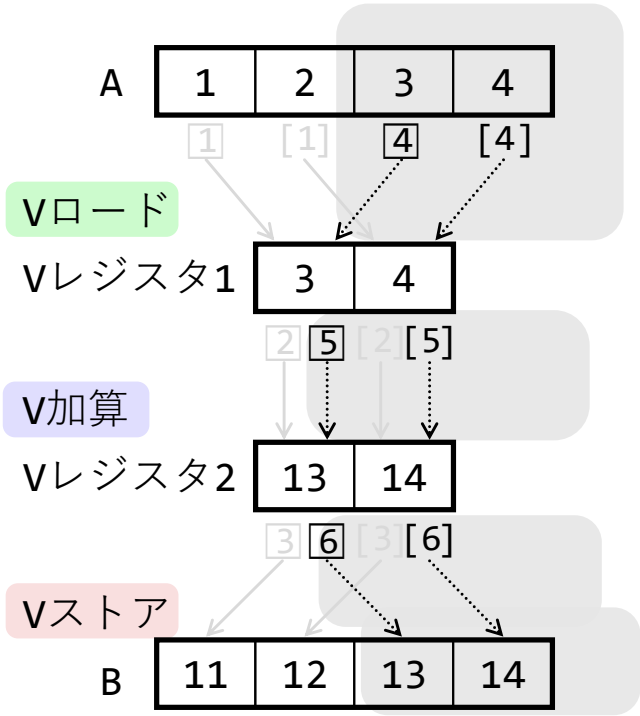
■ 同様にA(3)、A(4)に対し、

4 と [4] で「Vロード」、

5 と [5] で「V加算」、

6 と [6] で「Vストア」

を行う。



以上でIF文を含まない場合のSIMD化の動作イメージの説明を終わる。

SIMD化されたDOループのイメージ

```
DO I=1,4,2  0
```

Vロード	Vレジスタ1(1)=A(I)	1 4	Vレジスタ1(2)=A(I+1)	[1][4]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0	2 5	Vレジスタ2(2)=Vレジスタ1(2)+10.0	[2][5]
Vストア	B(I)=Vレジスタ2(1)	3 6	B(I+1)=Vレジスタ2(2)	[3][6]

```
ENDDO
```

■ SIMD化の概念説明

次に、IF文を含むDOループのSIMD化の動作イメージを説明します。
なお、説明の便宜上FortranのDOループを用いて説明しています。

本小節の内容

- IF文を含まないDOループのSIMD化の動作イメージ
- IF文を含むDOループのSIMD化の動作イメージ

■ 動作説明のためのループ例と簡易モデル

■ ループ例

右の図に示すIF文を含むDOループについて、SIMD化された場合の動作イメージを以降のスライドに示す。

ループ例

```
DO I=1,4
  IF (IFLAG(I)>0) THEN
    B(I) = A(I) + 10.0
  ENDIF
ENDDO
```

■ 簡易モデル（再掲）

- SIMD命令は1命令で2要素を処理するとする(*1)。

  (*1) 実際の演算器では一つのSIMD命令で倍精度実数の場合8要素を処理できる。

- 以降のスライドでは、通常の命令に対応するSIMD命令の呼び名・表記を以下のようにする（再掲）。

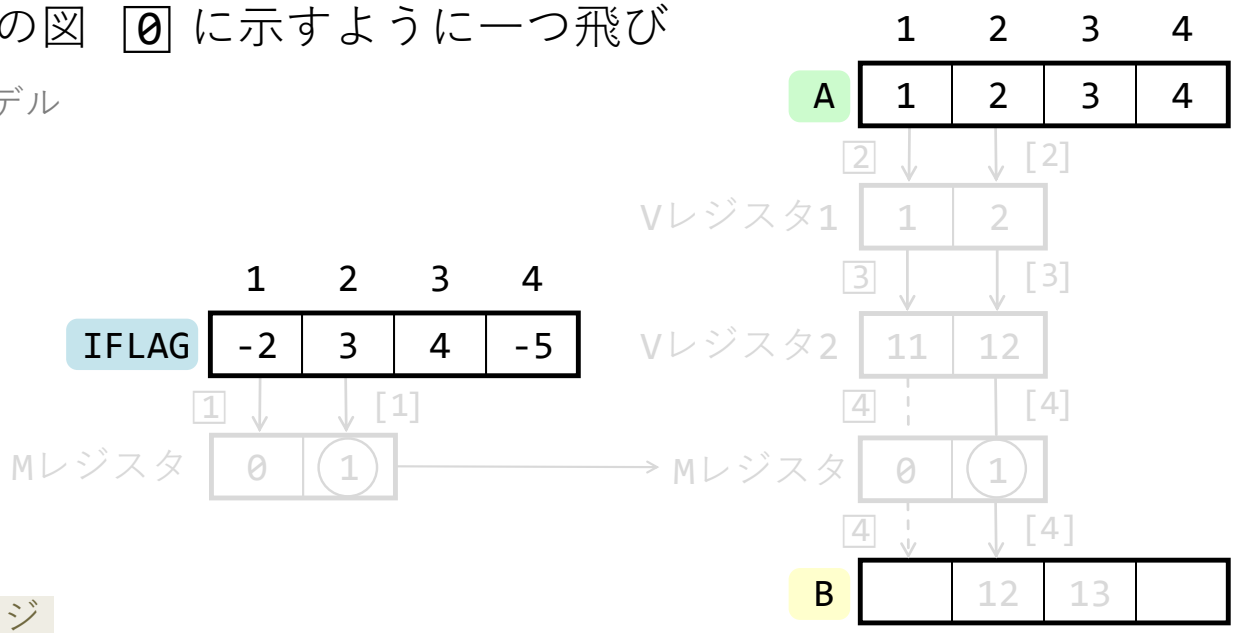
命令	対応するSIMD命令の呼び名	表記
ロード	ベクトルロード	「Vロード」
加算	ベクトル加算	「V加算」
ストア	ベクトルストア	「Vストア」

■ SIMD化されたDOループのイメージ [1/7]

■ SIMD命令は2要素※を一度に（同時に）処理するため、
図のループは、一番下の図 [0] に示すように一つ飛びに反復する。 ※ 簡易モデル

■ （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN    ①
    B(I) = A(I) + 10.0    ②
  ENDIF
ENDDO
```



SIMD化されたDOループのイメージ

DO I=1,4,2 [0]

【IFLAG(I)>0が真なら】Mレジスタ(1) = 1	1	【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1	[1]
【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0	1	【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0	[1]
Vレジスタ1(1) = A(I)	2	Vレジスタ1(2) = A(I+1)	[2]
Vレジスタ2(1) = Vレジスタ1(1) + 10.0	3	Vレジスタ2(2) = Vレジスタ1(2) + 10.0	[3]
【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)	4	【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)	[4]

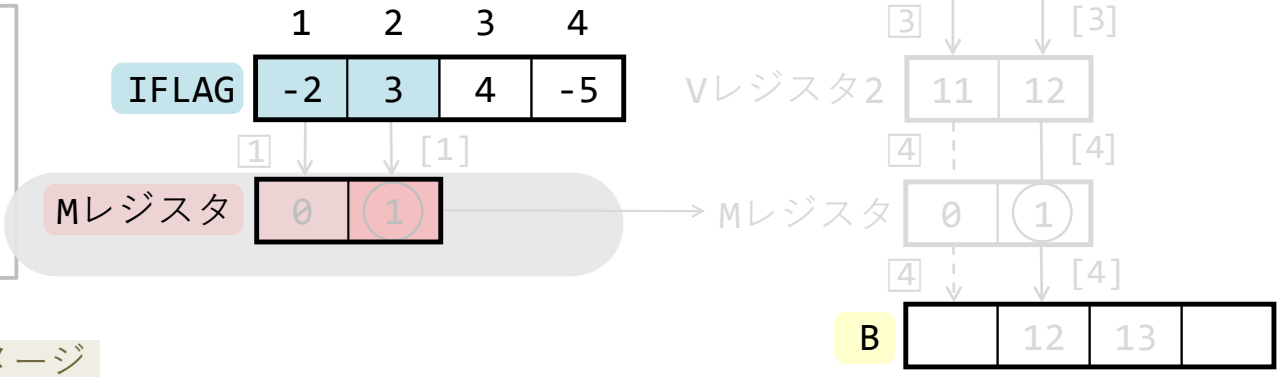
ENDDO

■ SIMD化されたDOループのイメージ [2/7]

■ IF文の比較結果を設定するベクトルレジスタを、以下ではベクトルマスクレジスタと呼び、「Mレジスタ」と表す。

■ （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN    ①
    B(I) = A(I) + 10.0    ②
  ENDIF
ENDDO
```



SIMD化されたDOループのイメージ

```
DO I=1,4,2  [0]
```

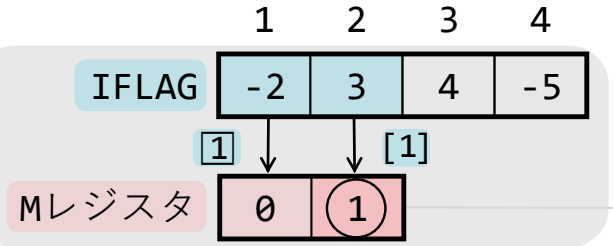
【IFLAG(I)>0が真なら】Mレジスタ(1) = 1	1	【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1	[1]
【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0	1	【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0	[1]
Vレジスタ1(1) = A(I)	2	Vレジスタ1(2) = A(I+1)	[2]
Vレジスタ2(1) = Vレジスタ1(1) + 10.0	3	Vレジスタ2(2) = Vレジスタ1(2) + 10.0	[3]
【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)	4	【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)	[4]

```
ENDDO
```

■ SIMD化されたDOループのイメージ [3/7]

- 1 と[1]で、I=1とI=2に対し、①のIF文の条件式を一度に比較し、真ならば1を、偽ならば0を、「Mレジスタ」に設定する。
- （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN    ①
    B(I) = A(I) + 10.0    ②
  ENDIF
ENDDO
```



SIMD化されたDOループのイメージ

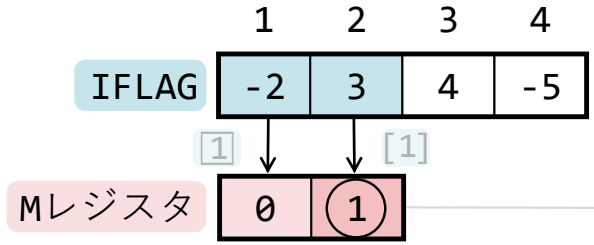
```
DO I=1,4,2  0
  【IFLAG(I)>0が真なら】Mレジスタ(1) = 1          1   【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1          [1]
  【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0          1   【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0          [1]
  Vレジスタ1(1) = A(I)                           2   Vレジスタ1(2) = A(I+1)                          [2]
  Vレジスタ2(1) = Vレジスタ1(1) + 10.0           3   Vレジスタ2(2) = Vレジスタ1(2) + 10.0           [3]
  【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)     4   【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)     [4]
ENDDO
```

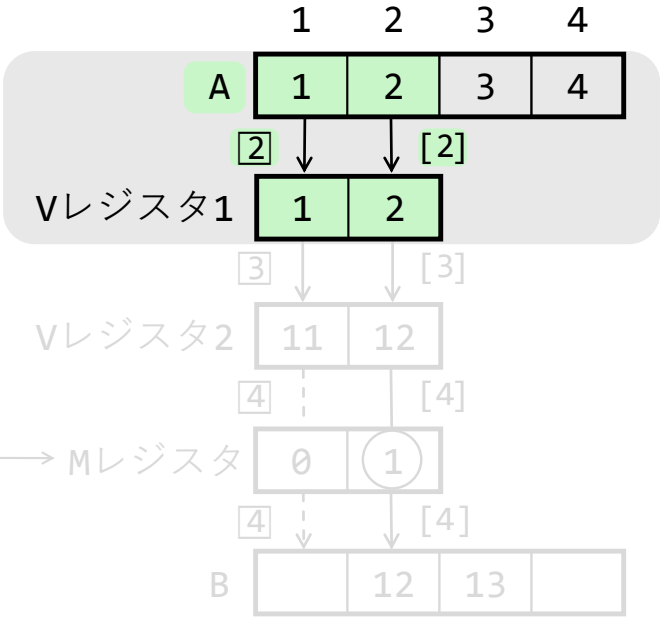
■ SIMD化されたDOループのイメージ [4/7]

■ 2 と[2]で、②の右辺の配列Aを「Vレジスタ1」にロードする。

■ （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN    ①
    B(I) = A(I) + 10.0    ②
  ENDIF
ENDDO
```





SIMD化されたDOループのイメージ

```
DO I=1,4,2  0
【IFLAG(I)>0が真なら】Mレジスタ(1) = 1          1   【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1          [1]
【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0          1   【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0          [1]
Vレジスタ1(1) = A(I)                           2   Vレジスタ1(2) = A(I+1)                          [2]
Vレジスタ2(1) = Vレジスタ1(1) + 10.0           3   Vレジスタ2(2) = Vレジスタ1(2) + 10.0           [3]
【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)     4   【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)     [4]
ENDDO
```

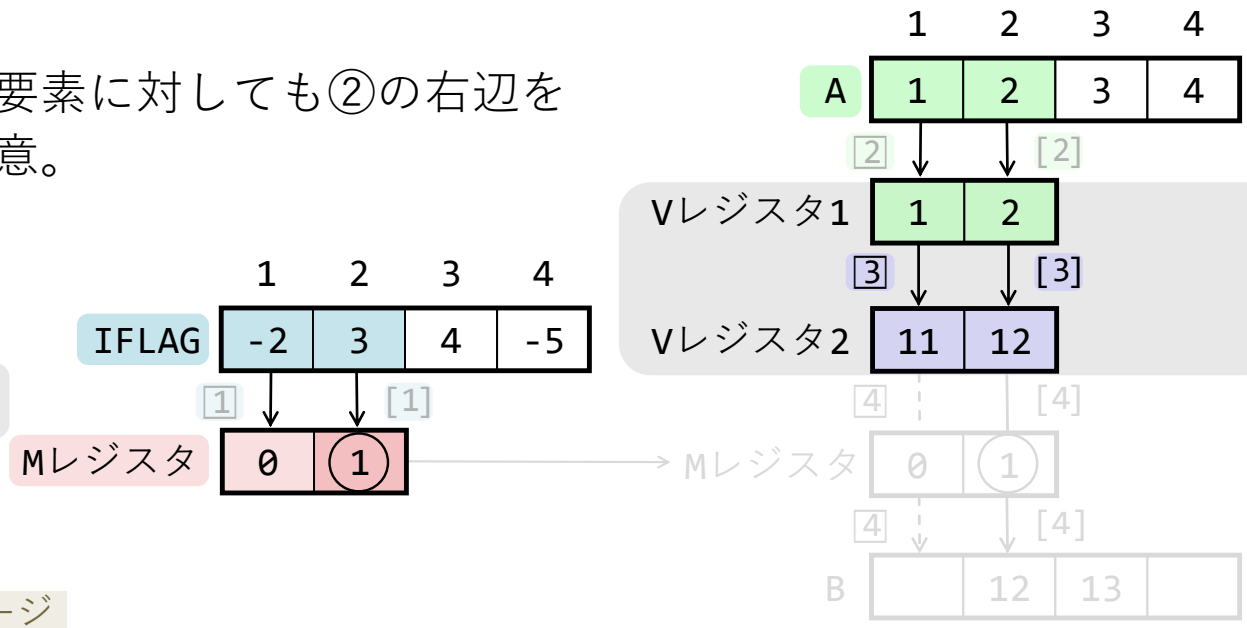
■ SIMD化されたDOループのイメージ [5/7]

■ [3] と [3] で足し算を行って結果を「Vレジスタ2」に代入する。

⚠ ①の条件式が偽の要素に対しても②の右辺を実行することに注意。

■ （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN  ①
    B(I) = A(I) + 10.0  ②
  ENDIF
ENDDO
```



SIMD化されたDOループのイメージ

```
DO I=1,4,2  [0]
```

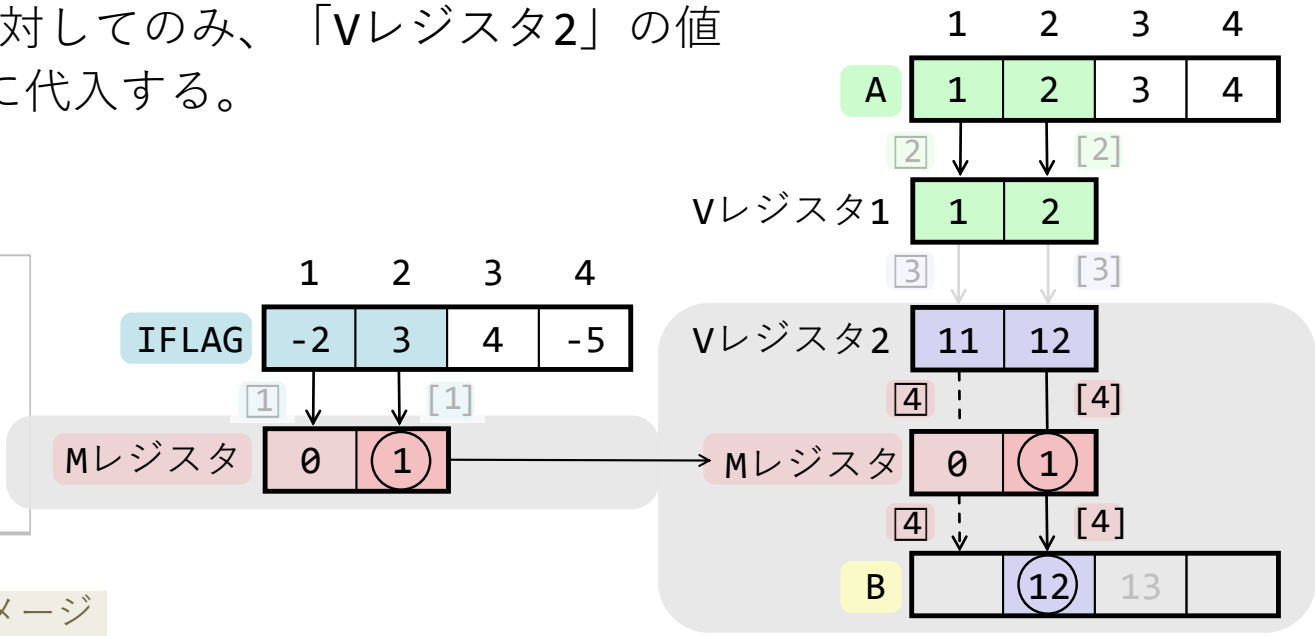
【IFLAG(I)>0が真なら】Mレジスタ(1) = 1	[1]	【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1	[1]
【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0	[1]	【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0	[1]
Vレジスタ1(1) = A(I)	[2]	Vレジスタ1(2) = A(I+1)	[2]
Vレジスタ2(1) = Vレジスタ1(1) + 10.0	[3]	Vレジスタ2(2) = Vレジスタ1(2) + 10.0	[3]
【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)	[4]	【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)	[4]

```
ENDDO
```

■ SIMD化されたDOループのイメージ [6/7]

- 4 と [4] で、「Mレジスタ」の値が1（つまり①の条件式が真）の要素に対してのみ、「Vレジスタ2」の値を②の左辺の配列Bに代入する。
- （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN  ①
    B(I) = A(I) + 10.0  ②
  ENDIF
ENDDO
```



SIMD化されたDOループのイメージ

```
DO I=1,4,2  0
```

【IFLAG(I)>0が真なら】Mレジスタ(1) = 1	1	【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1	[1]
【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0	1	【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0	[1]
Vレジスタ1(1) = A(I)	2	Vレジスタ1(2) = A(I+1)	[2]
Vレジスタ2(1) = Vレジスタ1(1) + 10.0	3	Vレジスタ2(2) = Vレジスタ1(2) + 10.0	[3]
【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)	4	【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)	[4]

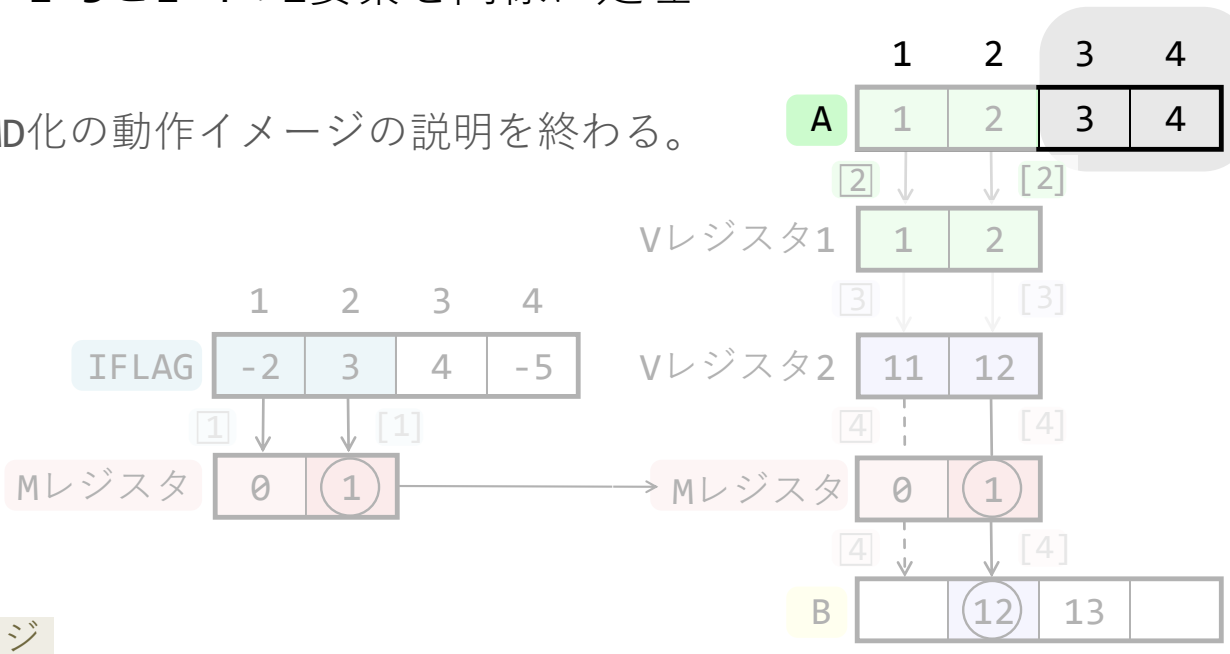
```
ENDDO
```

■ SIMD化されたDOループのイメージ [7/7]

■ 次に 0 でI=3となり、I=3とI=4の2要素を同様に処理する。

以上でIF文を含む場合のSIMD化の動作イメージの説明を終わる。

```
DO I=1,4
  IF (IFLAG(I)>0) THEN    ①
    B(I) = A(I) + 10.0    ②
  ENDIF
ENDDO
```



SIMD化されたDOループのイメージ

```
DO I=1,4,2  0
```

【IFLAG(I)>0が真なら】Mレジスタ(1) = 1	1	【IFLAG(I+1)>0が真なら】Mレジスタ(2) = 1	[1]
【IFLAG(I)>0が偽なら】Mレジスタ(1) = 0	1	【IFLAG(I+1)>0が偽なら】Mレジスタ(2) = 0	[1]
Vレジスタ1(1) = A(I)	2	Vレジスタ1(2) = A(I+1)	[2]
Vレジスタ2(1) = Vレジスタ1(1) + 10.0	3	Vレジスタ2(2) = Vレジスタ1(2) + 10.0	[3]
【Mレジスタ(1)=1なら】B(I) = Vレジスタ2(1)	4	【Mレジスタ(2)=1なら】B(I+1) = Vレジスタ2(2)	[4]

```
ENDDO
```

内容

■ SIMD化のコンパイルオプション

- -O2を指定すると、SIMD化に関して、FortranおよびC/C++ tradモードの場合-Ksimd=autoが、C/C++ clangモードの場合-fvectorizeオプションが誘導される。

下線付きがデフォルト

オプション（Fortranおよび C/C++ tradモードの場合）	オプション（C/C++ clang モードの場合）	意味
-Ksimd=1	-fvectorize	SIMD拡張命令を利用したオブジェクトを生成することを指示する。
-Ksimd=2		-Ksimd=1オプションの機能に加え、if文などを含むループに対して、SIMD拡張命令を利用したオブジェクトを生成することを指示する。 ⚠ if文内の命令を冗長実行するため、if文の真率によっては実行性能が低下する場合がある。 ⚠ if文内の式を投機実行するため、プログラムの論理上、実行されないはずの命令が実行され、実行結果に副作用（実行時の例外発生など）を生じることがある。
-Ksimd=auto (*1)		ループをSIMD化するか否かをコンパイラが自動的に判定することを指示する。if文を含むループのSIMD化が促進される。

(*1) C/C++ tradモードの-Ksimdオプションはclangモードで-fvectorizeオプションに置き換わる。

■ -Ksimdに対応する指示行

指示行（Fortranの場合）(*1)	指示行（C/C++ tradモードの場合）(*1)	指示行（C/C++ clangモードの場合）(*1)	意味
!OCL SIMD	#pragma loop simd (*2)	#pragma clang loop vectorize(enable)	SIMD化を有効にする。
!OCL NOSIMD	#pragma loop nosimd (*2)	#pragma clang loop vectorize(disable)	SIMD化を無効にする。
		#pragma clang loop vectorize_width(*n*, scalable)	SIMD化を有効にする。 SVEを利用(*3)したSIMD化においてSIMD長が*n*の定数倍になる(*4)。

(*1) 指示行を有効にするために、FortranおよびC/C++ tradモードの場合-Koclオプションを、C/C++ clangモードの場合-ffj-oclオプションを指定する必要がある。

(*2) tradモードのこれらの指示行はclangモードで対応する指示行に置き換わる。

(*3) C/C++ clangモードで-msve-vector-bits=scalableオプションが有効な場合、SIMD長が可変であると指定する。SIMD長が可変場合、SVEのベクトルレジスタを特定サイズとみなさずに翻訳して実行時にサイズを決定する。

(*4) SVEのベクトルレジスタサイズを最小の128ビットとみなした場合の、一つのSIMD命令で処理するデータの要素数を、*n*で指定する。例えばSVEのベクトルレジスタサイズが512ビットの場合、(512÷128)×*n*となり、4×*n*がSIMD長になる。*n*に指定できる値は、2または4である。*n*に4を指定した場合、SIMD長は、単精度で16要素、倍精度で8要素になる。

その他のSIMD化関連指示行は後の「SIMD化の例」の節の中で、および「【補足】FortranおよびC/C++での各種指示行について」の節で説明します。

■ IF文を含むDOループのSIMD化の注意事項

⚠ SIMD化の動作説明で指摘したように、IF文の条件式が偽の場合もIFブロック内の代入文の右辺の計算が実行される。

⚠ IF文を含むDOループのSIMD化は、以下の可能性がある。

(1)速度が遅くなる可能性がある。
(2)副作用を生じる可能性がある。

⚠ -Ksimd=2、または!OCL SIMDを指定すると、IF文を含むDOループもSIMD化の対象となる。ただし、上記(1)(2)の問題が発生する可能性があるため、-Ksimd=auto、または、(1)(2)の問題が発生しないDOループのみに対し、指示行で指定することを推奨する。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
    - SIMD化の概念説明
    - SIMD化のコンパイルオプション
    - SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）
    - 【補足情報】C/C++ clangモードのコンパイルリストの注意事項
    - 【補足情報】関数がインライン展開された場合のコンパイルリストについて
    - SIMD化の例
    - 【補足】FortranおよびC/C++での各種指示行について
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）

■ SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合※） [1/4]

※ C/C++ clangモードの場合については以降の「【補足情報】C/C++ clangモードのコンパイルリストの注意事項」の節を参照して下さい。

- DOループがSIMD化された場合、コンパイルリストには、①、②の最適化メッセージが表示される。
- 上記に加え、 1 、 2 にSIMD化に関する情報が表示される。

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 8)                                    ①
                    <<<    SOFTWARE PIPELINING(（省略）)
                    <<<    PREFETCH(HARD) Expected by compiler :
                    <<<      A
                    <<< Loop-information  End >>>
26      1       2v 1     DO I=1,N
27      1       2v 2       A(I) = A(I) + 1D0
28      1       2v       ENDDO
```

```
jwd6001s-i  "a.F90", line 26: このDOループをSIMD化しました。(名前:I)   ②
```

SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）

■ SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合） [2/4]

```
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<<    PREFETCH(HARD) Expected by compiler :
                        <<<      A
                        <<< Loop-information  End >>>
26      1       2v [1]    DO I=1,N
27      1       2v          A(I) = A(I) + 1D0
28      1       2v        ENDDO
```

①

■ [1] の表示

- DOループ全体（または配列演算）に対する、下記のSIMD化の状況が表示される。

v	DOループおよび配列演算がSIMD化されたことを示します。
m	DOループおよび配列演算がSIMD化された部分とされなかった部分があることを示します。
s	DOループおよび配列演算がSIMD化されなかったことを示します。
空白	DOループおよび配列演算がSIMD化対象でないことを示します。

SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）

■ SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合） [3/4]

```
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)                          ①
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<<    PREFETCH(HARD) Expected by compiler :
                        <<<      A
                        <<< Loop-information  End >>>
26      1       2v [1]      DO I=1,N
27      1       2v [2]        A(I) = A(I) + 1D0
28      1       2v          ENDDO
```

■ [2] の表示

- [1] で、DO文に「v、m、s」が表示された場合、
  DOループ内の各実行文に対する下記のSIMD化の状況が表示される。

v	SIMD化可能な実行文であることを示します。
m	SIMD化可能な部分と不可能な部分が含まれる実行文であることを示します。
s	SIMD化不可能な実行文であることを示します。

SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）

■ SIMD化に関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合） [4/4]

```
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<<    PREFETCH(HARD) Expected by compiler :
                        <<<      A
                        <<< Loop-information  End >>>
26      1       2v [1]     DO I=1,N
27      1       2v [2]       A(I) = A(I) + 1D0
28      1       2v         ENDDO
```

①

（注1）DOループまたは配列演算がSIMD化可能であっても、性能を考慮してSIMD化しない場合がある。その場合、[1] には「s」が表示され、[2] には解析結果がそのまま表示される。

（注2）[2] 内にある、非実行文および実行の制御を行う実行文（ELSE、ENDIF、CONTINUE、ENDDO、ELSEWHEREなど）には、前後の実行文に付加されたのと同じSIMD化情報の記号が表示されるか、空白になる。

内容

本小節の内容

ここでは、C/C++ clangモードのコンパイルリストの各種注意事項を説明します。最初に、tradモードのコンパイルリストとの主な相違点を説明します。

- tradモードのコンパイルリストとの主な相違点
- clangモードのコンパイルリストについての注意点

■ tradモードのコンパイルリストとの主な相違点 [1/3]

- 以下のループについて、tradモードとclangモードのコンパイルリストの相違点を記す。

```
#pragma omp parallel for
  for (int i=0; i<N; i++){
    for (int j=0; j<N; j++){
      B[i][j] = i*10.0 + j;
    }
  }
```

■ tradモードのコンパイルリストとの主な相違点 [2/3]

- tradモード：-Kfast,openmp,optmsg=2 -Nlst=t を付けてコンパイル

```
(line-no.)(optimize)
       63              #pragma omp parallel for
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    PREFETCH(HARD) Expected by compiler :
                        <<<      B
                        <<< Loop-information  End >>>
       64   p           for (int i=0; i<N; i++){
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING( （省略） )
                        <<<    PREFETCH(HARD) Expected by compiler : （省略）
                        <<< Loop-information  End >>>
       65   p     2v       for (int j=0; j<N; j++){
       66   p     2v         B[i][j] = i*10.0 + j;
       67   p     2v       }
       68   p           }
 （中略）
Optimization messages
  jwdxxxxz-y  "a.cpp", line n:  （省略）
 （中略）
Option information
  Command line options : ～
  Effective options    : ～
```

■ tradモードのコンパイルリストとの主な相違点 [3/3]

■ clangモード：-Nclang -Ofast -fopenmp -ffj-lst=t -Rpass=.* を付けてコンパイル

```
(line-no.)(optimize)
       63              #pragma omp parallel for
       64                for (int i=0; i<N; i++){
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit Interleave: 1)
                        <<< Loop-information End >>>
       65          v       for (int j=0; j<N; j++){
       66                    B[i][j] = i*10.0 + j;
       67                  }
       68                }
```

	tradモード	clang モード
OpenMPスレッド並列化されたことを示す記号「p」	各行に表示される	表示されない
SIMD化可能／されたことを示す記号「v」	各行に表示される	ループ制御文のみに表示される
「jwdxxxxz-y」の形式の診断メッセージ	表示される	表示されない

Option information
  Command line options : ～
  Effective options : ～
の行
	表示される	表示されない

本小節の内容

次に、clangモードのコンパイルリストについての注意点を説明します。

- tradモードのコンパイルリストとの主な相違点
- clangモードのコンパイルリストについての注意点

■ clangモードのコンパイルリストについての注意点

- 以下のループ例のように、変数Mの値がコンパイル時に不明なため後方依存性（※）の可能性があり、SIMD化できないにもかかわらず、SIMD化された記号やメッセージが表示されることがある。 ※ 後の「SIMD化の例」の節で説明します。

コンパイルリスト

```
(line-no.)(optimize)
       29                int M;
       31                std::ifstream input_file("c1_input.txt");
       32                input_file >> M;
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit Interleave: 1)
                        <<< Loop-information End >>>
       47          v     for (int i=M; i<N; i++){
       48                  A[i] = b*A[i-M] + c;
       49                }
```

標準エラー出力

```
c1.cpp:47:3: remark: vectorized loop (vectorization width: 2, bit width:
128, interleaved count: 1) [-Rpass=sve-loop-vectorize]
  for (int i=M; i<N; i++){
  ^
```

- 最終的にSIMD化が行われたかどうかは、詳細プロファイラなどで調べるか経過時間を測定するようにする。
- 本例でM=1を入力とした場合、詳細プロファイラfappで計測したところ実際SIMD化されていない。

内容

■ 【補足情報】関数がインライン展開された場合のコンパイルリストについて [1/4]

- C/C++ tradモードの場合、-Kfast（または-O3）を指定すると-x-が誘導され、以下の①と②に示すように条件に合致したユーザー定義関数はインライン展開される。

```
Command line options : -Kfast,parallel -Nnoclang -std=c++11 -Kocl,optmsg=2 -Nlst=t
                       -Nfjomplib
Effective options    : （中略）-x- （中略） -Kparallel（中略）-Ksimd=auto（以下、省略）
```

```
28              void sub3(double *s, double *a, int n)
29              {
                  （次スライドで説明するためここでは省略）
33              }
```

```
81              int main(int argc, char *argv[])
82              {
84                constexpr int N = 10000;
85                double a[N] = {};
89                double s = 0.0;
90                sub3(&s,a,N);
                  （以下、省略）
106              }
```

① i

② `jwd8101o-i  "a.cpp", line 90: 利用者定義の関数'_Z4sub3PdS_i'をインライン展開しました。`

■ 【補足情報】関数がインライン展開された場合のコンパイルリストについて [2/4]

- 以降のスライドで例示するように、C/C++（やFortran）で、サブルーチンや関数がインライン展開された場合、SIMD化や自動並列化に関して相反するメッセージが表示されたり、何も表示されないことがある。

- C/C++では、-O3を指定すると、条件に合致した関数は全てインライン展開されるオプション（-x-）がオンになるので、この現象が発生する頻度が高くなる。

- 最終的にSIMD化や自動並列化が行われたかどうかは、詳細プロファイラなどで調べるか、経過時間を測定するようにする。

■ 【補足情報】関数がインライン展開された場合のコンパイルリストについて [3/4]

- ③、⑤：ループが「SIMD化された」ことを示す。
- ④、⑥：ループが「SIMD化されなかった」ことを示す。

```
Command line options : -Kfast,parallel -Nnoclang -std=c++11 -Kocl,optmsg=2 -Nlst=t
-Nfjomplib
Effective options    : （中略）-x-（中略） -Kparallel（中略）-Ksimd=auto（以下、省略）
```

```
28              void sub3(double *s, double *a, int n)
29              {
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]
                 <<<    Standard iteration count: 1778
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)                          ③
                 （以下、省略）
          ④      <<< Loop-information  End >>>
30          2s    for (int i=0; i<n; i++){
31          2s      *s += a[i];
32          2s    }
33              }
```

⑤ jwd6004s-i "a.cpp", line 30: リダクション演算を含むループ制御変数'i'のループをSIMD化しました。

⑥ jwd6208s-i "a.cpp", line 31: 変数's'を定義・参照する順序がわからないため、定義・参照する順序が逐次実行と変わる可能性があり、このループはSIMD化できません。

■ 【補足情報】関数がインライン展開された場合のコンパイルリストについて [4/4]

- ⑦、⑨：ループが「自動並列化された」ことを示す。
- ⑧：「自動並列化されなかった」ことを示す記号（空白）が表示されている。

```
Command line options : -Kfast,parallel -Nnoclang -std=c++11 -Kocl,optmsg=2 -Nlst=t -
Nfjomplib
Effective options    : （中略）-x-（中略） -Kparallel（中略）-Ksimd=auto（以下、省略）
```

```
28              void sub3(double *s, double *a, int n)
29              {
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]                    ⑦
                 <<<    Standard iteration count: 1778
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 （以下、省略）
                 <<< Loop-information  End >>>
30          2s    for (int i=0; i<n; i++){
31          2s      *s += a[i];
32          2s    }
33              }
```

「p」が表示されない ⑧

⑨ jwd5004p-i "a.cpp", line 30: リダクション演算を含むループ制御変数'i'のループを並列化しました。

内容

■ SIMD化の例

本小節では、SIMD化するためのコンパイルオプション及び指示行の使い方について、Fortranのループ例を用いて説明します。
C/C++の場合の指示行については本節の最後の小節「【補足】Fortranおよび C/C++での各種指示行について」を参照して下さい。

以降のスライドではSIMD化される／されない以下の各種例を挙げます。最初にIF文を含まないDOループの場合を、次にIF文を含むDOループの場合を説明します。

- ■ IF文を含まないDOループ
  - ■ SIMD化可能な部分と不可能な部分が存在するループ
  - ■ 後方依存性のあるループ
  - ■ 前方依存性のあるループ
  - ■ 間接アドレス指定のDOループ
  - ■ ポインタ変数が含まれるループ
- ■ IF文を含むDOループ
  - ■ SIMD化する例
  - ■ リストベクトル変換

■ SIMD化可能な部分と不可能な部分が存在するループ [1/2]

- ①が示すように、-O2オプションは-Ksimd=autoを誘導し、-Ksimd=autoは-Kloop_part_simdを誘導する。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t -Knoswp
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Kloop_part_simd（中略）-Ksimd=auto （以下、省略）
```
①

- 以下のループ例では②で演算を行い、③でサブルーチンをコールしている。
- 最適化メーッセージは③のサブルーチンコールがSIMD化不可能であることを示している。
- さらに④の指示行!OCL LOOP_NOPART_SIMDを指定すると-Kloop_part_simdの指定を打ち消し、以下のループは部分的にSIMD化されない。

  ※ 指示行を有効にするために-Koclオプションを指定する必要がある。

```
11                          !OCL LOOP_NOPART_SIMD          ④
12      1         s           DO I=1,N
13      1         v             A(I) = A(I) + 1D0            ②
14      1         s             CALL SUB2(B(I))              ③
15      1         v           ENDDO
```

```
jwd6122s-i  "d.F90", line 14: DOループ内に、SIMD化の制約となる手続引用が存在します。
```

■ SIMD化可能な部分と不可能な部分が存在するループ [2/2]

- ⑤の指示行!OCL LOOP_PART_SIMDを指定した場合（あるいは指示行!OCL LOOP_NOPART_SIMDを削除し-Kloop_part_simdを有効にした場合）、ループが内部的に右のように分割され、②の演算を実行するループがSIMD化される。

※ 指示行を有効にするために-Koclオプションを指定する必要がある。

［ソースイメージ］

```
DO I=1,N
  A(I) = A(I) + 1D0
ENDDO
DO I=1,N
  CALL SUB2(B(I))
ENDDO
```

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t -Knoswp
                       -Kprefetch_nosequential -Knounroll
Effective options    :  （中略）-Kloop_part_simd（中略）-Ksimd=auto（以下、省略）   ①
```

```
35                      !OCL LOOP_PART_SIMD                          ⑤
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SIMD(VL: 8)
                      <<< Loop-information  End >>>
36      1         m       DO I=1,N
37      1         v         A(I) = A(I) + 1D0                        ②
38      1         s         CALL SUB2(B(I))                          ③
39      1         v       ENDDO
```

```
jwd6005s-i  "d.F90", line 36: このDOループを部分的にSIMD化しました。(名前:I)
```

■ 後方依存性のあるループ [1/3]

ループ例

- A(I)の計算ではA(I-1)の要素を参照している。

```
DO I=2,N
  A(I) = A(I-1) + B(I)
ENDDO
```

最初の2反復を展開

- A(2)は①で定義された後、②で参照される。
- DOループを展開したときに、↘の関係が現れる場合、本スライドでは後方依存性があると呼ぶ。

```
A(2) = A(1) + B(2)      ①
A(3) = A(2) + B(3)      ②
```

■ 後方依存性のあるループ [2/3]

最初の2反復を展開した図（再掲）

- ②で、更新後のA(2)を参照。

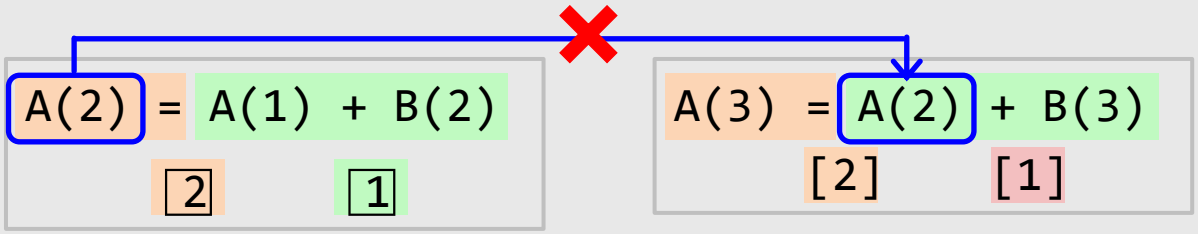
```
A(2) = A(1) + B(2)    ①
A(3) = A(2) + B(3)    ②
```

左の図がSIMD化された場合

■ 最初の2反復の①と②が一度に処理される。

- まず右辺の計算 [1] と［1］が一度に行われる。
  ⟹ A(2)+B(3)の計算［1］において、更新前のA(2)を参照してしまうことになる③。
- 次に左辺への代入 [2] と［2］が一度に行われる。

⚠ ③の「更新前のA(2)を参照」となってしまうため、本来の「更新後のA(2)を参照すべき」という関係でなくなり（つまり左の図の↘に相当するの関係がなくなり）、A(3)の値が左の②の場合と異なってしまう。



⚠ DOループを展開し、↘の依存関係（後方依存性）がある場合、コンパイラはこのDOループをSIMD化しない。

■ 後方依存性のあるループ [3/3]

- 本ループ例を-Ksimd=autoを有効にしてコンパイルした場合、コンパイルリストでは、SIMD化されなかったことを示す記号「s」と、「SIMD化できません」というメッセージが表示される。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Ksimd=auto（中略）-Kswp（以下、省略）
```

```
                                <<< Loop-information Start >>>
                                <<<  [OPTIMIZATION]
                                <<<    SOFTWARE PIPELINING(（省略）)
                                <<< Loop-information  End >>>
11      1         s         DO I=2,N
12      1         s           A(I) = A(I-1) + B(I)
13      1         s         ENDDO
```

```
jwd6202s-i  "f.F90", line 12: データの定義引用の順序が逐次実行と変わるため、
このDOループはSIMD化できません。(名前:A)
```

■ 前方依存性のあるループ [1/5]

ループ例

- A(I)の計算ではA(I+1)の要素を参照している。

```
DO I=1,N
  A(I) = A(I+1) + B(I)
ENDDO
```

最初の2反復を展開

- A(2)は①で参照された後、②で定義される。
- DOループを展開したときに、↗の関係が現れる場合、本スライドでは前方依存性があると呼ぶ。

```
A(1) = A(2) + B(1)    ①

A(2) = A(3) + B(2)    ②
```

■ 前方依存性のあるループ [2/5]

最初の2反復を展開した図（再掲）

- ①で、更新前のA(2)を参照。

```
A(1) = A(2) + B(1)   ①
A(2) = A(3) + B(2)   ②
```

左の図がSIMD化された場合

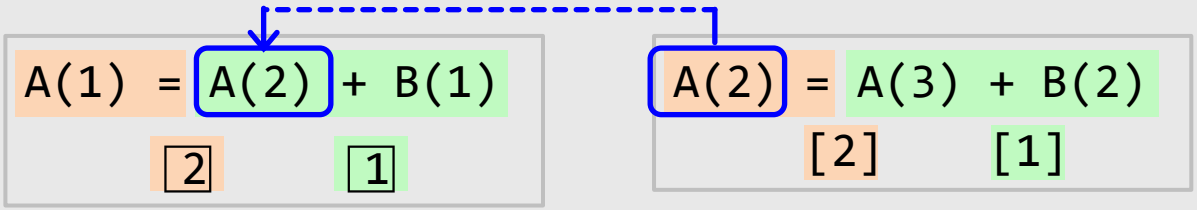
■ 最初の2反復の①と②が一度に処理される。

- まず右辺の計算 [1] と ［1］ が一度に行われる。

  ⟹ A(2)+B(1)の計算 [1] で、更新前のA(2)を参照。従って、左の展開図と同じ順序である。

- 次に左辺への代入 [2] と ［2］ が一度に行われる。

■ 従って、A(1)の値は左の①の場合と同じになる。



◆ DOループを展開し、↗の関係（前方依存性）がある場合、SIMD化される。

■ 前方依存性のあるループ [3/5]

- 本ループ例は前方依存性がコンパイラに分かるためSIMD化される。
- 本ループ例をSIMD化を有効にしてコンパイルした場合、コンパイルリストでは「SIMD（～）」というメッセージ、SIMD化されたことを示す記号「v」、および「SIMD化しました」というメッセージが表示される。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Ksimd=auto（中略）-Kswp（以下、省略）
```

```
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<< Loop-information  End >>>
20     1         v       DO I=1,N-1
21     1         v         A(I) = A(I+1) + B(I)
22     1         v       ENDDO
```

```
jwd6001s-i  "f.F90", line 20: このDOループをSIMD化しました。(名前:I)
```

■ 前方依存性のあるループ [4/5]

- 以下のループ例ではMが変数になっている。
- Mの値が必ず0以上だとすると、後方依存性がないのでSIMD化が可能である。
- しかし、コンパイラはMの値が不明なのでSIMD化しない。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Ksimd=auto（中略）-Kswp（以下、省略）
```

```
36      1          s          DO I=1,N-M
37      1          s            A(I) = A(I+M) + B(I)
38      1          s          ENDDO
```

```
jwd6203s-i  "f.F90", line 37: 添字式中の変数'M'は、配列'A'の定義引用順序を逐次実行と変える可能性があるため、このDOループはSIMD化できません。
```

■ 前方依存性のあるループ [5/5]

- Mの値が必ず0以上で配列Aに後方依存関係がないことをコンパイラに知らせるために、指示行「!OCL NOVREC」を指定（※）すると、このDOループはSIMD化される。

※ 括弧内に配列Aを指定する。指示行を有効にするために-Koclオプションを指定する必要がある。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Ksimd=auto（中略）-Kswp（以下、省略）
```

```
52                      !OCL NOVREC(A)
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SIMD(VL: 8)
                      <<< Loop-information  End >>>
53     1        v         DO I=1,N-M
54     1        v           A(I) = A(I+M) + B(I)
55     1        v         ENDDO
```

```
jwd6001s-i  "f.F90", line 53: このDOループをSIMD化しました。(名前:I)
```

⚠ Mの値が必ずしも0以上とは限らず、従って配列Aに後方依存関係がある場合に誤って本指示行を指定すると、コンパイル時に警告は出ず、誤ってSIMD化され、計算結果は保証されないので、注意して下さい。

■ 間接アドレス指定のDOループ [1/4]

■ 右のループ例では配列Aの添字が配列IFLAGにより間接アドレス指定となっている。

```
DO I=1,N
  A(IFLAG(I)) = A(IFLAG(I)) + B(I)
ENDDO
```

❖ 配列IFLAG内に同じ値（右の例では3）が存在する場合

- ループを展開すると、後方依存性↘が存在するためSIMD化できない。

N=4の場合の例

	1	2	3	4
IFLAG	4	2	3	3

```
A(4) = A(4) + B(1)
A(2) = A(2) + B(2)
A(3) = A(3) + B(3)
A(3) = A(3) + B(4)
```

■ 間接アドレス指定のDOループ [2/4]

```
DO I=1,N
  A(IFLAG(I)) = A(IFLAG(I)) + B(I)
ENDDO
```

❖ IFLAG内の値が全て異なる場合 [1/3]

- 後方依存性が存在しないのでSIMD化は可能。
- しかし、コンパイラには配列IFLAGの中身が分からないのでSIMD化されない。

N=4の場合の例

	1	2	3	4
IFLAG	4	2	3	1

```
A(4) = A(4) + B(1)
A(2) = A(2) + B(2)
A(3) = A(3) + B(3)
A(1) = A(1) + B(4)
```

- SIMD化を有効にしてコンパイルした場合、コンパイルリストでは記号「s」と「SIMD化できません」というメッセージが表示される。

```
78      1        s        DO I=1,N
79      1        m          A(IFLAG(I)) = A(IFLAG(I)) + B(I)
80      1        v        ENDDO
```

```
jwd6228s-i  "f.F90", line 79: データの定義引用の順序が逐次実行と変わる可能性があるため、このDOループはSIMD化できません。
```

■ 間接アドレス指定のDOループ [3/4]

❖ IFLAG内の値が全て異なる場合 [2/3]

- 配列Aに依存関係がないことをコンパイラに知らせるために、指示行「!OCL NOVREC」を指定（※）すると、以下のDOループはSIMD化される。

  ※ 括弧内に配列Aを指定する。指示行を有効にするために-Koclオプションを指定する必要がある。

- なお、本例ではソフトウェアパイプライニング化も行われている。

  コンパイルリスト中のソフトウェアパイプライニングに関する部分については「ソフトウェアパイプライニング」の節で説明します。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Ksimd=auto（中略）-Kswp（以下、省略）
```

```
85                     !OCL NOVREC(A)
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<<    SOFTWARE PIPELINING(（省略）)
                     <<< Loop-information  End >>>
86     1         v       DO I=1,N
87     1         v         A(IFLAG(I)) = A(IFLAG(I)) + B(I)
88     1         v       ENDDO
```

```
jwd6001s-i  "f.F90", line 86: このDOループをSIMD化しました。(名前:I)
```

■ 間接アドレス指定のDOループ [4/4]

❖ IFLAG内の値が全て異なる場合 [3/3]

- 本例では、配列Aの要素が回転を跨いで定義引用されないことを指示する指示行「!OCL NORECURRENCE」を指定（※）しても、SIMD化、ソフトウェアパイプライニング化される。

  ※ 「!OCL NOVREC」同様に括弧内に配列名を指定することもできる。指示行を有効にするために-Koclオプションを指定する必要がある。

```
93                        !OCL NORECURRENCE
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(（省略）)
                        <<< Loop-information  End >>>
94      1          v        DO I=1,N
95      1          v          A(IFLAG(I)) = A(IFLAG(I)) + B(I)
96      1          v        ENDDO
```

```
jwd6001s-i  "f.F90", line 94: このDOループをSIMD化しました。(名前:I)
```

- 本指示行を適用した実際の実行結果例は「ソフトウェアパイプライニング」の節のループ例「指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果」を参照して下さい。

⚠ 配列Aの要素が回転を跨いで定義引用されているにも関わらず誤って本指示行を指定した場合、計算結果は保証されないので、注意して下さい。

■ ポインタ変数が含まれるループ [1/3]

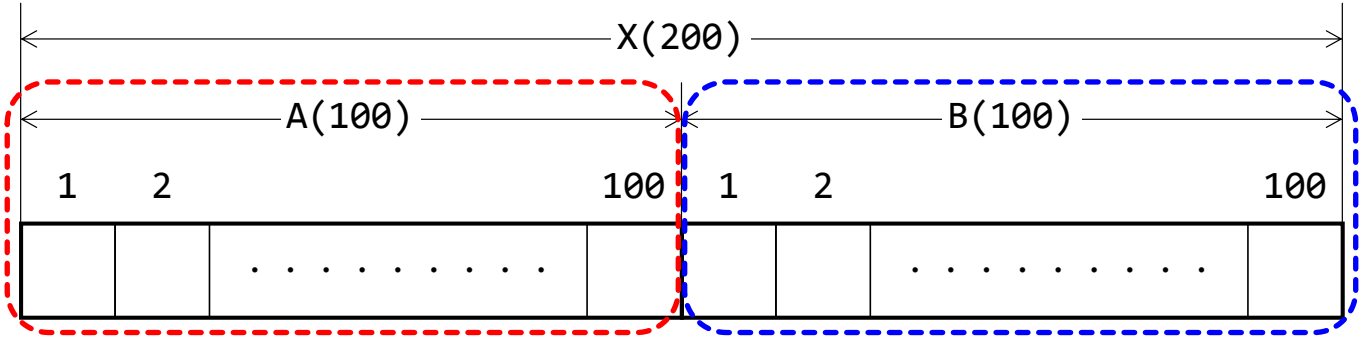
■ ①でターゲットの配列Xに対して、配列A、Bをポインターとして定義。

```
DOUBLE PRECISION,TARGET::X(200)
DOUBLE PRECISION,POINTER::A(:),B(:)
A => X(  1:100)                          ①
B => X(101:200)

DO I=1,100                               ②
  A(I) = B(I) + 1D0
ENDDO
```

■ 各配列はメモリ上に下図のように配置される。

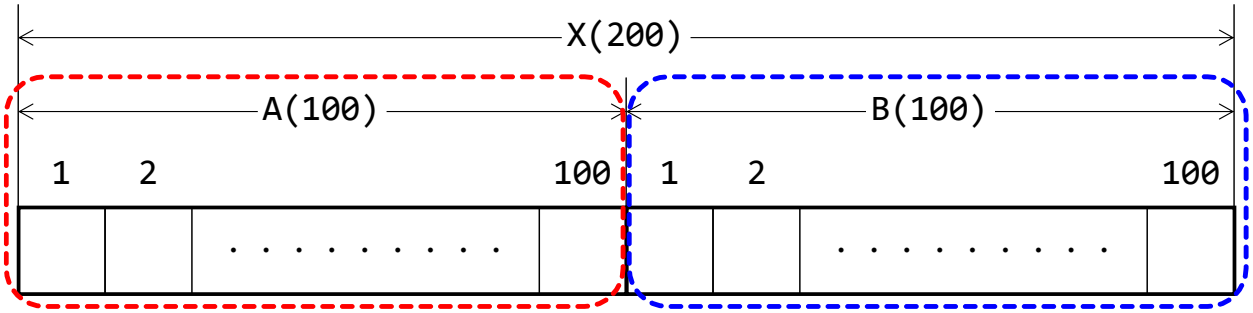
- 二つのポインターが示す記憶域がメモリ上で重なっていないので、②のDOループに後方依存性はなく、SIMD化は可能。



■ ポインタ変数が含まれるループ [2/3]

```
DOUBLE PRECISION,TARGET::X(200)
DOUBLE PRECISION,POINTER::A(:),B(:)
A => X( 1:100)                          ①
B => X(101:200)

DO I=1,100                              ②
  A(I) = B(I) + 1D0
ENDDO
```



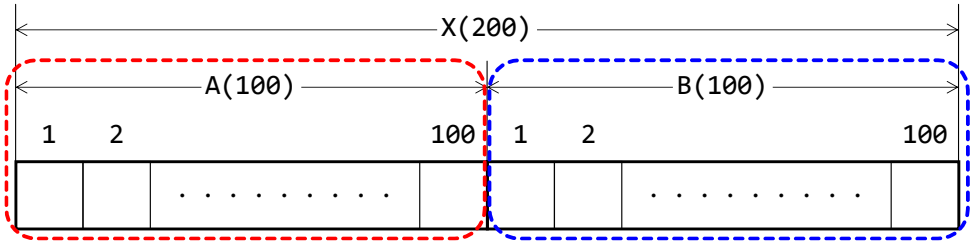
⚠ コンパイルした場合、コンパイラは①の指定からでは配列A、Bのメモリ上での配置を判断できないためSIMD化せず、以下の最適化メッセージが出力される。

```
jwd6228s-i  "h.F90", line 12: データの定義引用の順序が逐次実行と変わる可能性があるため、このDOループはSIMD化できません。
```

■ ポインタ変数が含まれるループ [3/3]

■ 「異なるポインター変数が同一の記憶域を指すことはない」ことをコンパイラに知らせる指示行「!OCL NOALIAS」を指定（※）すると、SIMD化できる。

※ 指示行を有効にするために-Koclオプションを指定する必要がある。




```
 5                          DOUBLE PRECISION,TARGET::X(200)
 6                          DOUBLE PRECISION,POINTER::A(:),B(:)
18                          A => X(  1:100)
19                          B => X(101:200)
21                        !OCL NOALIAS
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<< Loop-information  End >>>
22      1         v         DO I=1,100
23      1         v           A(I) = B(I) + 1D0
24      1         v         ENDDO
```

```
jwd6001s-i  "h.F90", line 22: このDOループをSIMD化しました。(名前:I)
```

⚠ 異なるポインタ変数が同一の記憶域を指しているにも関わらず誤って本指示行を指定した場合、計算結果は保証されないので、注意して下さい。

■ SIMD化の例

以降のスライドではIF文を含むDOループの場合についてSIMD化の各種例を挙げます。

- ■ IF文を含まないDOループ
  - ■ SIMD化可能な部分と不可能な部分が存在するループ
  - ■ 後方依存性のあるループ
  - ■ 前方依存性のあるループ
  - ■ 間接アドレス指定のDOループ
  - ■ ポインタ変数が含まれるループ
- ■ IF文を含むDOループ
  - ■ SIMD化する例
  - ■ リストベクトル変換

■ SIMD化する例

■ -Ksimd=auto

- 以下のループ例ではIF文により配列LFLAGの要素の値の真偽を判定している。
- 本ループ例では-Ksimd=autoにより、ループをSIMD化するかどうかをコンパイラが自動的に判定することを指示している。
- -Ksimd=autoにより、本例ではIF文を含んでいてもSIMD化可能と判断され、SIMD化されている。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll
Effective options    : （中略）-Ksimd=auto（以下、省略）
```

```
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<<    SOFTWARE PIPELINING(（省略）)
                     <<< Loop-information  End >>>
16     1         v       DO I=1,N
17     2         v         IF (LFLAG(I)) THEN
18     2         v           A(I) = A(I) + 1D0
19     2         v         ENDIF
20     1         v       ENDDO
```

```
jwd6001s-i  "k.F90", line 16: このDOループをSIMD化しました。(名前:I)
```

■ リストベクトル変換

- IF文の真率が低くかつIF文内の命令数が多いループに対しては、リストベクトル変換を適用することで性能向上できる可能性がある。
- リストベクトル変換は指示行「!OCL SIMD_LISTV」で動作する※。

※ 指示行を有効にするために-Koclオプションを指定する必要がある。

［ソースコード例］

```
DO I=1, N
!OCL SIMD_LISTV
  IF (M(I)) THEN
    A(I)=B(I)+C(I)
  END IF
END DO
```

■ 「!OCL SIMD_LISTV」使用の動作イメージ

- 右のソースコード例ではIF文で配列Mの要素の値の真偽を判定している。
- IF文に対して指示行「!OCL SIMD_LISTV」を指定している。

[最適化適用後のソースイメージ]

```
J=1
DO I=1,N
  IF (M(I)) THEN
    IDX(J)=I
    J=J+1
  END IF
END DO
!OCL NORECURRENCE
DO K=1,J-1
  I=IDX(K)
  A(I)=B(I)+C(I)
END DO
```

- 最適化適用後のソースイメージにおける、
  - DO Iのループは、IF文の条件を満たすループ変数の値を配列IDXに登録するループ、
  - DO Kのループは、登録した配列のサイズ分だけ回転して元のIF文内の命令を実行するループ

  となる。
- 変換した後半のDO Kのループはリストアクセスとなるが、SIMD化・ソフトウェアパイプライニングの対象となる。

■ リストベクトル変換の効果をみるループ例 [1/4]

■ 以下のIF文を含むループ②は、①の配列Mの設定によりIF文の真率が5%と低く、また多くの命令を含んでいる。

ループ②について、以下の三つのケース

(1) SIMD化しない場合、

(2) SIMD化した場合、

(3) SIMD化およびリストベクトル変換した場合

の計算時間を詳細プロファイラfappで計測して比較した。

```
INTEGER,PARAMETER::N=100000
DOUBLE PRECISION::A(N),B(N),C(N),D(N)
DOUBLE PRECISION::E(N),F(N),G(N),H(N)
DOUBLE PRECISION::P(N)
LOGICAL::M(N)
M=.FALSE.
DO I=1,N
  M(I)=(MOD(I,20).EQ.0)                                    ①
ENDDO
DO I=1,N                                                   ②
  IF (M(I)) THEN
    P(I) = A(I) + B(I)/2D0 + C(I)/3D0 + D(I)/4D0 &
       & + E(I)/5D0 + F(I)/6D0 + G(I)/7D0 + H(I)/8D0
  ENDIF
ENDDO
```

■ リストベクトル変換の効果をみるループ例 [2/4]

■ コンパイルリスト [1/2]

- コンパイルオプションで-Ksimd=autoを有効にしている。

```
Command line options : -O2 -Kprefetch_nosequential -Knounroll -Knoswp
                       -Kocl,optmsg=guide -Nlst=t
Effective options    : （中略）-Ksimd=auto（以下、省略）
```

(1) SIMD化しない場合 (※)

- -Ksimd=autoの指定を、指示行「!OCL NOSIMD」で打ち消してSIMD化を抑止している。

```
!OCL NOSIMD
  DO I=1,N
    IF (M(I)) THEN
      P(I)=...
    ENDIF
  ENDDO
```

(2) SIMD化した場合 (※)

- -Ksimd=autoにより、IF文を含んでいてもSIMD化されている。

```
      <<< Loop-information Start >>>
      <<<  [OPTIMIZATION]
      <<<    SIMD(VL: 8)
      <<< Loop-information  End >>>
v         DO I=1,N
v           IF (M(I)) THEN
v             P(I)=...
v           ENDIF
v         ENDDO
```

※ fappの計測ルーチン部分は次スライドで説明するので省略する。

■ リストベクトル変換の効果をみるループ例 [3/4]

■ コンパイルリスト [2/2]

(3) SIMD化およびリストベクトル変換した場合

- IF文の直前に指示行「!OCL SIMD_LISTV」指定により、リストベクトル変換を指示している。「リストベクトル変換を適用しました」というメッセージが表示される。
- DO Iのループの回転数だけでは計算時間が短かいため、DO Iのループの繰り返し実行の目的で外側にDO LLのループを導入し、DO LLのループを詳細プロファイラfappで計測する。

```
271                       CALL FAPP_START("SIMD_LISTV",1001,0)      ← fappの計測ルーチン
272     1                 DO LL=1,NLOOP
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<< Loop-information  End >>>
273     2         v       DO I=1,N
274     2                 !OCL SIMD_LISTV
275     3         v         IF (M(I)) THEN
276     3         v           P(I) = A(I) + B(I)/2D0 + C(I)/3D0 + D(I)/4D0 &
277     3                       & + E(I)/5D0 + F(I)/6D0 + G(I)/7D0 + H(I)/8D0
278     3         v         ENDIF
279     2         v       ENDDO
280     1                 ENDDO ! LL
281                       CALL FAPP_STOP("SIMD_LISTV",1001,0)       ← fappの計測ルーチン
```

```
INTEGER,PARAMETER::NLOOP=1000
```

```
jwd6001s-i  "l.F90", line 273: このDOループをSIMD化しました。(名前:I)
jwd6025s-i  "l.F90", line 276: IF構文内の実行文にリストベクトル変換を適用しました。
```

■ リストベクトル変換の効果をみるループ例 [4/4]

■ fapp計測結果（詳細プロファイラより）

- 単にSIMD化のみしたtune1ではIF文内の計算はIF文の条件の真偽によらず演算数にカウントされる。本ループ例ではIF文の真率が5%と低いため、演算数がasisの場合の演算数（条件が真のみの場合の本来あるべき演算数）に比べて増大している。その結果tune1はasisに比べて経過時間が増大している。
- 対してtune2ではIF文の条件が真の要素のみをリストにして必要な計算を実行しているため、演算数はasisの場合の演算数と一致している。なおかつSIMD命令率がasisから向上している。結果、経過時間はasisより短縮している。

バージョン	最適化の種類	経過時間（単位：秒）(*1)	asisからの高速化率	演算数の比（tune/asis）	命令数の比（tune/asis）	SIMD命令率
asis	(1) SIMD化しない場合	0.66	-	1.00	1.00	0.0%
tune1	(2) SIMD化した場合	4.38	0.15	20.00	0.73	72.7%
tune2	(3) SIMD化およびリストベクトル変換した場合	0.34	1.91	1.00	0.34	60.5%

(*1) 配列要素数N=100000、反復回数NLOOP=1000を指定した場合の累積時間。

内容

■ 【補足】FortranおよびC/C++での各種指示行について [1/3]

■ SIMD化の促進に関連する指示行

指示行（Fortran）(*1)	指示行（C/C++ tradモード）(*1)	指示行（C/C++ clangモード）(*1)	意味
!OCL NOVREC	#pragma loop novrec		ループ中の配列(*2)が後方依存関係でないこと（SIMD化が可能なこと）を指示する。
!OCL NORECURRENCE	#pragma loop norecurrence		ループ内の演算対象となる配列(*2)の要素が、回転を跨いで定義引用されないことを指示する。
		#pragma clang loop vectorize (assume_safety)	ループ中で配列の要素またはポインタ変数に対して依存がないことを指示する(*3)。

(*1) 指示行を有効にするために、FortranおよびC/C++ tradモードの場合-Koclオプションを、C/C++ clangモードの場合-ffj-oclオプションを指定する必要がある。

(*2) (配列名1,配列名2,...)の形式で対象となる配列名を指定する。括弧および配列名を省略すると、対象範囲内のすべての配列に有効となる。

(*3) 機能的にはtradモードの#pragma loop norecurrenceに相当する。

次スライドへ続く

■ 【補足】FortranおよびC/C++での各種指示行について [2/3]

■ 部分的SIMD化の指示行

指示行（Fortran）(*1)	指示行（C/C++ tradモード）(*1)	指示行（C/C++ clangモード）	意味
!OCL LOOP_PART_SIMD	#pragma loop loop_part_simd		ループを分割して部分的にSIMD化する機能を有効にする。

■ ポインタ変数に関連する指示行

指示行（Fortran）(*1)	指示行（C/C++ tradモード）(*1)	指示行（C/C++ clangモード）	意味
!OCL NOALIAS	#pragma loop noalias		異なるポインタ変数が同一の記憶領域を指さないことを指示する。

(*1) 指示行を有効にするために、FortranおよびC/C++ tradモードの場合-Koclオプションを、C/C++ clangモードの場合-ffj-oclオプションを指定する必要がある。

次スライドへ続く

■ 【補足】FortranおよびC/C++での各種指示行について [3/3]

■ リストベクトル変換の指示行

下線付きがデフォルト

指示行（Fortran）(*1)	指示行（C/C++ tradモード）(*1)	指示行（C/C++ clangモード）	意味
!OCL SIMD_LISTV [({ALL|THEN|ELSE})]	#pragma statement simd_listv [all|then|else]		IF構文内の実行文をリストベクトル変換することを指示する。本指示行は文単位で指定する。直後のIF文に対して適用される。対象とするブロックを括弧内に指定する。

(*1) 指示行を有効にするために、FortranおよびC/C++ tradモードの場合-Koclオプションを、C/C++ clangモードの場合-ffj-oclオプションを指定する必要がある。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

内容

内容

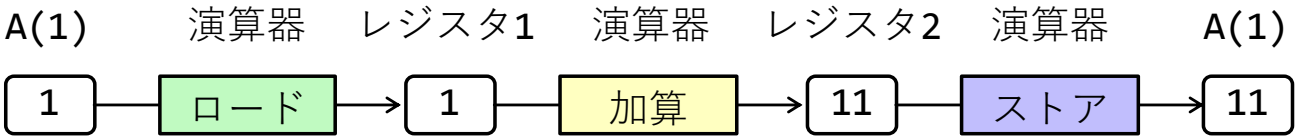
- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
    - ソフトウェアパイプライニングの概念説明
    - ソフトウェアパイプライニングのコンパイルオプション
    - ソフトウェアパイプライニングに関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）
    - ソフトウェアパイプライニングの例
    - 【補足】FortranおよびC/C++での各種指示行について
  - プリフェッチ
- 【付録】参考文献

■ 動作説明のためのループ例、および簡単化したモデルの説明

ループ例

```
DO I=1,6
  A(I) = A(I) + 10.0
ENDDO
```

- 上の図のループを実行し、1反復目の動作は以下のようになるとする。



簡単化したモデルの説明（実際のモデルを簡単化して説明するので、実際の動作と同じではない。）

- コア内にはロード、加算、ストアを行う三つの演算器があり(*1)、
  それぞれ処理を1単位時間で行なうとする。

  (*1) 実際のモデルと個数は異なる。実際のコアでは、ロードとストアの演算器は兼用で2個あり、ロードまたはストアの「アドレス計算」を行なう。

- 各演算器は同時に実行できるとする。

コア内の演算器（簡単化モデル）	ロード	加算	ストア
処理時間	1単位時間	1単位時間	1単位時間
	同時に実行できるとする		

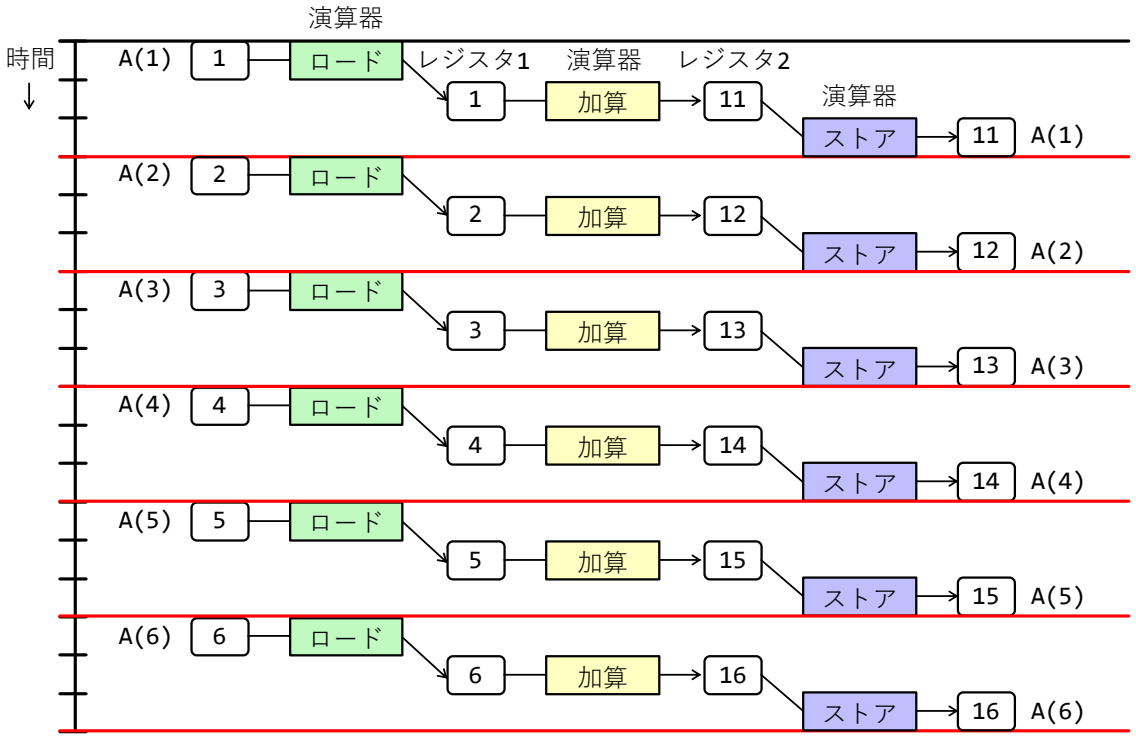
■ ソフトウェアパイプライニングが行われない場合

- 1反復目が完了してから2反復目を実行。タイムチャートは以下の図のようになる。
- 演算器は1時点で一つのみ稼働。
- 6反復の処理に6×3＝18単位時間かかる。

ループ例

```
DO I=1,6
  A(I) = A(I) + 10.0
ENDDO
```

タイムチャート



- 各演算器は3単位時間ごとに一度しか稼働しておらず、各演算器は2単位時間演算待ちの状態である。
- このような演算待ちのある状態を命令スケジューリングが悪い状態という。

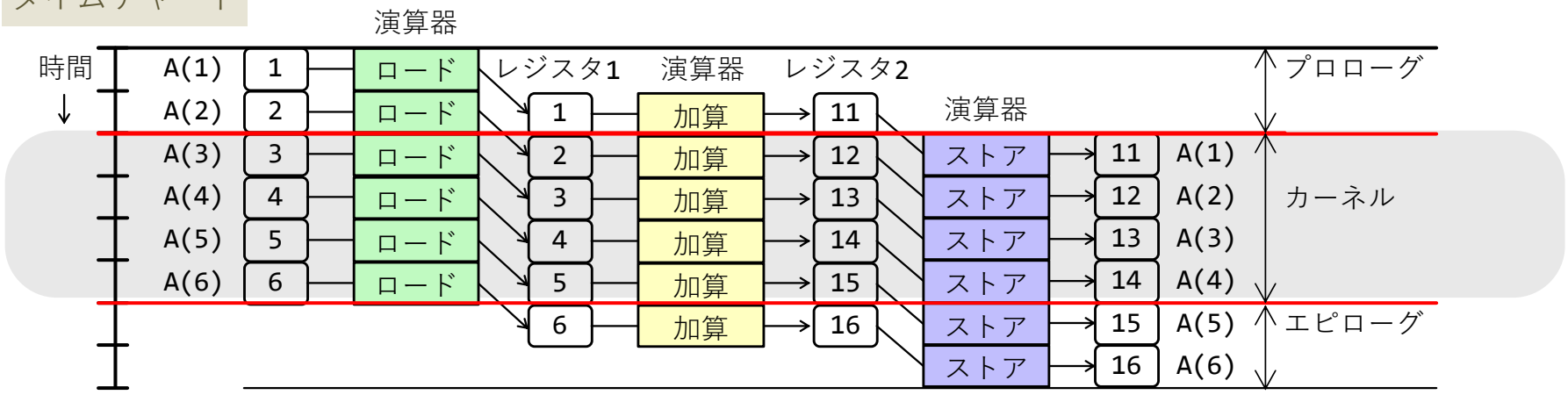
■ ソフトウェアパイプライニングが行われた場合

ループ例

```
DO I=1,6
  A(I) = A(I) + 10.0
ENDDO
```

- 1反復目の実行が終了する前に2反復目を開始。
  タイムチャートは以下の図のようになる。
- 演算器は1時点で三つ同時に稼働できる。
  - 例えば、1反復目がストアする時に、2反復目が加算することができ、
    さらに3反復目がロードすることが、同時にできる。
- 6反復の処理は8単位時間に短縮。
  - 三つの演算器がすべて稼働している定常状態の部分をカーネル、
    カーネルの前の部分をプロローグ、後の部分をエピローグと呼ぶ。

タイムチャート



- 各演算器は演算待ち無く各単位時間で稼働している。
- このような状態を命令スケジューリングが良い状態という。

内容

■ ソフトウェアパイプライニングのコンパイルオプション

- Fortranおよび C/C++ tradモードの場合、-O2を指定すると、ソフトウェアパイプライニングに関して、-Kswpオプションが誘導される。

オプション（Fortranおよび C/C++ tradモードの場合）	オプション（C/C++ clangモードの場合）	意味
-Kswp (*1)	-ffj-swp	ソフトウェアパイプライニングの最適化を行うことを指示する。ただし、最適化の効果が期待できないものに対しては最適化を行わない。

(*1) C/C++ tradモードの-Kswpオプションはclangモードで-ffj-swpオプションに置き換わる。

⚠ C/C++ clangモードの場合の注意事項

- C/C++ clangモードの場合、ソフトウェアパイプライニングのオプション-ffj-swpは-O2からは（-Ofastからも）誘導されない。
- SIMD長が可変である(*2)と指定(*3)してSIMD化されたループではソフトウェアパイプライニングが行われない。

(*2) SIMD長が可変な場合、SVEのベクトルレジスタを特定サイズとみなさずに翻訳して実行時にサイズを決定する。

(*3) -msve-vector-bits=scalableに指定する。本指定は、-msve-vector-bitsオプションのデフォルトである。

このため、SIMD化したループをソフトウェアパイプライニング化する場合は、-msve-vector-bits=512オプションの指定（固定なSIMD長に指定）が必要である。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
    - ソフトウェアパイプライニングの概念説明
    - ソフトウェアパイプライニングのコンパイルオプション
    - ソフトウェアパイプライニングに関する最適化情報と最適化メッセージ
      （FortranおよびC/C++ tradモードの場合）
    - ソフトウェアパイプライニングの例
    - 【補足】FortranおよびC/C++での各種指示行について
  - プリフェッチ
- 【付録】参考文献

▸ソフトウェアパイプライニングに関する最適化情報と最適化メッセージ
（FortranおよびC/C++ tradモードの場合）

■ ソフトウェアパイプライニングが行われた場合の例 [1/2]

- コンパイルリストには、以下の図の①～③が表示される。

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 8)
                    <<<    SOFTWARE PIPELINING(IPC: 2.50, ITR: 176, MVE: 14, POL: S)  ①
                    <<< Loop-information  End >>>
17      1       v       DO I=1,N
18      1       v         A(I) = A(I) + 1D0
19      1       v       ENDDO
```

② `jwd8204o-i  "d.F90", line 17:` ループにソフトウェアパイプライニングを適用しました。

③ `jwd8205o-i  "d.F90", line 17:` ループの繰返し数が176回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。

③は、ループ反復回数が

- 176回以上の場合はソフトウェアパイプライニングが適用されたループが実行される
- 176回より少ない場合は適用されないループが実行される

ことを示す。

▸ソフトウェアパイプライニングに関する最適化情報と最適化メッセージ
（FortranおよびC/C++ tradモードの場合）

■ ソフトウェアパイプライニングが行われた場合の例 [2/2]

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 8)
                    <<<    SOFTWARE PIPELINING(IPC: 2.50, ITR: 176, MVE: 14, POL: S)
                    <<< Loop-information  End >>>
17      1       v       DO I=1,N
18      1       v         A(I) = A(I) + 1D0
19      1       v       ENDDO
```

記号※	意味
IPC	ソフトウェアパイプライニングが適用されたループの、サイクル当たりの命令数（Instructions Per Cycle）の予測値です。
ITR	ソフトウェアパイプライニングが適用されたループが実行時に選択されるために、必要なループの繰返し数です。 翻訳時メッセージjwd8205o-iで出力される値と同じです。
MVE	ソフトウェアパイプライニングによるループの展開数です。
POL	使用された命令スケジューリングアルゴリズムを示します。「S」は小さなループに、「L」は大きなループに適したアルゴリズムが使用されたことを意味します。それぞれ、-Kswp_policy=small、-Kswp_policy=largeオプションを指定したときに使用されるアルゴリズムです。

※ C/C++ clangモードではSOFTWARE PIPELININGの各項目は表示されない。今後変わる可能性があります。

内容

■ ソフトウェアパイプライニングが行われない場合の例

- 以下の図のループ例ではIF文（分岐命令）を含んでいる。

⚠ 分岐命令のあるループでは、SIMD化が行われない場合、ソフトウェアパイプライニングも行われない。

- 本例の場合、!OCL NOSIMDを指定（※）しているのでSIMD化されない。

  ※ 指示行を有効にするために-Koclオプションを指定する必要がある。

- SIMD化されない場合、ソフトウェアパイプライニングを促進する指示行!OCL SWP（指示行の詳細は後の【補足】の小節を参照）を指定しても、ソフトウェアパイプライニングも行なわれず、④の「分岐命令があるため、ソフトウェアパイプライニングを適用できません」というメッセージが表示される。

```
41                          !OCL NOSIMD
42                          !OCL SWP
43      1                     DO I=1,N
44      2                       IF (M(I)) THEN
45      2                         A(I) = A(I) + 1D0
46      2                       ENDIF
47      1                     ENDDO
```

④ jwd8670o-i "a.F90", line 43: ループ内に分岐命令があるため、ソフトウェアパイプライニングを適用できません。

■ SIMD化され、さらにソフトウェアパイプライニングも行なわれる場合の例

- 前スライドのループ例の場合、!OCL NOSIMDを削除し、-Ksimd=autoおよび-Kswpが有効であると、SIMD化され、さらにソフトウェアパイプライニングも行なわれる。

```
Command line options : -O2 -Kocl,optmsg=guide -Nlst=t
                       -Kprefetch_nosequential -Knounroll -Kloop_nofusion
Effective options    : （中略）-Ksimd=auto（中略）-Kswp （以下、省略）
```

```
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<<    SOFTWARE PIPELINING(IPC: 1.60, ITR: 104, MVE: 6, POL: S)
                     <<< Loop-information  End >>>
33     1         v       DO I=1,N
34     2         v         IF (M(I)) THEN
35     2         v           A(I) = A(I) + 1D0
36     2         v         ENDIF
37     1         v       ENDDO
```

```
jwd6001s-i  "a.F90", line 33: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "a.F90", line 33: ループにソフトウェアパイプライニングを適用しました。
jwd8205o-i  "a.F90", line 33: ループの繰返し数が104回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
```

■ 指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果 [1/5]

■ 改善前ソースコードのコンパイルリスト

- 配列IFLAGにより、配列Aの添字が間接アドレス指定となっているループ例を取り上げる。
- IFLAG内の値が全て異なる場合には配列Aに依存関係が無く後方依存性が存在しないのでSIMD化は可能。しかし、コンパイラには配列IFLAGの中身が分からないのでSIMD化されない。
- ソフトウェアパイプライニングは適用されている。ただしサイクル当たりの命令数の予測値IPCが0.36と低いことに注意。

```
Command line options : -O2 -Kocl,optmsg=2 -Nlst=t
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SOFTWARE PIPELINING(IPC: 0.36, ITR: 6, MVE: 2, POL: S)
                     <<<    PREFETCH(HARD) Expected by compiler :
                     <<<      B, IFLAG
                     <<< Loop-information  End >>>
69     2       2s        DO I=1,N
70     2       2m          A(IFLAG(I)) = A(IFLAG(I)) + B(I)
71     2       2v        ENDDO
jwd8204o-i  "f2.F90", line 69: ループにソフトウェアパイプライニングを適用しました。
jwd8205o-i  "f2.F90", line 69: ループの繰返し数が3回以上の時、ソフトウェアパイプライニング
を適用したループが実行時に選択されます。
jwd6228s-i  "f2.F90", line 70: データの定義引用の順序が逐次実行と変わる可能性があるため、
このDOループはSIMD化できません。
```

■ 指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果 [2/5]

■ 改善後ソースコードのコンパイルリスト

- 配列Aに依存関係がないことをコンパイラに知らせるために、指示行「!OCL NORECURRENCE」を指定（※）すると、SIMD化され、またソフトウェアパイプライニングのIPC値が2.16に向上し、ソフトウェアパイプライニングも促進されると見込まれる。

※ 指示行を有効にするために-Koclオプションを指定する必要がある。

```
Command line options : -O2 -Kocl,optmsg=2 -Nlst=t
```

```
135     1               !OCL NORECURRENCE
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(IPC: 2.16, ITR: 208, MVE: 11, POL: S)
                        <<<    PREFETCH(HARD) Expected by compiler :
                        <<<      IFLAG, B
                        <<< Loop-information  End >>>
136     2       2v          DO I=1,N
137     2       2v            A(IFLAG(I)) = A(IFLAG(I)) + B(I)
138     2       2v          ENDDO
```

```
jwd6001s-i  "f2.F90", line 136: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "f2.F90", line 136: ループにソフトウェアパイプライニングを適用しました。
jwd8205o-i  "f2.F90", line 136: ループの繰返し数が208回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
```

■ 指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果 [3/5]

■ fapp計測結果（CPU性能解析レポートより） [1/3]

- 指示行!OCL NORECURRENCEの指定により、表より、経過時間は1.49倍高速化し、実行命令数は92%減少した。また、SIMD命令率の向上によりSIMD化の促進が確認される。

バージョン	!OCL NORECURRENCE の指定	経過時間（単位：秒）(*1)	「改善前」からの高速化率	実行命令数 (*1)	SIMD命令率
改善前	指定なし	1.38	-	8.50E+08	0.00%
改善後	指定あり	0.93	1.49	6.82E+07	92.5%

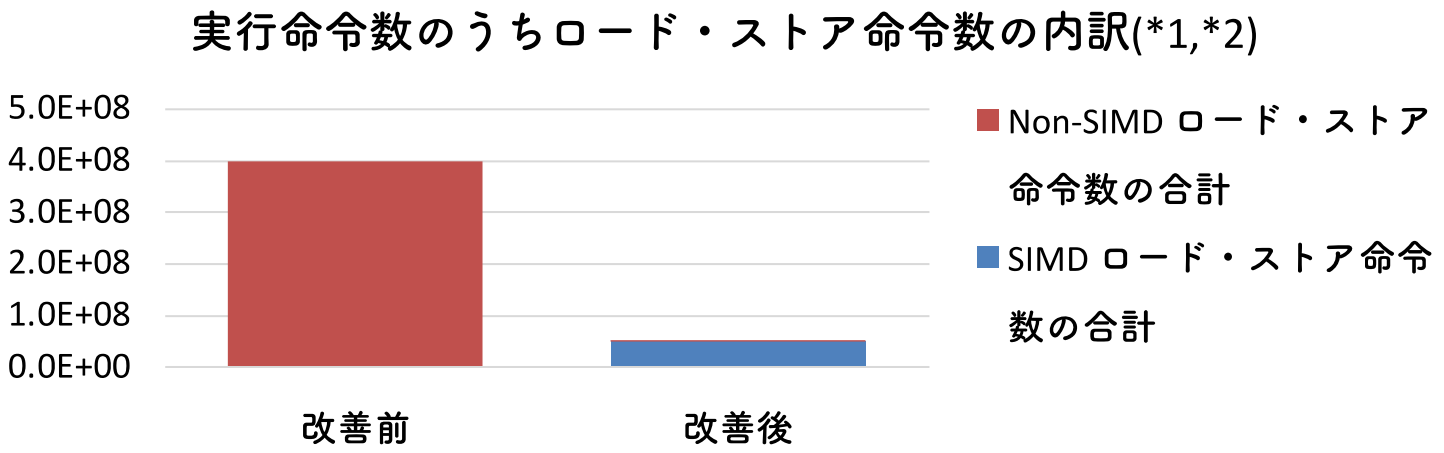
(*1) 配列要素数N=1000000、反復回数NLOOP=100の場合の累積数。

■ 指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果 [4/5]

■ fapp計測結果（CPU性能解析レポートより） [2/3]

- 実行命令数のうち、ロード・ストア命令数について、SIMD命令であるかNon-SIMD命令であるかに関して内訳をカウントしたものをグラフに示す。

- グラフより、Non-SIMD命令がSIMD命令に置き換わっていることからも、SIMD化の促進が確認される。



(*1) 配列要素数N=1000000、反復回数NLOOP=100の場合の累積数。

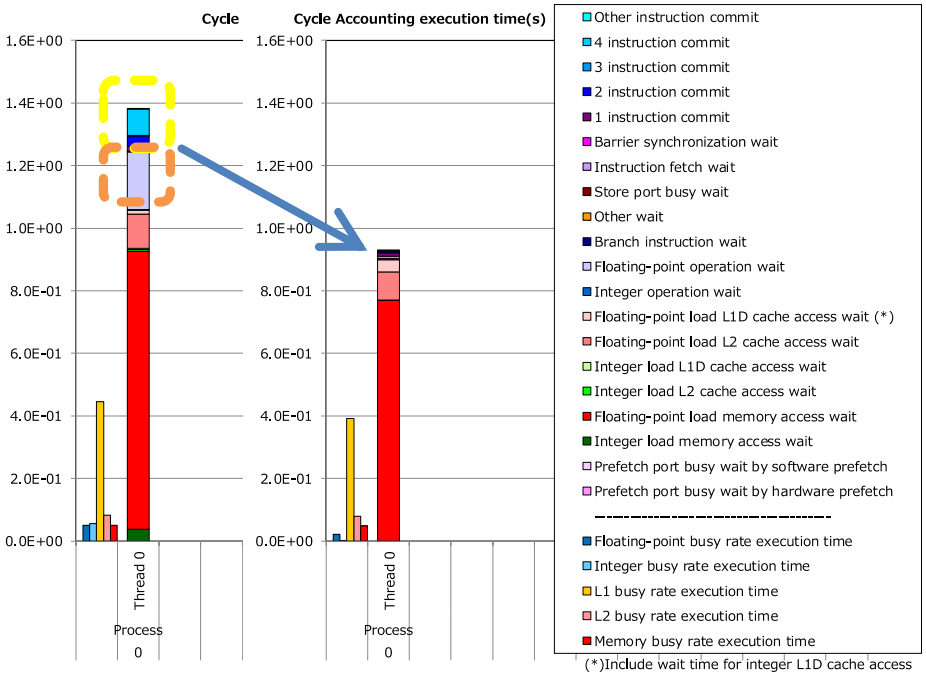
(*2) 実行された命令のうち、ロード・ストア命令以外の命令（演算やプリフェッチ命令等）は除いている。

■ 指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果 [5/5]

■ fapp計測結果（CPU性能解析レポートより） [3/3]

- 命令コミットの合計（主に青色系部分の合計）が86%減少した。
- 浮動小数点演算待ちが99%減少した。

!OCL NORECURRENCEなし（改善前）(*3)　　!OCL NORECURRENCEあり（改善後）(*3,*4)



(*3) -Kparallelオプションを付けると、指示行!OCL NORECURRENCEなしでは自動並列化されないが、この指示行ありでは自動並列化もされる。その場合、スレッド非並列とスレッド並列の比較になり、CPU単体の比較に不平等であるので、-Kparallelなしでスレッド非並列同士で比較することにした。

(*4) グラフの濃い赤部分のメモリアクセス待ちを改善する例については「プリフェッチ」節の「指示行!OCL PREFETCH_INDIRECTの効果をみるループ例」を参照して下さい。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
    - ソフトウェアパイプライニングの概念説明
    - ソフトウェアパイプライニングのコンパイルオプション
    - ソフトウェアパイプライニングに関する最適化情報と最適化メッセージ（FortranおよびC/C++ tradモードの場合）
    - ソフトウェアパイプライニングの例
    - 【補足】FortranおよびC/C++での各種指示行について
  - プリフェッチ
- 【付録】参考文献

■ 【補足】FortranおよびC/C++での各種指示行について [1/3]

■ ソフトウェアパイプライニングの促進に関連する指示行

指示行（Fortran）(*1)	指示行（C/C++ tradモード）(*1)	指示行（C/C++ clangモード）(*1)	意味
!OCL SWP (*2)	#pragma loop swp	#pragma fj loop swp	ソフトウェアパイプライニングの最適化を行うことを指示する。
!OCL SWP_WEAK	#pragma loop swp_weak		実行文の重なりを小さくしたソフトウェアパイプライニングの最適化を行うことを指示する。

(*1) 指示行を有効にするために、FortranおよびC/C++ tradモードの場合-Koclオプションを、C/C++ clangモードの場合-ffj-oclオプションを指定する必要がある。

(*2) !OCL SWPは-Kswp_strongオプション（以下の「参考」を参照）と同等の効果である。

■ 参考

■ -Kswp_strong

- ソフトウェアパイプライニング適用の条件を緩和し、ソフトウェアパイプライニングを促進することを指示する。
- 本オプションを指定することで翻訳メモリや翻訳時間が大幅に増加する場合がある。

■ -Kswp_weak

- ソフトウェアパイプライニングを調整し、ループ内の実行文の重なりを小さくすることを指示する。
- ソフトウェアパイプライニング適用に必要なループ繰返し数が小さくなるため、ループの繰返し数が翻訳時に不明であり、かつループ繰返し数が小さい場合に最適化の効果が期待できる。
- 本オプションを指定することで、実行文の重なりが小さくなるため、実行性能が低下する場合がある。

■ 【補足】FortranおよびC/C++での各種指示行について [2/3]

■ 命令スケジューリングアルゴリズムの選択に関連する指示行

指示行（Fortran）	指示行（C/C++ tradモード）	指示行（C/C++ clangモード）	意味
!OCL SWP_POLICY({AUTO|SMALL|LARGE})	#pragma loop swp_policy {auto|small|large}		ソフトウェアパイプライニングで使用する命令スケジューリングアルゴリズムの選択基準を指示する。
AUTO	auto		ループ毎に命令スケジューリングアルゴリズムを自動で選択する。
SMALL	small		小さなループ（例えば必要レジスタ数が少ないループ）に適した命令スケジューリングアルゴリズムを使用する。
LARGE	large		大きなループ（例えば必要レジスタ数が多いループ）に適した命令スケジューリングアルゴリズムを使用する。

■ 【補足】FortranおよびC/C++での各種指示行について [3/3]

■ 使用可能なレジスタ数の条件を変更する指示行

- 以下の三種類の指示行はソフトウェアパイプライニングで使用するレジスタ数の条件を変更することを指示する。レジスタ数に関する条件を変更することで、ソフトウェアパイプライニングの適用を調整できる。
- *n*は使用可能なレジスタ数の割合（百分率）を表す1~1000までの整数値である。レジスタが不足するためソフトウェアパイプライニングが適用されない場合、100より大きな値を*n*に指定することで、ソフトウェアパイプライニングが適用できることがある。

⚠ 本オプションを指定することで、レジスタのメモリへの退避・復元命令が変化し、実行性能が低下する場合がある。

指示行（Fortran）	指示行（C/C++ tradモード）	指示行（C/C++ clangモード）	意味
!OCL SWP_FREG_RATE(*n*)	#pragma loop swp_freg_rate *n*		浮動小数点レジスタおよびSVEのベクトルレジスタについて指示する。
!OCL SWP_IREG_RATE(*n*)	#pragma loop swp_ireg_rate *n*		整数レジスタについて指示する。
!OCL SWP_PREG_RATE(*n*)	#pragma loop swp_preg_rate *n*		プレディケートレジスタ（IF文の条件の真偽を格納する等フラグとして利用されるレジスタ）について指示する。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
- 【付録】参考文献

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ

    本節では、プリフェッチの考え方やプリフェッチするためのコンパイルオプション及び指示行の使い方について説明します。
    - データキャッシュ構成
    - プリフェッチの概念説明
    - プリフェッチのコンパイルオプション
    - プリフェッチに関する最適化情報（Fortranおよび`C/C++ trad`モードの場合）
    - プリフェッチの例
- 【付録】参考文献

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
    - データキャッシュ構成

      プリフェッチはデータキャッシュに対する操作であるため、まず予備知識として、データキャッシュの構成を説明します。
      - プリフェッチの概念説明
      - プリフェッチのコンパイルオプション
      - プリフェッチに関する最適化情報（FortranおよびC/C++ tradモードの場合）
      - プリフェッチの例
- 【付録】参考文献

■ 「富岳」のデータキャッシュの大きさ

1次（L1）キャッシュ	ウェイ数(*1)	4ウェイ	4ウェイの合計 64 KiB	各コアごとに存在
	ウェイ当たりバイト数	16 KiB		
	ウェイ当たりキャッシュライン数（=16 KiB/256 B）	64ライン		
	ライン当たり要素数［倍精度の場合（=256/8）］	32個		
	［単精度の場合（=256/4）］	64個		

2次（L2）キャッシュ	ウェイ数(*1)	16ウェイ	16ウェイの合計 8 MiB	各CMGごとに存在
	ウェイ当たりバイト数	0.5 MiB		
	ウェイ当たりキャッシュライン数（=0.5 MiB/256 B）	2,048ライン		
	ライン当たり要素数［倍精度の場合（=256/8）］	32個		
	［単精度の場合（=256/4）］	64個		

1次キャッシュ

	ウェイ0	ウェイ1	ウェイ2	ウェイ3
ライン0	A(1)	A(2049)	A(4097)	A(6145)
	:	:	:	:
	A(32)	A(2080)	A(4128)	A(6176)
:	:	:	:	:
:	:	:	:	:
:	:	:	:	:
ライン63	A(2017)	A(4065)	A(6113)	A(8161)
	:	:	:	:
	A(2048)	A(4096)	A(6144)	A(8192)

2次キャッシュ

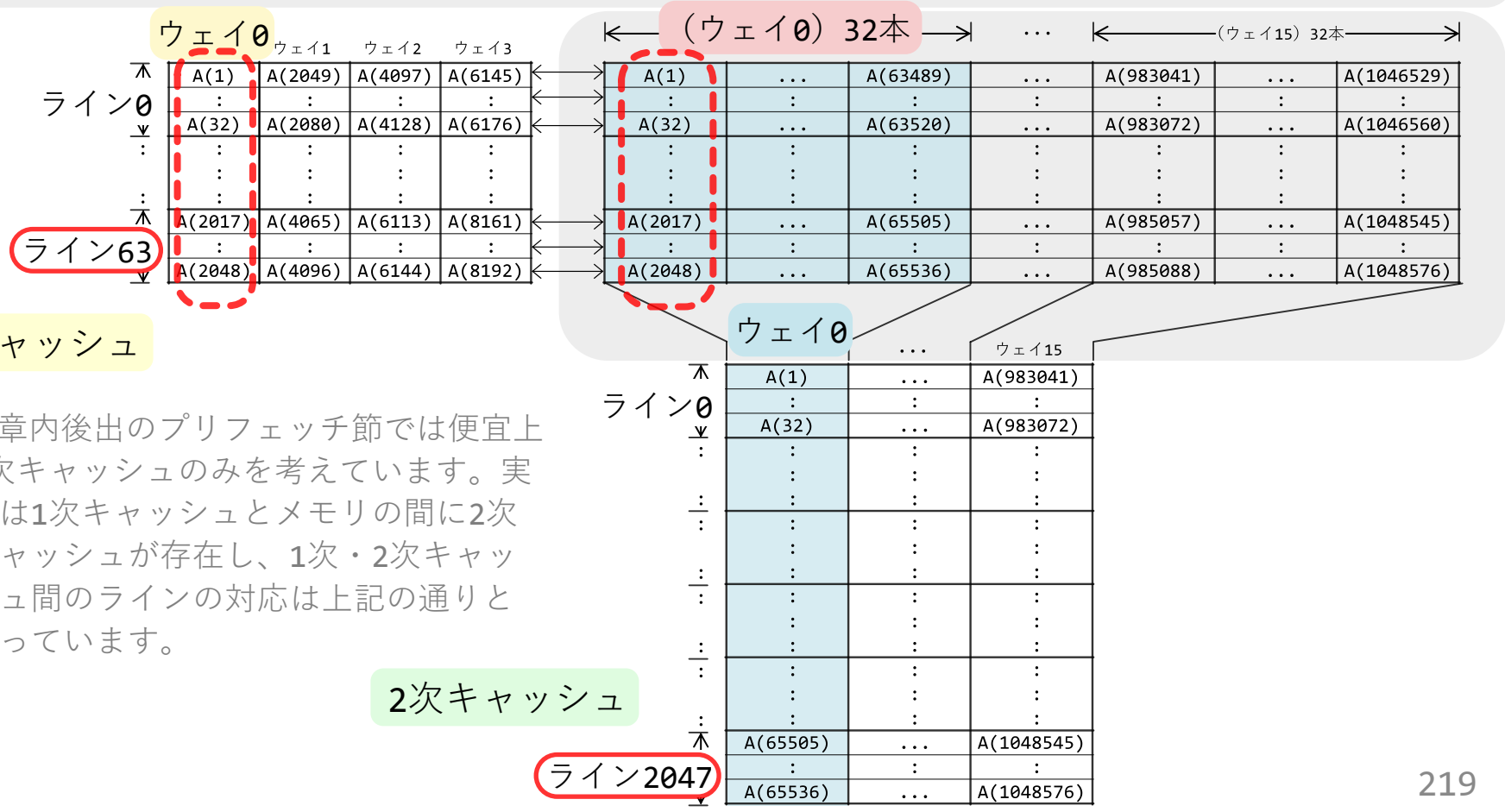
	ウェイ0	...	ウェイ15
ライン0	A(1)	...	A(983041)
	:	:	:
	A(32)	...	A(983072)
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
ライン2047	A(65505)	...	A(1048545)
	:	:	:
	A(65536)	...	A(1048576)

(*1) ウェイについては、後の「【補足】ウェイについて」を参照。

■1次キャッシュと2次キャッシュは1ウェイ当たりのライン数が異なる。

1次キャッシュ	ウェイ当たりキャッシュライン数	64ライン
2次キャッシュ	ウェイ当たりキャッシュライン数	2,048ライン

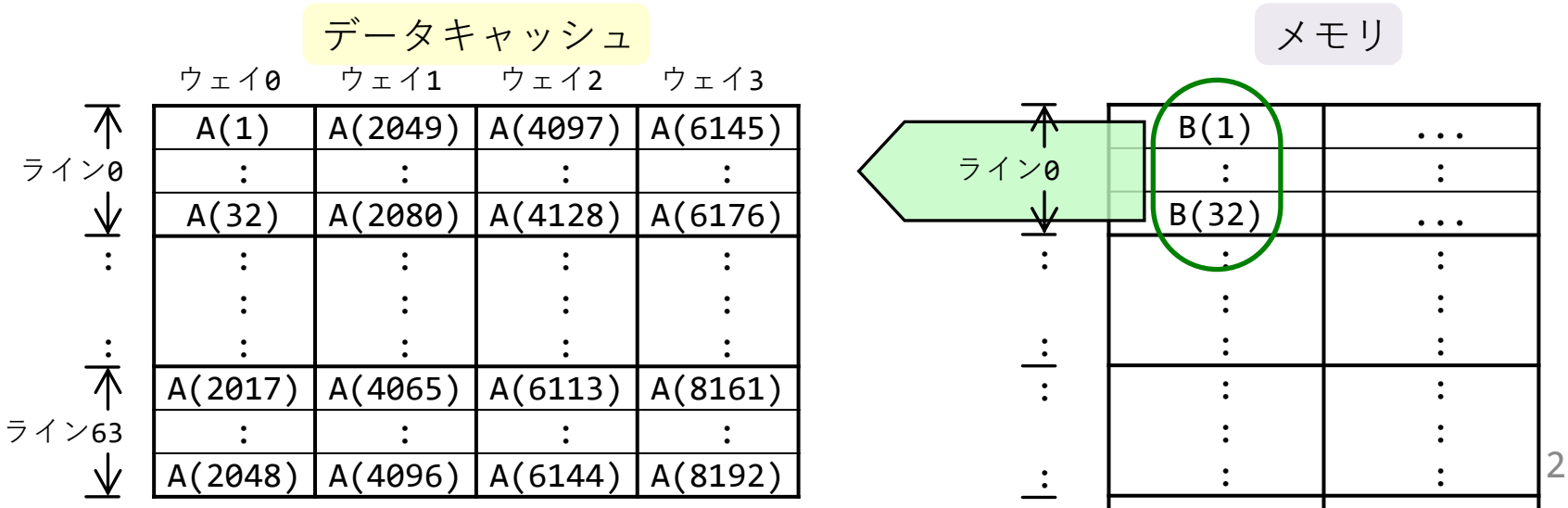
■2次キャッシュが図のように2048本のラインが64×32に並べてられていると見なして1次キャッシュと対応付ける(*2)。



(*2) 本章内後出のプリフェッチ節では便宜上1次キャッシュのみを考えています。実際は1次キャッシュとメモリの間に2次キャッシュが存在し、1次・2次キャッシュ間のラインの対応は上記の通りとなっています。

■ 【補足】ウェイについて

- ウェイとは、同じライン番号のキャッシュラインのデータを複数保持できるようにするための仕組みです。
- メモリとデータキャッシュの間のデータ転送では、データはキャッシュライン単位で転送される。
- データ（配列要素または変数）がどのキャッシュライン番号に属するかは、そのデータのメモリ上のアドレスから決まる。
- 以下では倍精度実数の配列要素B(1)が属するラインの番号が0であるとする。また、以下では、配列要素B(1)がデータキャッシュ上に存在せず、B(1)をメモリからデータキャッシュに転送する必要が生じた場合を考える※。　※ 説明の便宜上1次キャッシュのみを考え、単にデータキャッシュと呼ぶこととします。
  - B(1)を含む一つのライン単位のデータが、メモリからデータキャッシュに転送される。
  - 転送されたデータは、データキャッシュ上のライン番号0について、いずれかのウェイに空きがあれば、そのウェイのライン番号0のラインに納まる。
  - データキャッシュ上のどのウェイのライン0にも空きがなければ（下図のデータキャッシュのような場合）、一番過去にアクセスされたウェイのライン0のデータがキャッシュから追い出され、転送されたデータはそこに納まる。



内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
    - データキャッシュ構成
    - プリフェッチの概念説明
    - プリフェッチのコンパイルオプション
    - プリフェッチに関する最適化情報（FortranおよびC/C++ tradモードの場合）
    - プリフェッチの例
- 【付録】参考文献

- 「富岳」のデータキャッシュは1次（L1）キャッシュと2次（L2）キャッシュの2種類だが、以下の説明では単にデータキャッシュとする(*1)。

  (*1) 本節では説明の便宜上1次キャッシュのみを考え、単にデータキャッシュと呼ぶこととします。実際は1次キャッシュとメモリの間に2次キャッシュが存在し、1次・2次キャッシュ間のラインの対応は前節で述べた通りです。

- 下の図のループ例について、まずプリフェッチを行わない場合の動作を説明し、次にプリフェッチを行う場合の動作を説明する。

ループ例

```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```

プリフェッチを行わない場合の動作の説明　[1/5]

- まずA(1)が必要になるが、A(1)がデータキャッシュ上に（最初なので）存在しないため、キャッシュミスが発生し、

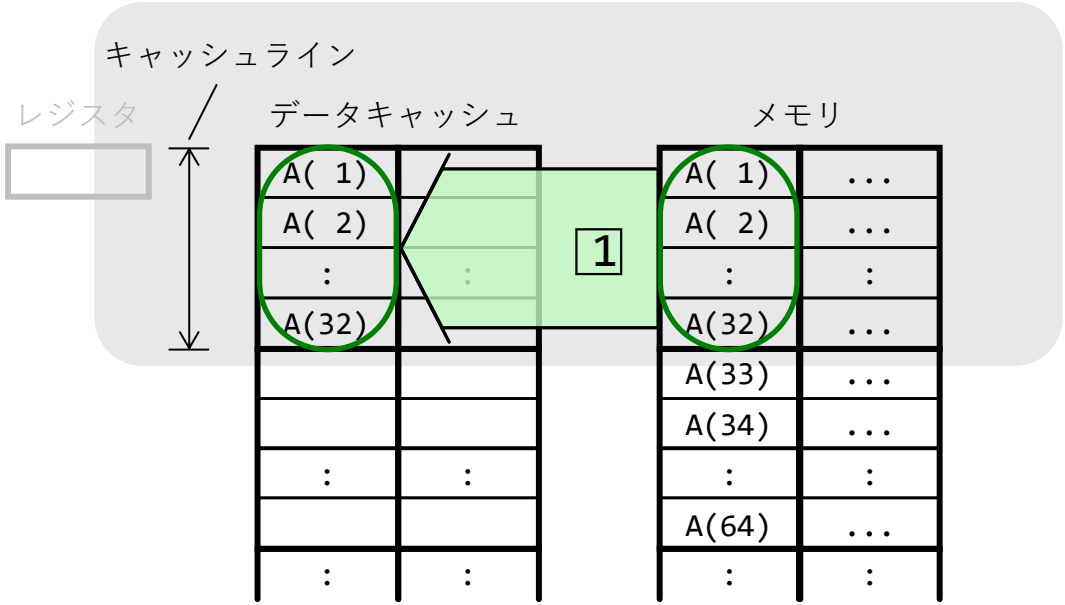
  1 に示すように A(1)～A(32)（キャッシュの1ライン分） が

  メモリからデータキャッシュに転送される。

  - この転送には時間がかかる。

ループ例

```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```

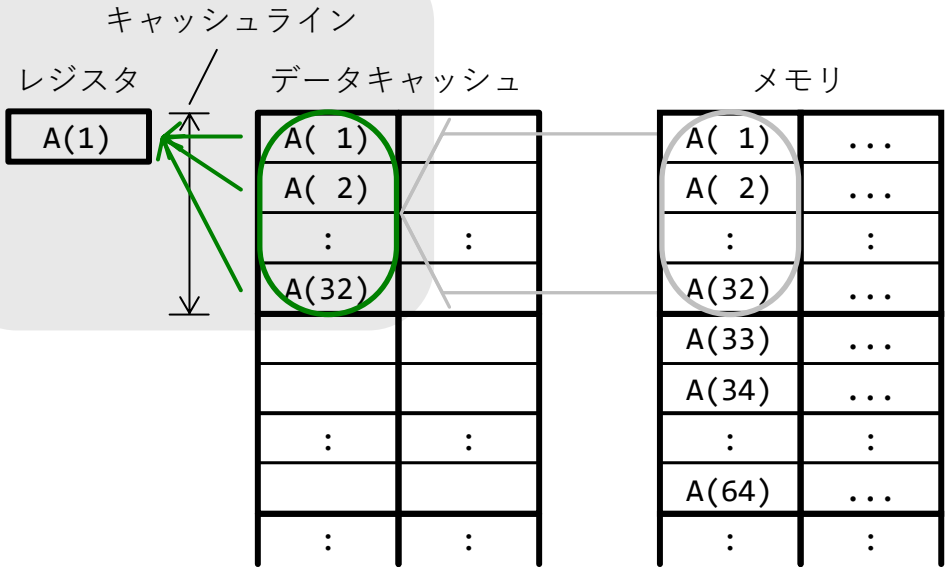


プリフェッチを行わない場合の動作の説明　[2/5]

- 配列Aの各要素が、A(1)、A(2)、・・・、A(32)の順にデータキャッシュからレジスタに転送され、A(1)～A(32)の計算が行われる。

ループ例

```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```

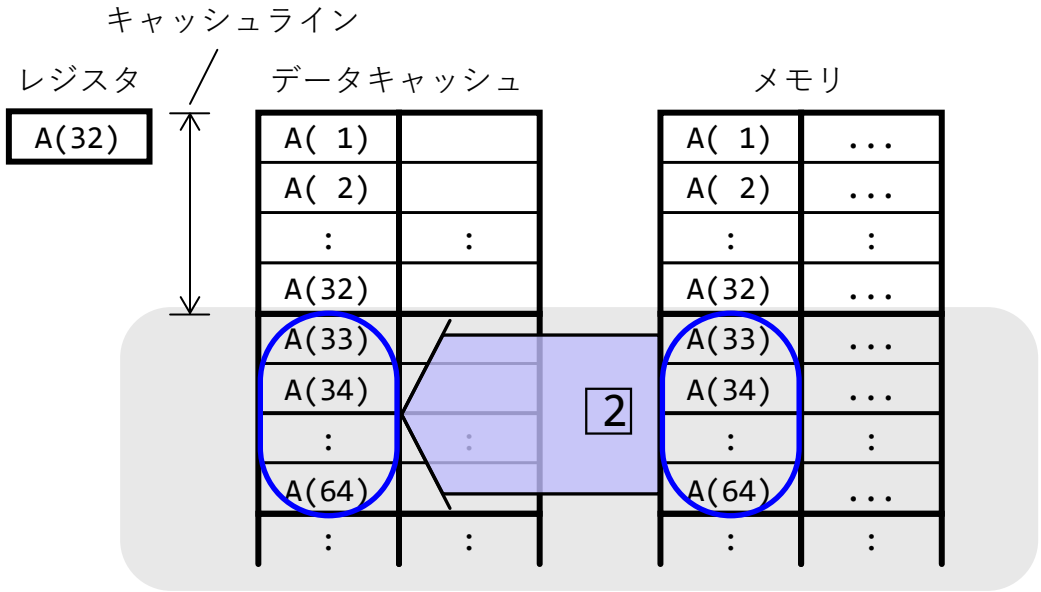


プリフェッチを行わない場合の動作の説明　[3/5]

- 次にA(33)が必要になるが、A(33)がデータキャッシュ上にないため、再びキャッシュミスが発生し、 2 の転送が行われる。

ループ例

```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```

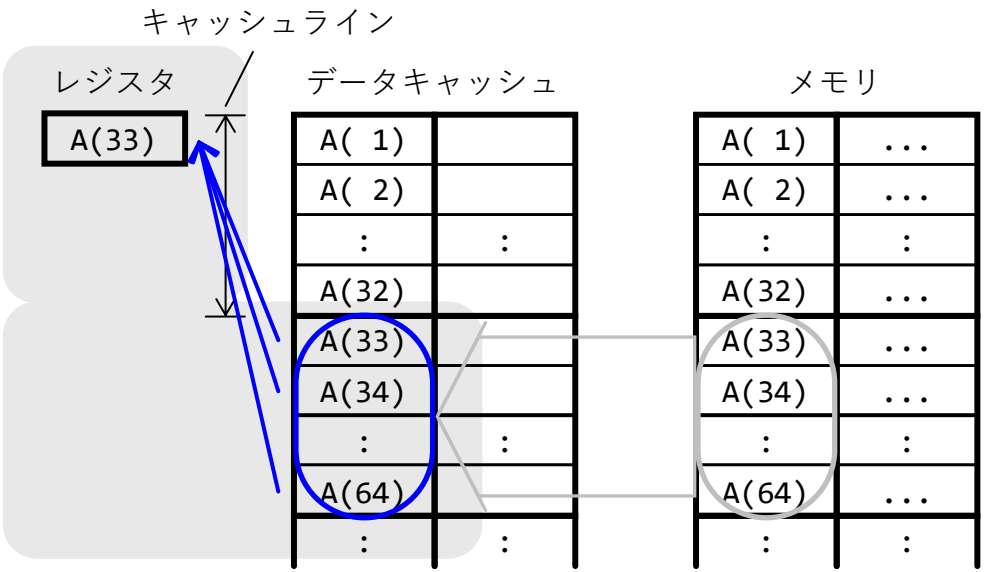


プリフェッチを行わない場合の動作の説明　[4/5]

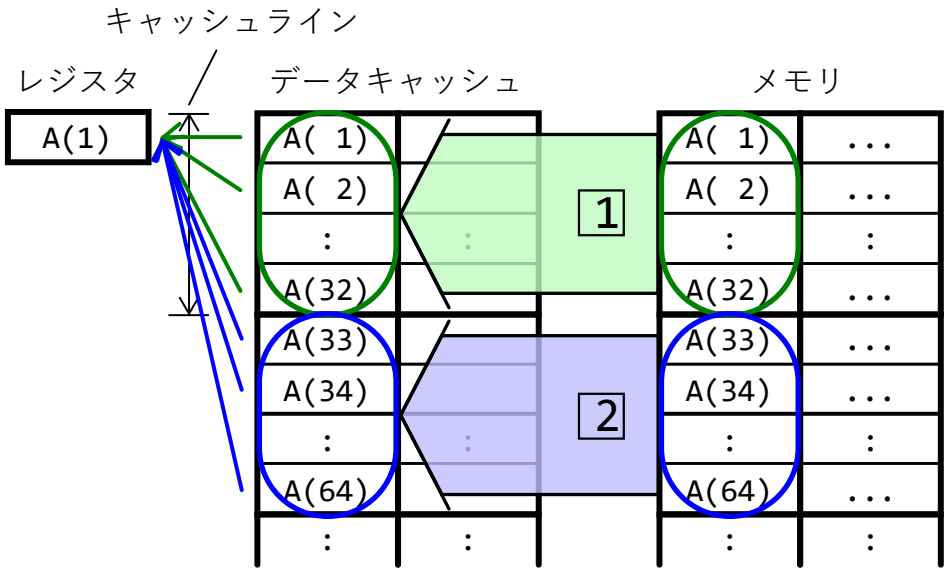
- A(33)、A(34)、・・・、A(64)の順に
  データキャッシュからレジスタに転送され、
  A(33)～A(64)の計算が行われる。

ループ例

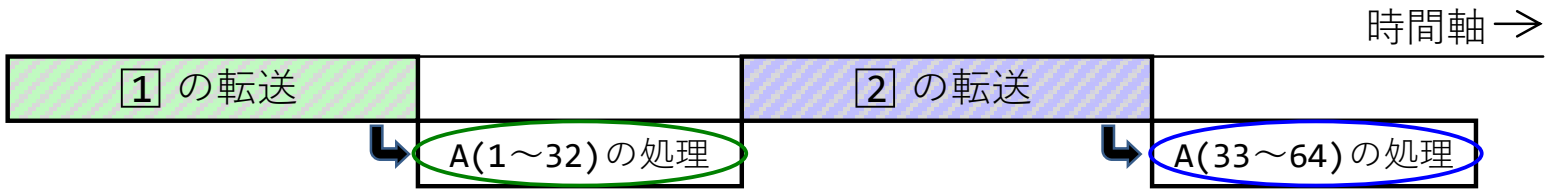
```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```



プリフェッチを行わない場合の動作の説明　[5/5]

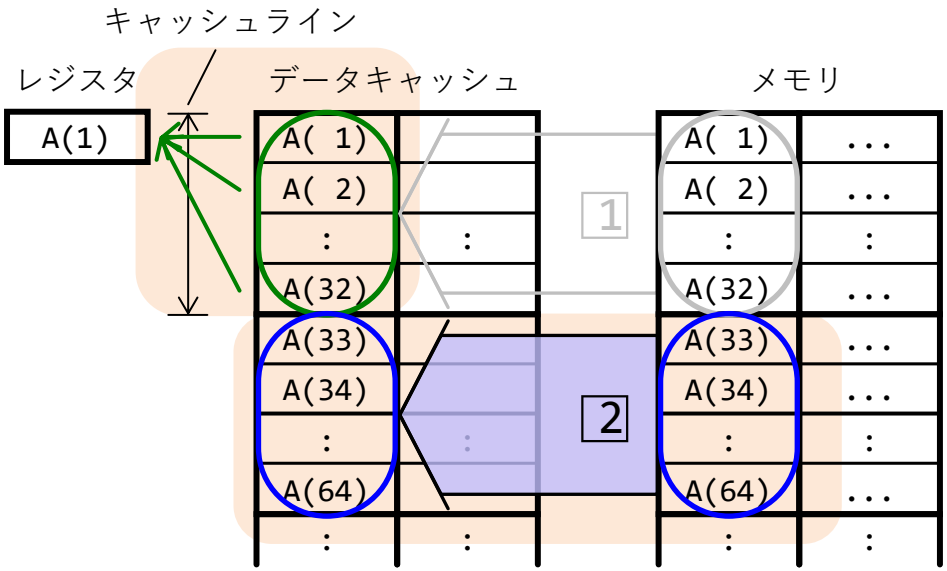


◆ 以上の動作のタイムチャートを以下の図に示す。

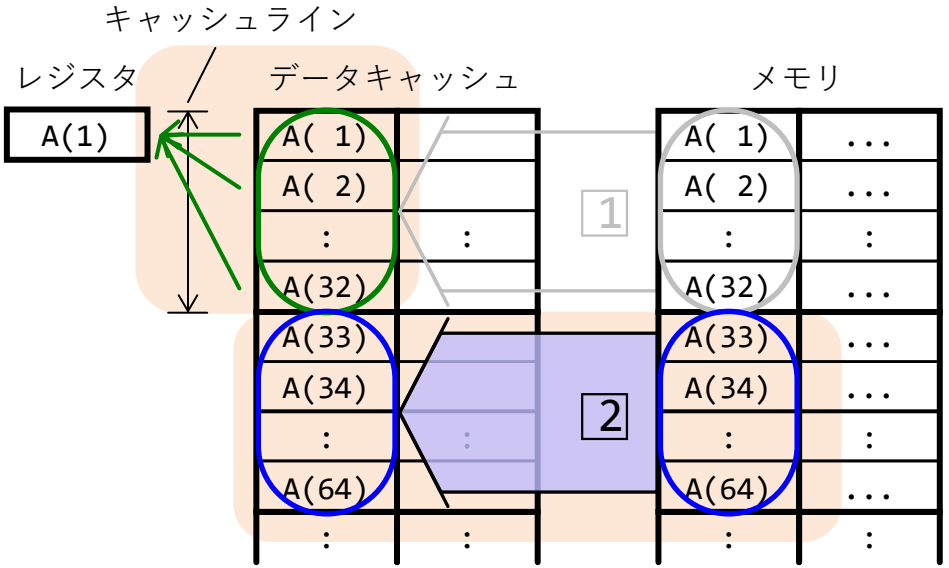


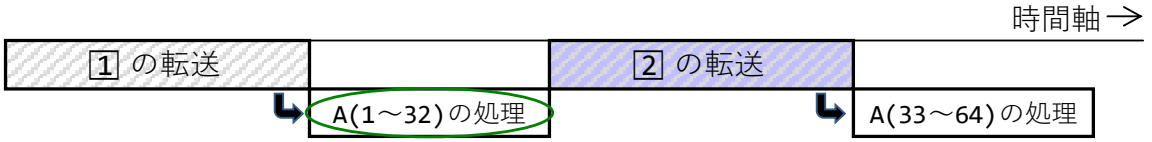
プリフェッチを行う場合の動作の説明　[1/2]

- 配列Aに対してプリフェッチの最適化が行われた場合、
  A(1)～A(32) の処理が行われている間に、
  その次に必要となる 2 の転送も同時に行われる。



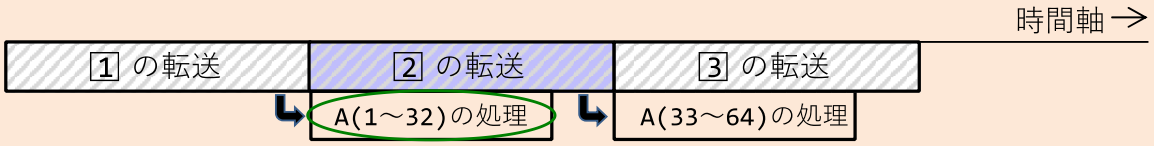
プリフェッチを行う場合の動作の説明　[2/2]





プリフェッチを行わない場合

- タイムチャートは右の図のようになり、時間のかかる 2 の転送時間をある程度隠蔽することができる。

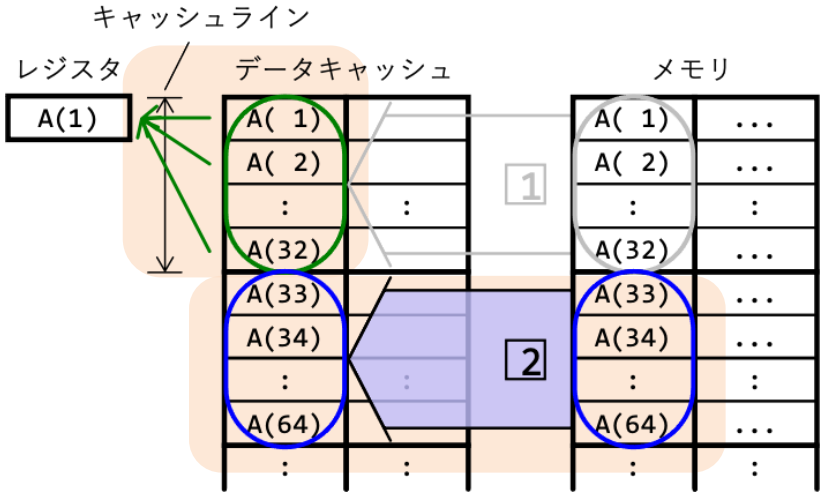


プリフェッチを行った場合

■ プリフェッチの種類

- ■ ハードウェアプリフェッチ：ハードウェアが自動的に行う方法。
- ■ ソフトウェアプリフェッチ：プログラム内にprefetch命令を生成する方法。
  - 以降の小節で説明するコンパイルオプションや指示行で行うプリフェッチはソフトウェアプリフェッチである。

■ プリフェッチについての注意事項



- プリフェッチの有無に関わらず、本例でA(33)のデータが必要になりキャッシュミスが起きる。

  プリフェッチはデータのメモリからの [2] の転送時間を隠蔽することが目的であり、キャッシュの利用効率向上によるキャッシュミス数の低減が目的ではない。

- プリフェッチを行わない場合
  - この時の [2] の転送はロード命令での転送であり、この時のメモリアクセスをデマンドアクセスという。また、この時のキャッシュミスをデマンドアクセスでのキャッシュミスという。

- プリフェッチを行った場合
  - この時の [2] の転送でのメモリアクセスを（ハードウェアまたはソフトウェア）プリフェッチによるメモリアクセスという。また、この時のキャッシュミスを（ハードウェアまたはソフトウェア）プリフェッチでのキャッシュミスという。

- CPU性能解析レポートでは、上記のキャッシュミスは、プリフェッチを行わない場合は「キャッシュミス（デマンド率）」に計上され、プリフェッチを行った場合は「キャッシュミス（ハードウェアまたはソフトウェアプリフェッチ率）」に計上される。
  - プリフェッチはキャッシュミスのデマンド率を低減することが目的であるといえる。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
    - データキャッシュ構成
    - プリフェッチの概念説明
    - プリフェッチのコンパイルオプション
    - プリフェッチに関する最適化情報（FortranおよびC/C++ tradモードの場合）
    - プリフェッチの例
- 【付録】参考文献

■ 誘導されるプリフェッチ関連オプション

■ -O2で誘導されるコンパイルオプション

- -O2を指定すると、プリフェッチに関して以下のオプションが誘導される。
- C/C++ tradモードのこれらのオプションはclangモードで対応するオプションに置き換わる。

下線付きがデフォルト

オプション（FortranおよびC/C++ tradモードの場合）	オプション（C/C++ clangモードの場合）	意味
-Kprefetch_cache_level={1|2|all}	-ffj-prefetch-cache-level={1|2|all}	どのキャッシュレベルにデータをプリフェッチするかを指示する。
-Kprefetch_sequential[={auto|soft}]	-ffj-prefetch-sequential[={auto|soft}]	ループ内で使用される連続的にアクセスされる配列に対して、ハードウェアプリフェッチを使用するか、prefetch命令を生成するかどうかを指示する。
auto	auto	ハードウェアプリフェッチを利用するか、prefetch命令を生成するかをコンパイラが自動的に選択する。
soft	soft	ハードウェアプリフェッチを利用せずに、prefetch命令を生成する。

- なお、-O2を指定すると、FortranおよびC/C++ tradモードの場合 -Kprefetch_strong および -Kprefetch_strong_L2 も有効になり、C/C++ clangモードの場合 -ffj-prefetch-strong および -ffj-prefetch-strong-L2 も有効になる。
- 参考：prefetch命令の属性
  「Strong属性」：ハードウェアは可能な限りプリフェッチのリクエストを完了させようとします。
  「Weak属性」：ハードウェアの資源に余裕がなければプリフェッチのリクエストは削除されます。

■ 【補足】誘導されないプリフェッチ関連オプション [1/6]

■ -Kfastあるいは-Ofastからは誘導されないコンパイルオプション [1/4]

- プリフェッチに関する以下のコンパイルオプションは-Kfast（FortranおよびC/C++ tradモードの場合）あるいは-Ofast（C/C++ clangモードの場合）からは誘導されない。従って-O3や-O2からも誘導されない。
- C/C++ tradモードのこれらのオプションはclangモードで対応するオプションに置き換わる。

後のループ例の節で、-Kprefetch_indirectに対応する指示行!OCL PREFETCH_INDIRECTの効果をみるループ例を示します。

オプション（FortranおよびC/C++ tradモードの場合）	オプション（C/C++ clangモードの場合）	意味
-Kprefetch_conditional	-ffj-prefetch-conditional	IF構文やCASE構文に含まれるブロックの中で使用される配列データに対して、prefetch命令を生成する最適化を行う。
-Kprefetch_indirect		ループ内で使用される間接的にアクセス（リストアクセス）される配列データに対して、prefetch命令を生成する最適化を行う。
-Kprefetch_infer		プリフェッチの距離が不明な場合でも連続アクセスのプリフェッチを出力することを指示する。

次スライドへ続く

■ 【補足】誘導されないプリフェッチ関連オプション [2/6]

■ -Kfastあるいは-Ofastからは誘導されないコンパイルオプション [2/4]

- プリフェッチに関する以下のコンパイルオプションは-Kfast（FortranおよびC/C++ tradモードの場合）あるいは-Ofast（C/C++ clangモードの場合）からは誘導されない。従って-O3や-O2からも誘導されない。
- C/C++ tradモードのこれらのオプションはclangモードで対応するオプションに置き換わる。

下線付きがデフォルト

オプション（FortranおよびC/C++ tradモードの場合）		オプション（C/C++ clangモードの場合）	意味
-Kprefetch_stride[={soft|hard_auto|hard_always}]		-ffj-prefetch-stride	ループ内で使用されるキャッシュのラインサイズ※よりも大きなストライドでアクセスされる配列データに対して、プリフェッチを実施するかどうかを指示する。※倍精度だと32要素、単精度だと64要素。
	soft		prefetch命令を生成して、プリフェッチを実施する。
	hard_auto		ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施する。キャッシュ上にないデータのみプリフェッチするようストライドプリフェッチャーを設定する。
	hard_always		ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施する。常にプリフェッチを行うようストライドプリフェッチャーを設定する。

次スライドへ続く

■ 【補足】誘導されないプリフェッチ関連オプション [3/6]

■ -Kfastあるいは-Ofastからは誘導されないコンパイルオプション [3/4]

- プリフェッチに関する以下のコンパイルオプションは-Kfast（FortranおよびC/C++ tradモードの場合）あるいは-Ofast（C/C++ clangモードの場合）からは誘導されない。従って-O3や-O2からも誘導されない。
- C/C++ tradモードのこれらのオプションはclangモードで対応するオプションに置き換わる。

オプション（FortranおよびC/C++ tradモードの場合）	オプション（C/C++ clangモードの場合）	意味
-Kprefetch_iteration=*N*	-ffj-prefetch-iteration=*N*	prefetch命令を生成する際、ループの*N*回転後に参照または定義されるデータを対象とすることを指示する。SIMD化が適用されるループの場合、*N*にはSIMD化された後のループの繰返し数を指定する。1次キャッシュ用。1 ≦ *N* ≦ 10000。
-Kprefetch_line=*N*	-ffj-prefetch-line=*N*	prefetch命令を生成する際、*N*キャッシュライン先に該当するデータをプリフェッチの対象とすることを指示する。1次キャッシュ用。1 ≦ *N* ≦ 100。

次スライドへ続く

■ 【補足】誘導されないプリフェッチ関連オプション [4/6]

■ -Kfastあるいは-Ofastからは誘導されないコンパイルオプション [4/4]

- プリフェッチに関する以下のコンパイルオプションは-Kfast（FortranおよびC/C++ tradモードの場合）あるいは-Ofast（C/C++ clangモードの場合）からは誘導されない。従って-O3や-O2からも誘導されない。
- C/C++ tradモードのこれらのオプションはclangモードで対応するオプションに置き換わる。

オプション（FortranおよびC/C++ tradモードの場合）	オプション（C/C++ clangモードの場合）	意味
-Kprefetch_iteration_L2=*N*	-ffj-prefetch-iteration-L2=*N*	prefetch命令を生成する際、ループの*N*回転後に参照または定義されるデータを対象とすることを指示する。SIMD化が適用されるループの場合、*N*にはSIMD化された後のループの繰返し数を指定する。2次キャッシュ用。1 ≦ *N* ≦ 10000。
-Kprefetch_line_L2=*N*	-ffj-prefetch-line-L2=*N*	prefetch命令を生成する際、*N*キャッシュライン先に該当するデータをプリフェッチの対象とすることを指示する。2次キャッシュ用。1 ≦ *N* ≦ 100。

■ 【補足】誘導されないプリフェッチ関連オプション [5/6]

■ 対応するコンパイルオプションのない指示行（Fortranの場合）

- プリフェッチに関する以下の指示行は対応するコンパイルオプションはない。

後のループ例の小節で、指示行!OCL PREFETCH_WRITEの効果をみるループ例を示します。

指示行（Fortranの場合）(*1)	意味
!OCL PREFETCH	コンパイラの自動prefetch機能を有効にする。自動プリフェッチ機能とは、コンパイラが自動的にprefetch命令を挿入するのに最適な位置を判断し、prefetch命令を生成する機能です(*2)。
!OCL NOPREFETCH	コンパイラの自動prefetch機能を無効にする。
!OCL PREFETCH_READ (name [,level={1|2}] [,strong={0|1}])	参照されているデータに対してprefetch命令を生成することを指示する。 nameには配列要素（または部分配列）を指定する。 本指示行はループの直前に指定するのではなく、文の直前に指定する。
!OCL PREFETCH_WRITE (name [,level={1|2}] [,strong={0|1}])	定義、または定義かつ参照されているデータに対してprefetch命令を生成することを指示する。 nameには配列要素（または部分配列）を指定する。 本指示行はループの直前に指定するのではなく、文の直前に指定する。

(*1) 指示行を有効にするために-Koclオプションを指定する必要がある。

(*2) 本指示行!OCL PREFETCHを指定したループに対し、下記のコンパイルオプションが指定されたのと同じ効果を持つ：-Kprefetch_cache_level=all -Kprefetch_sequential -Kprefetch_stride -Kprefetch_indirect -Kprefetch_conditional

■ 【補足】誘導されないプリフェッチ関連オプション [6/6]

■ 対応するコンパイルオプションのない指示行（C/C++ tradモードおよびclangモードの場合）

- プリフェッチに関する以下の指示行は対応するコンパイルオプションはない。

指示行（C/C++ tradモードの場合）(*1)	指示行（C/C++ clangモードの場合）(*3)	意味
`#pragma loop prefetch`	`#pragma fj loop prefetch`	コンパイラの自動prefetch機能を有効にする。自動プリフェッチ機能とは、コンパイラが自動的にprefetch命令を挿入するのに最適な位置を判断し、prefetch命令を生成する機能です。
`#pragma loop noprefetch`	`#pragma fj loop noprefetch`	コンパイラの自動prefetch機能を無効にする。

(*1) 指示行を有効にするために-Koclオプションを指定する必要がある。
(*3) 指示行を有効にするために-ffj-oclオプションを指定する必要がある。

内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
  - 【前提知識、予備知識】最適化メッセージ（コンパイルリスト）の出力方法
  - SIMD化
  - ソフトウェアパイプライニング
  - プリフェッチ
    - データキャッシュ構成
    - プリフェッチの概念説明
    - プリフェッチのコンパイルオプション
    - プリフェッチに関する最適化情報（Fortranおよび`C/C++ trad`モードの場合）
    - プリフェッチの例
- 【付録】参考文献

■ プリフェッチに関する最適化情報（FortranおよびC/C++ tradモードの場合）

```
<<< Loop-information Start >>>
<<<  [OPTIMIZATION]
<<<    PREFETCH(HARD) Expected by compiler :
<<<      B2, B1, B3, B4, B5, B6, B7, B8, B9, B10
<<<      B11, B13, B12, B14, B15, B16
<<<    PREFETCH(SOFT) : 2
<<<     SEQUENTIAL : 2
<<<      A: 2
<<< Loop-information  End >>>
    DO I = 1, N
      A(I) = B1(I) + B2(I) + （中略） + B15(I) + B16(I)
    ENDDO
```

記号	意味
PREFETCH(HARD) Expected by compiler:	連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用することを期待しprefetch命令を生成しないことを示します。
PREFETCH(SOFT) :	ループ中に存在するすべてのprefetch命令の個数を示します。
SEQUENTIAL : N	連続的にアクセスされる配列データに対するprefetch命令の個数を示します。
配列名 : N	配列ごとのprefetch命令の個数を示します。以下の項目についても同様です。
STRIDE : N	キャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対するprefetch命令の個数を示します。
INDIRECT : N	間接的にアクセス（リストアクセス）される配列データに対するprefetch命令の個数を示します。
SPECIFIED : N	最適化指示子PREFETCH_READまたはPREFETCH_WRITEの指定によって生成したprefetch命令の個数を示します。

※ C/C++ clangモードでは表示されない項目があり、今後変わる可能性があります。

内容

■ 指示行!OCL PREFETCH_INDIRECTの効果をみるループ例 [1/6]

以下のループについて、プリフェッチの効果を指示行!OCL PREFETCH_INDIRECTの指定の有無により調べた。

- ループ内の配列Aは間接的にアクセス（リストアクセス）されている。
  - 配列IFLAGには全て異なる値が格納されているとして、!OCL NORECURRENCEを指定している。
- -Kprefetch_indirectオプションは-Kfastからは誘導されない。そこでループに対して指示行!OCL PREFETCH_INDIRECTを指定することとした。
  - !OCL PREFETCH_INDIRECTの指定無し：配列Aのプリフェッチ無し。
  - !OCL PREFETCH_INDIRECTの指定有り：配列Aのプリフェッチ有り。

※ 指示行を有効にするために-Koclオプションを指定する必要がある。

```
  DOUBLE PRECISION::A(N),B(N)
!OCL NORECURRENCE
!OCL PREFETCH_INDIRECT
  DO I = 1, N
    A(IFLAG(I)) = A(IFLAG(I)) + B(I)
  ENDDO
```

■ 指示行!OCL PREFETCH_INDIRECTの効果をみるループ例 [2/6]

■ 配列Aのプリフェッチ無しの場合のコンパイルリスト

- コンパイルオプション：

```
Command line options ※ : -O2 -Kocl,optmsg=2 -Nlst=t
```

※ 配列Aのプリフェッチ無しの本例は「ソフトウェアパイプライニング」の節の「指示行追加によるSIMD化とソフトウェアパイプライニング促進の効果」のループ例と同一であり、その節での実行結果（スレッド非並列の実行結果）を本節での実行結果と比較する便宜のため、本節でもスレッド非並列で実行することとした。

- 以下のコンパイルリストには、配列Aのプリフェッチに関する最適化メッセージは表示されない。
- 配列IFLAGおよびBに対するハードウェアプリフェッチ適用のメッセージは表示されている。

```
135     1                !OCL NORECURRENCE
                       <<< Loop-information Start >>>
                       <<<  [OPTIMIZATION]
                       <<<    SIMD(VL: 8)
                       <<<    SOFTWARE PIPELINING(IPC: 2.16, ITR: 208, MVE: 11, POL: S)
                       <<<    PREFETCH(HARD) Expected by compiler :
                       <<<      IFLAG, B
                       <<< Loop-information  End >>>
136     2       2v         DO I=1,N
137     2       2v           A(IFLAG(I)) = A(IFLAG(I)) + B(I)
138     2       2v         ENDDO
```

■ 指示行!OCL PREFETCH_INDIRECTの効果をみるループ例 [3/6]

■ 配列Aのプリフェッチ有りの場合のコンパイルリスト

- 指示行「!OCL PREFETCH_INDIRECT」を指定することで、本例では配列Aのプリフェッチに関して①に示すように、間接アクセスされる配列Aに対してprefetch命令が生成されたことを示す最適化メッセージが表示される。

```
Command line options : -O2 -Kocl,optmsg=2 -Nlst=t
168     1                !OCL NORECURRENCE
169     1                !OCL PREFETCH_INDIRECT
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 8)
                        <<<    SOFTWARE PIPELINING(IPC: 1.71, ITR: 48, MVE: 2, POL: S)
                        <<<    PREFETCH(HARD) Expected by compiler :
                        <<<      B, IFLAG
                        <<<    PREFETCH(SOFT) : 8                                    ①
                        <<<     INDIRECT : 8
                        <<<      A: 8
                        <<< Loop-information  End >>>
170     2       2v          DO I=1,N
171     2       2v            A(IFLAG(I)) = A(IFLAG(I)) + B(I)
172     2       2v          ENDDO
```

■ 指示行!OCL PREFETCH_INDIRECTの効果をみるループ例 [4/6]

■ fapp計測結果（CPU性能解析レポートより） [1/3]

- 配列Aのプリフェッチ指定により、1.75倍高速化した。

配列Aのプリフェッチの有無	!OCL PREFETCH_INDIRECTの指定	経過時間（単位：秒）(*1)	「プリフェッチなし」からの高速化率
プリフェッチなし	指定なし	0.93	-
プリフェッチあり	指定あり	0.53	1.75

(*1) 逐次実行（スレッド非並列）、配列要素数N=1000000、反復回数NLOOP=100の場合の累積時間。

■ 指示行!OCL PREFETCH_INDIRECTの効果をみるループ例 [5/6]

■ fapp計測結果（CPU性能解析レポートより） [2/3]

- 配列Aのプリフェッチ指定により、ソフトウェアプリフェッチがかかり、L1およびL2キャッシュミスのデマンド率(*2)が低減した。

(*2) ロード命令やストア命令によるアクセスでのキャッシュミスの割合。

配列Aのプリフェッチの有無	L1Dミス数(*3)	L1Dミスの内訳比率 %（／L1Dミス数） デマンド率	ハードウェアプリフェッチ率	ソフトウェアプリフェッチ率
プリフェッチなし	1.23E+08	97.5%	2.49%	0.00%
プリフェッチあり	1.11E+08	6.31%	4.23%	89.5%

配列Aのプリフェッチの有無	L2ミス数(*3)	L2ミスの内訳比率 %（／L2ミス数） デマンド率	ハードウェアプリフェッチ率	ソフトウェアプリフェッチ率
プリフェッチなし	2.68E+07	64.4%	35.7%	0.00%
プリフェッチあり	2.70E+07	0.02%	17.2%	82.8%

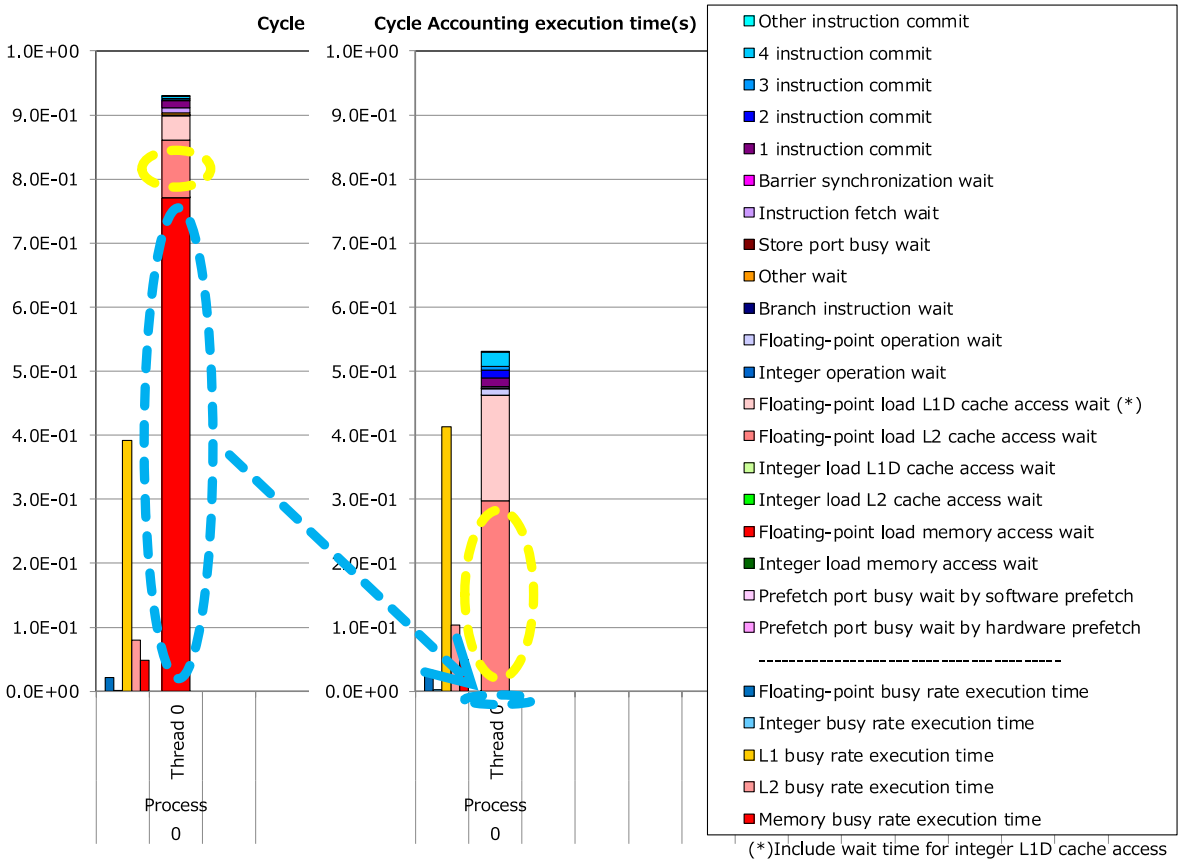
(*3) 逐次実行（スレッド非並列）、配列要素数N=1000000、反復回数NLOOP=100の場合の累積数。

■ 指示行!OCL PREFETCH_INDIRECTの効果をみるループ例 [6/6]

■ fapp計測結果（CPU性能解析レポートより） [3/3]

- 配列Aのプリフェッチ指定によりL2キャッシュミスのデマンド率が減少したことで、メモリアクセス待ちの時間が約99.96%減少した。

プリフェッチなし（改善前）　　プリフェッチあり（改善後）



■ 指示行!OCL PREFETCH_WRITEの効果をみるループ例 [1/5]

以下のループについて、プリフェッチの効果を指示行!OCL PREFETCH_WRITEの指定の有無により調べた。

- 指示行の書式は次スライドで示す。
- ループ内の配列Aは要素の間隔100でストライドアクセスされている。
  - 倍精度の場合一つのキャッシュラインに32要素入るため、倍精度の場合のストライド100は4ライン先に相当する。
- !OCL PREFETCH_WRITEの指定無し：配列Aのプリフェッチ無し。
- !OCL PREFETCH_WRITEの指定有り：配列Aのプリフェッチ有り。
  - 指示行ではA(I)の100要素先の配列要素A(I+100)をプリフェッチすることを指示する。

※ 指示行を有効にするために-Koclオプションを指定する必要がある。

```
  DOUBLE PRECISION::A(N)
  DO I = 1, N, 100
!OCL PREFETCH_WRITE(A(I+100))
    A(I) = A(I) + 1D0
  ENDDO
```

- コンパイルオプション：

```
Command line options : -Kfast,parallel -Kocl,optmsg=2 -Nlst=t
```

■ 指示行!OCL PREFETCH_WRITEの効果をみるループ例 [2/5]

■ !OCL PREFETCH_WRITE(*name*[,level={1|2}][,strong={0|1}])

- 定義、または定義かつ参照されているデータに対してprefetch命令を生成することを指示する。
- *name*には配列要素または部分配列を指定する。
- levelの省略値は1（L1キャッシュにデータをプリフェッチする）、strongの省略値は1（strong prefetchとする）。
- 参考：prefetch命令の属性
  「Strong属性」：ハードウェアは可能な限りプリフェッチのリクエストを完了させようとします。
  「Weak属性」：ハードウェアの資源に余裕がなければプリフェッチのリクエストは削除されます。

```
  DOUBLE PRECISION::A(N)
  DO I = 1, N, 100
!OCL PREFETCH_WRITE(A(I+100))
    A(I) = A(I) + 1D0
  ENDDO
```

■ 指示行!OCL PREFETCH_WRITEの効果をみるループ例 [3/5]

■ fapp計測結果（CPU性能解析レポートより） [1/3]

- 配列Aのプリフェッチ指定により、1.11倍高速化した。

配列Aのプリフェッチの有無	!OCL PREFETCH_WRITEの指定	経過時間（単位：秒）(*1)	「プリフェッチなし」からの高速化率
プリフェッチなし（改善前）	指定なし	0.97	-
プリフェッチあり（改善後）	指定あり	0.87	1.11

(*1) 配列要素数N=1000000、反復回数NLOOP=100000の場合の累積時間。

■ 指示行!OCL PREFETCH_WRITEの効果をみるループ例 [4/5]

■ fapp計測結果（CPU性能解析レポートより） [2/3]

- 配列Aのプリフェッチ指定により、L1キャッシュにソフトウェアプリフェッチがかかり、L1キャッシュミスのデマンド率(*2)が低減した。

(*2) ロード命令やストア命令によるアクセスでのキャッシュミスの割合。

配列Aのプリフェッチの有無	L1Dミス数(*3)	L1Dミスの内訳比率 %（／L1Dミス数） デマンド率	ハードウェアプリフェッチ率	ソフトウェアプリフェッチ率
プリフェッチなし	1.10E+09	99.6%	0.36%	0.01%
プリフェッチあり	1.07E+09	7.64%	0.38%	92.0%

- L2ミス数・L2ミスデマンド率が減少した。

配列Aのプリフェッチの有無	L2ミス数(*3)	L2ミスの内訳比率 %（／L2ミス数） デマンド率	ハードウェアプリフェッチ率	ソフトウェアプリフェッチ率
プリフェッチなし	5.18E+03	60.9%	44.5%	0.00%
プリフェッチあり	3.86E+03	51.8%	58.5%	0.00%

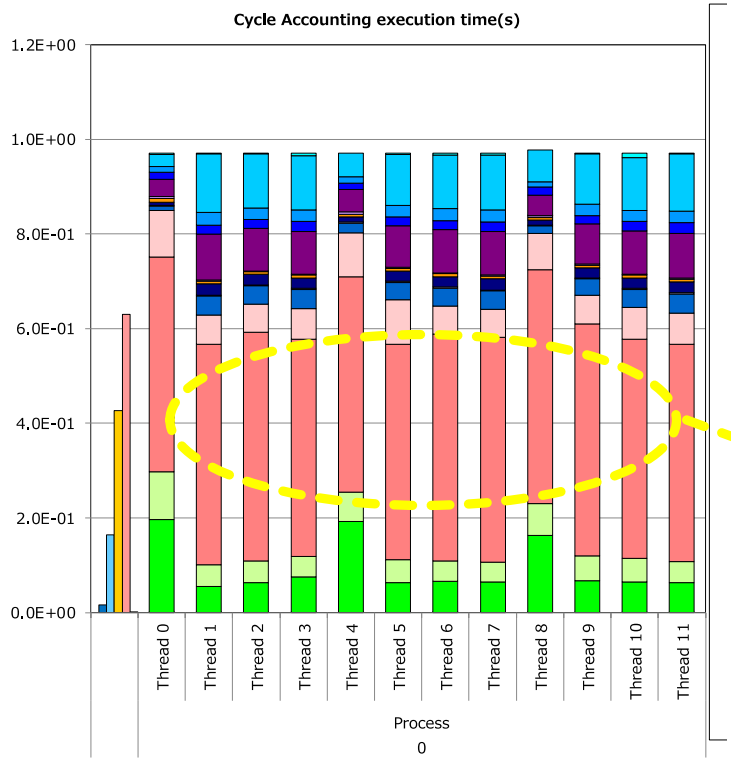
(*3) 配列要素数N=1000000、反復回数NLOOP=100000の場合の累積数。

■ 指示行!OCL PREFETCH_WRITEの効果をみるループ例 [5/5]

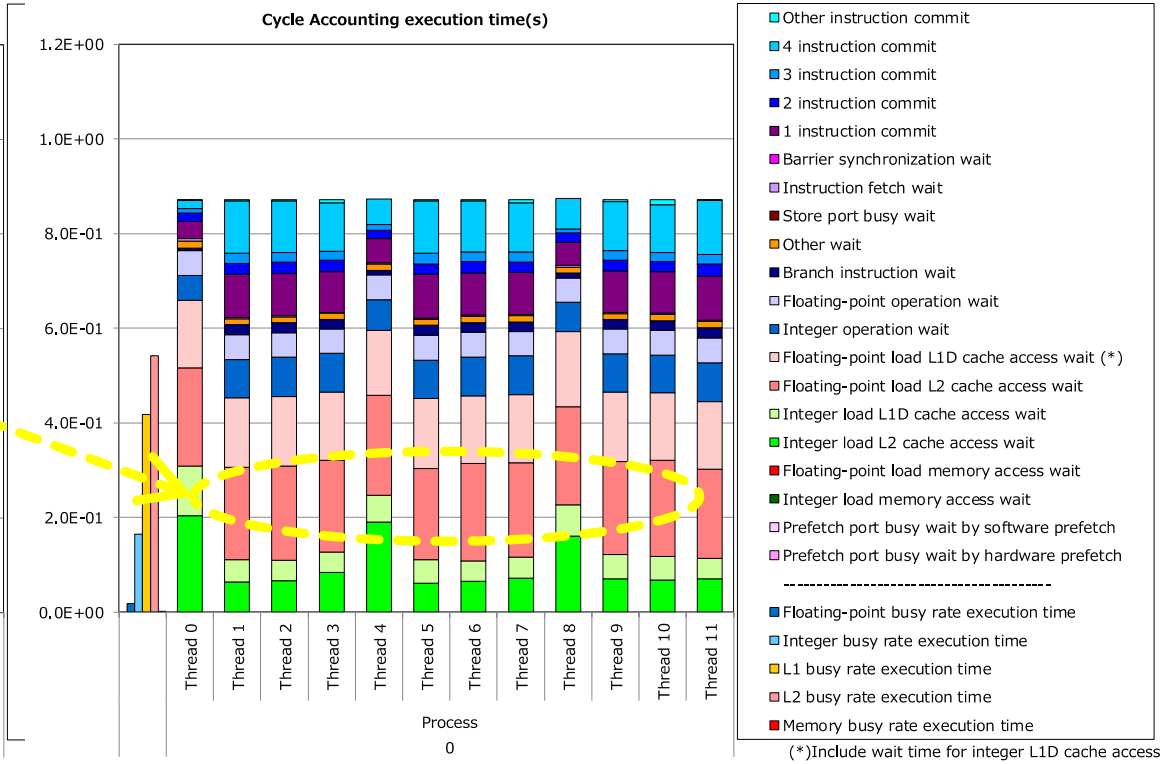
■ fapp計測結果（CPU性能解析レポートより） [3/3]

- 配列Aのプリフェッチ指定によりL1キャッシュミスデマンド率が減少したことで、L2キャッシュアクセス待ちの時間が約57%減少した。

プリフェッチなし（改善前）



プリフェッチあり（改善後）



内容

- 「富岳」の概略
- プロファイラ
- 重要な最適化手法
- 【付録】参考文献

[1] Fortran使用手引書
[2] C言語使用手引書
[3] C++言語使用手引書
[4] MPI使用手引書
[5] ジョブ運用ソフトウェア エンドユーザ向けガイド
[6] プロファイラ使用手引書
[7] プログラミングガイド チューニング編
[8] https://www.r-ccs.riken.jp/fugaku/system/

※ [1]～[7]はスーパーコンピュータ「富岳」のサイトにアクセス可能ならば以下のURLよりダウンロード可能です。

https://www.fugaku.r-ccs.riken.jp/docs/manuals_r01

以上