FUJITSU

Fujitsu Software
Technical Computing Suite V4.0L20

ジョブ運用ソフトウェア
エンドユーザ向けガイド

J2UL-2534-01Z0(09)
2024年3月

まえがき

本書の目的

本書では、Technical Computing Suiteに含まれる「ジョブ運用ソフトウェア」でのジョブの操作方法について説明します。

本書の読者

本書は、ジョブ運用ソフトウェアによるジョブの運用と管理をする管理者と、実際にジョブを操作するエンドユーザが対象です。

本書を読むためには、以下の知識が必要です。

- Linuxに関する基本的な知識
- 「ジョブ運用ソフトウェア 概説書」による、ジョブ運用ソフトウェアの概要についての知識
- Linuxのコンテナ仮想化技術およびDockerの知識
- Linuxの仮想化技術(KVM)および仮想管理用のユーティリティ(libvirt,virsh)の知識

本書の構成

本書は、次の構成になっています。

第1章 ジョブの仕組み

ジョブの仕組みについて説明します。

第2章 ジョブの操作方法

ジョブの操作方法について説明します。

付録A ジョブの情報

ジョブの情報に関するコマンドの表示について説明します。

付録B ジョブ実行に関する通知メッセージ

ジョブ実行に異常が起こった場合に、ユーザーにメールで通知されるメッセージの意味について説明します。

付録C Development Studio以外の MPI 処理系の実行について

Technical Computing SuiteのDevelopment Studio以外のMPI処理系のMPIプログラムを、ジョブ運用ソフトウェア上で実行するための注意事項、実行方法について説明します。

付録D ジョブに対する操作について

ジョブの状態とジョブに対する操作の可否の関係を説明します。

付録E ジョブ実行環境の利用について

ジョブ実行環境のイメージファイルの作成方法と、ジョブ実行環境利用時のトラブルの対処方法を説明します。

本書の表記について

ユーザーの表現

ジョブ運用ソフトウェアのユーザーには、システムの管理やジョブ運用をする管理者と、システムを利用してプログラムを実行するエンドユーザが存在します。本書では特に断りがなければ、「ユーザー」とはエンドユーザを指します。

単位の表現

本書では、単位を表現する際の接頭語は以下のとおりです。基本的にディスクサイズは10のべき乗、メモリサイズは2のべき乗で表現します。コマンドの表示や入力時の指定で注意してください。

接頭語	値	接頭語	値
K (kilo)	10^3	Ki (kibi)	2^{10}
M (mega)	10^6	Mi (mebi)	2^{20}
G (giga)	10^9	Gi (gibi)	2^{30}

接頭語	値	接頭語	値
T (tera)	10^{12}	Ti (tebi)	2^{40}
P (peta)	10^{15}	Pi (pebi)	2^{50}

機種名の表現

本書では富士通製CPU A64FXを搭載した計算機を「FXサーバ」、FUJITSU server PRIMERGYを「PRIMERGYサーバ」(または単に「PRIMERGY」)と表記します。
また、本書で説明する機能の一部には、対象機種によって仕様に差があります。このような機能の説明では、以下のように対象機種を略称で表記します。
[FX] : FXサーバを対象にした機能です。
[PG] : PRIMERGYサーバを対象にした機能です。

コマンドのパス名の表記

操作例で、ディレクトリ /bin、/usr/bin、/sbin、または /usr/sbin 配下にあるコマンドについては絶対パスで示していない場合があります。

マニュアル内のアイコンについて

本書では、以下のアイコンを使用しています。

注意

特に注意が必要な事項を説明しています。必ずお読みください。

参照

詳細な情報が書かれている参照先を示しています。

参考

ジョブ運用ソフトウェアに関連した参考記事を説明しています。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

商標

- Linux®は米国及びその他の国におけるLinus Torvaldsの登録商標です。
- Red Hat、Red Hat Enterprise Linuxは米国およびその他の国において登録されたRed Hat, Inc.の商標です。
- Intelは、アメリカ合衆国および/またはその他の国におけるIntel Corporationまたはその子会社の商標です。
- そのほか、本マニュアルに記載されている会社名および製品名は、それぞれ各社の商標または登録商標です。

出版年月および版数

版数	マニュアルコード
2024年3月 第1.9版	J2UL-2534-01Z0(09)
2023年3月 第1.8版	J2UL-2534-01Z0(08)
2022年3月 第1.7版	J2UL-2534-01Z0(07)
2021年11月 第1.6版	J2UL-2534-01Z0(06)

版数	マニュアルコード
2021年8月 第1.5版	J2UL-2534-01Z0(05)
2021年3月 第1.4版	J2UL-2534-01Z0(04)
2020年12月 第1.3版	J2UL-2534-01Z0(03)
2020年9月 第1.2版	J2UL-2534-01Z0(02)
2020年6月 第1.1版	J2UL-2534-01Z0(01)
2020年2月 初版	J2UL-2534-01Z0(00)

変更履歴

変更内容	変更箇所	版数
未書出しファイルの数が1000を超えた場合に出力される情報を追加しました。	2.3.3.10	第1.9版
TCP/IP 通信を行うジョブは 512 ノード以下で実行するという注意事項を追加しました。	1.6.3.1	第1.8版
Development Studio 以外の MPI 処理系の実行例について、記述を改善しました。	C.1	
会話型ジョブの場合、ジョブ統計情報 PERF COUNT が出力されない説明を追加しました。	A.2	第1.7版
ジョブ統計情報の FX サーバのノード消費電力に関する説明を追加しました。		
ジョブの性能情報の算出方法を変更しました。 ・ メモリ読出し要求数 ・ メモリ書出し要求数	A.3	
LLIO の機能で使用する、ジョブ内での環境変数の注意事項を追加しました。	2.1.2	第1.6版
LLIO 性能情報の採取時の注意事項を追加しました。	2.3.3.9	
McKernelに関する情報の参照先URLを変更しました。	E.1.2	第1.5版
MPIプロセスに設定される環境変数PLE_RANK_ON_NODEをサポートしました。	2.3.6.10	第1.4版
McKernelモードに関して以下の説明を追加しました。 ・ McKernel用のジョブメモリ量の求め方と、ホストOS用メモリが不足した場合の挙動 ・ McKernelの起動パラメーターの指定 また、KVMモードの説明に、仮想マシンイメージファイルへの書き込みを有効にしたジョブ実行環境における注意事項を追加しました。	2.3.8 A.2	
McKernelモードにおけるUDI指定について、説明を追加しました。	1.9 2.3.8 E.1.2 E.2.2	第1.3版
共有テンポラリ領域が空き容量不足になる場合について説明を追加しました。 1ノード当たりに必要な最低限の第2階層ストレージのキャッシュのサイズを訂正しました。	2.3.3.1	
ジョブ実行に関する通知メッセージ"Killed by OOM killer."の対処を改善しました。	付録B	
McKernelに関する情報の参照先URLを変更しました。	1.9 E.1.2	第1.2版
ジョブ統計情報の収集処理によるジョブ実行性能への外乱について、説明を修正しました。	2.1.1.8	
ジョブ内で設定される環境変数として、PJM_NODE_ALLOCATION_IO_MODEを追加しました。	2.1.2	

変更内容	変更箇所	版数
llio_transferコマンドの注意事項を修正しました。	2.1.4	
pjaclコマンドの表示項目に以下を追加しました。 ・ 項目definesのパラメーターdefault allocation-io-mode ・ 項目executeのパラメーターpjsub(no-io-exclusive)とpjsub(io-exclusive)	2.2.2	
ジョブへのノード割り当て方法として、I/O専有モードとI/O共有モードをサポートしました。	1.2.6 2.3.2.11	
McKernelモードとKVMモードにおけるジョブ統計情報について注意事項を追加しました。	A.2	
管理者が設定する必要がある起動設定ファイルの説明は、「ジョブ運用ソフトウェア 管理者向けガイド ジョブ管理編」へ移動しました。	E.1.1	
KVMモードの仮想マシンに関するRed Hat社のドキュメントの入手方法を変更しました。	E.1.3	
ノード内テンポラリ領域と共有テンポラリ領域が確保されるジョブの実行単位について、説明を改善しました。	1.13	第1.1版
ジョブ実行性能への外乱を避ける方法について記載しました。	2.1.1.8	
mpiexecコマンドの標準出力、標準エラー出力ファイルに関する説明を追加しました。	2.3.2.10 2.6.1	
共有テンポラリ領域とノード内テンポラリ領域において、ファイルの管理のために消費されるサイズについて記載しました。	2.3.3.1	
会話型ジョブを<Ctrl+c>の入力後に終了すると表示されるメッセージ"[INFO] PLE 0094"について説明を追加しました。	2.3.4.5	
大規模MPIジョブを実行するときの標準出力、標準エラー出力ファイルに関する推奨オプションの説明を追加しました。	2.3.6.9	
MPIプロセスに設定される環境変数PMIX_RANKの説明を追加しました。	2.3.6.10	
Dockerモードでは、UDI指定で使用するコンテナイメージは計算ノードから参照できる必要があることを注意事項に追加しました。	2.3.8	
McKernelモードでは、MPIジョブを実行する場合、ホストOS側に割り当てるメモリ量を増やさなければいけない場合があることを注意事項に追加しました。		
KVMモードでは、UDI指定で使用するコンテナイメージは計算ノードから参照できる必要があることと、ジョブはホームディレクトリ配下で投入することを注意事項に追加しました。		
ジョブがどの状態のときにpjalterコマンドでジョブのパラメーターを変更できるかについて注意事項を追加しました。 pjalterコマンドでジョブを実行するリソースユニットやリソースグループを変更するときの注意事項を追加しました。	2.5.4	
メッセージ"Reason: Node down."が通知されたときのジョブの状態と対処を修正しました。	付録B	
Dockerモードのイメージファイル作成で、LLIOに関して必要なOSパッケージとマウントポイントについて記載を追加しました。	E.1.1	
KVMモードの仮想マシンに関して、情報の参照先を追加しました。	E.1.3	
KVMモードの仮想マシンの要件として、"デバイス名に紐づかないコネクション"を追加しました。		
KVMモードを利用するために必要な資源管理エージェントパッケージおよび資源管理エージェントソースパッケージの入手方法の説明を改善しました。		

目　次

第1章 ジョブの仕組み 1
1.1 ジョブとは 1
1.2 ジョブの種類 2
1.2.1 ジョブモデル 2
1.2.1.1 通常ジョブ 2
1.2.1.2 バルクジョブ 3
1.2.1.3 ステップジョブ 3
1.2.1.4 ワークフロージョブ 5
1.2.1.5 マスタ・ワーカ型ジョブ 5
1.2.2 バッチジョブと会話型ジョブ 6
1.2.3 逐次ジョブと並列ジョブ 6
1.2.4 シングルノードジョブとマルチノードジョブ 7
1.2.5 ノード資源の割り当て方による分類 8
1.2.6 I/Oノードの割り当て方による分類 [FX] 8
1.3 ジョブの状態 9
1.4 ジョブIDとサブジョブID 12
1.5 ジョブの出力 12
1.6 ノード資源の割り当て 12
1.6.1 CPU 資源の階層構造について 12
1.6.2 ノードと仮想ノード 13
1.6.3 ノード単位での割り当て 13
1.6.3.1 FXサーバのノード単位での割り当て 13
1.6.3.2 PRIMERGYサーバのノード単位での割り当て 17
1.6.4 仮想ノード単位での割り当て 17
1.6.5 NUMA割り当てポリシー 18
1.7 ジョブの実行順序 19
1.7.1 グループやユーザー、リソースグループの優先度 19
1.7.2 ジョブ優先度 20
1.7.3 フェアシェア機能 20
1.7.4 ジョブの実行開始時刻 20
1.8 カスタム資源 20
1.9 ジョブ実行環境 22
1.10 ジョブACL機能 26
1.11 ジョブ統計情報 26
1.12 プロローグ・エピローグ機能 26
1.13 階層化ストレージ 26
1.14 コマンドAPI 27

第2章 ジョブの操作方法 29
2.1 ジョブの作成方法 29
2.1.1 ジョブの作成方法 29
2.1.1.1 バルクジョブの作成 30
2.1.1.2 ステップジョブの作成 30
2.1.1.3 ワークフロージョブの作成 31
2.1.1.4 マスタ・ワーカ型ジョブの作成 32
2.1.1.5 MPIプログラムを実行するジョブの作成 32
2.1.1.6 仮想ノードにおける逐次プログラムの一斉実行 [PG] 32
2.1.1.7 PAPI ライブラリを使用するジョブの作成 [FX] 32
2.1.1.8 ジョブ実行性能への外乱の対処 33
2.1.2 ジョブ内での環境変数 33
2.1.3 ユーザープログラムの作成方法 40
2.1.4 ジョブが利用できるファイルシステム 41
2.2 ジョブに関する情報の確認 42
2.2.1 リソースユニット、リソースグループの確認 42
2.2.2 制限情報の確認 44

2.2.3 資源の確認....56
2.3 ジョブの投入....61
2.3.1 ジョブの基本的な投入方法....61
2.3.2 ジョブ投入時のオプション....62
2.3.2.1 資源の指定....62
2.3.2.2 ノード資源の指定....65
2.3.2.3 ジョブの経過時間制限値の指定....67
2.3.2.4 カスタム資源の指定....69
2.3.2.5 資源量の上限を超えたジョブの動作について....70
2.3.2.6 ジョブ統計情報出力の指定....71
2.3.2.7 ジョブの自動再実行についての指定....72
2.3.2.8 実行開始時刻の指定....72
2.3.2.9 ジョブ優先度の指定....73
2.3.2.10 バッチジョブの標準出力、標準エラー出力ファイルの指定....73
2.3.2.11 I/Oノードを意識した資源の割り当て [FX]....74
2.3.2.12 Tofuインターコネクトのリンクダウンに対する動作の指定 [FX]....77
2.3.3 LLIOに関するパラメーターの指定 [FX]....78
2.3.3.1 第1階層ストレージのサイズ....80
2.3.3.2 第2階層ストレージから読み込んだファイルのキャッシング....81
2.3.3.3 ストライプ....81
2.3.3.4 非同期クローズ....82
2.3.3.5 ファイルの自動先読み....83
2.3.3.6 計算ノード内キャッシュのサイズ....83
2.3.3.7 第1階層ストレージへの書込み時のキャッシュのしきい値....84
2.3.3.8 計算ノードで読み込んだファイルのキャッシング....84
2.3.3.9 LLIO性能情報の採取....84
2.3.3.10 未書出しファイルの情報....86
2.3.4 ジョブの種類ごとの投入方法....87
2.3.4.1 バルクジョブの投入方法....87
2.3.4.2 ステップジョブの投入方法....88
2.3.4.3 ワークフロージョブの投入方法....92
2.3.4.4 マスタ・ワーカ型ジョブの投入方法....92
2.3.4.5 会話型ジョブの投入方法....92
2.3.5 ノード選択ポリシーの指定 [PG]....94
2.3.5.1 仮想ノード配置ポリシー....94
2.3.5.2 ランクマップ....98
2.3.5.3 ノード選択方式....98
2.3.5.4 割り当てノード優先度制御....99
2.3.5.5 実行モードポリシー....99
2.3.6 MPI ジョブの投入....100
2.3.6.1 プロセスの形状の指定 [FX]....100
2.3.6.2 生成するプロセス数の指定....101
2.3.6.3 ランクに対するノードの割り当てルール....104
2.3.6.4 rank-map-bychip パラメーター....105
2.3.6.5 rank-map-bynode パラメーター....106
2.3.6.6 rankmap に指定するノード割り当て順序 [FX]....106
2.3.6.7 rank-map-hostfile パラメーター [FX]....110
2.3.6.8 mpiexecコマンドの --vcoordfile オプション [FX]....112
2.3.6.9 mpiexecコマンドの標準出力/標準エラー出力 [FX]....113
2.3.6.10 MPIプロセスに設定される環境変数 [FX]....116
2.3.7 MPIジョブの実行指定例....117
2.3.7.1 1次元のノード形状で1つのジョブを実行する場合....117
2.3.7.2 3次元のノード形状で1つのジョブを実行する場合....117
2.3.7.3 1つのジョブでプログラムを複数回実行する場合....118
2.3.7.4 rank-map-bynode パラメーターとrank-map-hostfile パラメーターを指定した1ノード複数プロセスジョブの実行例....119
2.3.7.5 rank-map-bychip パラメーターとrank-map-hostfile パラメーターを指定した1ノード複数プロセスジョブの実行例....120
2.3.7.6 MPMD モデルの MPI プログラムの実行方法....121

2.3.7.7 MPMD モデルの MPI プログラムに対するランク指定 121
2.3.7.8 同一ノード上で複数のMPIプログラムを実行する場合(静的プロセス) [FX] 122
2.3.7.9 同一ノード上で複数のMPIプログラムを実行する場合(動的プロセス) [FX] 123
2.3.7.10 プログラムのリモート実行 [FX] 127
2.3.7.11 ハイブリッド並列プログラムの実行方法 129
2.3.8 ジョブの実行環境の指定 130
2.4 ジョブの状況確認 135
2.4.1 ジョブの一覧表示 135
2.4.2 ジョブ統計情報の表示 140
2.4.3 ジョブのノード利用状況の表示 141
2.4.4 ジョブの終了の確認 143
2.5 ジョブの操作 144
2.5.1 ジョブの削除 144
2.5.2 ジョブへのシグナル送信 145
2.5.3 ジョブの固定と固定解除 145
2.5.4 ジョブのパラメーター変更 146
2.6 ジョブの結果の確認 147
2.6.1 ジョブ実行結果の参照 147
2.6.2 ジョブ統計情報の出力 149
2.7 ノードがダウンした場合のジョブへの影響 149

付録A ジョブの情報 151
A.1 pjstat や pjstat -v の出力 151
A.2 ジョブ統計情報の出力 157
A.3 ジョブの性能情報の算出について [FX] 160

付録B ジョブ実行に関する通知メッセージ 161

付録C Development Studio以外の MPI 処理系の実行について 165
C.1 MPI 処理系ごとの注意事項と実行例 165
C.2 プロセスへの CPU 資源のバインド [PG] 168
C.3 NUMA メモリの割り当てポリシーの設定 [PG] 172
C.4 ラッパーコマンドmpiexec.tcs_intelを使用したMPIプログラム実行 [PG] 172
C.5 ジョブで利用できる CPU 資源の範囲の設定 [PG] 173

付録D ジョブに対する操作について 175

付録E ジョブ実行環境の利用について 179
E.1 ジョブ実行環境のイメージファイルの作成 179
E.1.1 Docker モードの場合 179
E.1.2 Mckernel モードの場合 180
E.1.3 KVM モードの場合 180
E.2 トラブルシューティング 182
E.2.1 ジョブがPJMコード28で終了した 182
E.2.2 ジョブがPJMコード29で終了した 182
E.2.3 ジョブがPJMコード140で終了した 183

第1章 ジョブの仕組み

ジョブ運用ソフトウェアは、ユーザーが作成したプログラムを「ジョブ」という単位で処理します。ここでは、ジョブについて説明します。

1.1 ジョブとは

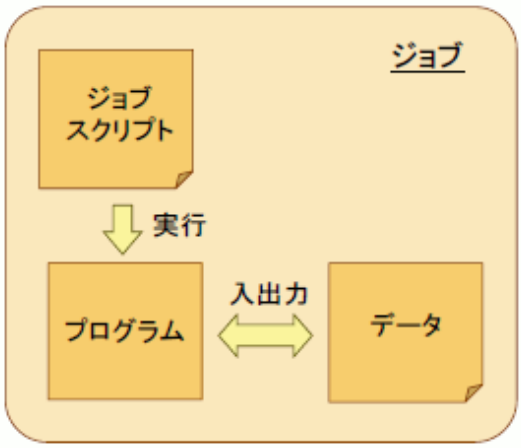
ジョブ運用ソフトウェアを導入したシステムでは、ユーザーはプログラムを直接実行するのではなく、ジョブ運用ソフトウェアのジョブ運用管理機能に実行を依頼します。ジョブ運用管理機能は実行依頼を受けたプログラムに対し、必要な計算機資源を確保し、プログラムを実行します。

このプログラムを処理する単位がジョブです。ジョブは、プログラムとその入出力データ、およびプログラムの実行方法を記述したジョブスクリプトから構成されます。

表1.1 ジョブの構成要素

構成要素	説明
プログラム	ユーザーが用意する実行可能なプログラムです。例えば、関連ソフトウェアのDevelopment Studioを使ってコンパイルした実行ファイルです。
データ	プログラムの入力および出力です。例えば、プログラムへの入力パラメーターが格納されたデータファイルや処理結果が格納される出力ファイル、プログラムの表示結果です。
ジョブスクリプト	プログラムの実行方法を記述したシェルスクリプトです。ジョブ運用ソフトウェアはこのジョブスクリプトに基づいてジョブを実行します。

図1.1 ジョブの構成



ジョブの実行は以下のような流れになります。

1. ユーザーがジョブ実行に必要なファイルを用意し、ログインノード上に配置します。
2. ユーザーはジョブ運用管理機能に対し、ジョブの実行を依頼します。これを「ジョブの投入」と呼びます。
   ジョブはpjsub コマンドでジョブスクリプトを指定して投入します。
3. ジョブ運用管理機能は、ジョブに計算機資源を割り当て、ジョブスクリプトを実行します。
4. ジョブスクリプトの内容に従って、プログラムが実行され、実行結果を出力します。
5. ジョブスクリプトが終了すると、ユーザーにメールで通知します(ジョブ投入時にメール通知を指示した場合だけ)。
6. ユーザーは、ジョブの実行結果をログインノード上で確認します。

参照

上記はジョブの実行についての大まかな流れです。詳細については "第2章 ジョブの操作方法" を参照してください。

1.2 ジョブの種類

ジョブは、それが必要とする計算機資源やプログラムの種類などによって、いくつかに分類されます。

ジョブ運用ソフトウェアでは、ジョブを以下のように分類しています。

表1.2 ジョブの種類

分類	ジョブの種類
ジョブの構造による分類 (ジョブモデル)	通常ジョブ
	バルクジョブ
	ステップジョブ
	ワークフロージョブ
	マスタ・ワーカ型ジョブ
ジョブの実行形態による分類 (ジョブタイプ)	バッチジョブ
	会話型ジョブ
並列度による分類	逐次ジョブ
	並列ジョブ
必要とする計算機資源(ノード資源)による分類	シングルノードジョブ
	マルチノードジョブ
ノード資源の割り当て方による分類	ノード割り当てジョブ (ノード専有ジョブ)
	仮想ノード割り当てジョブ (ノード専有ジョブ、ノード共有ジョブ)
I/Oノードの割り当て方による分類 [FX]	I/O専有ジョブ
	I/O共有ジョブ

注意

本書では、特に断りがない場合、「ノード」とはジョブを実行する計算ノードを指します。

以降では、ジョブの分類と種類についてそれぞれ説明します。

1.2.1 ジョブモデル

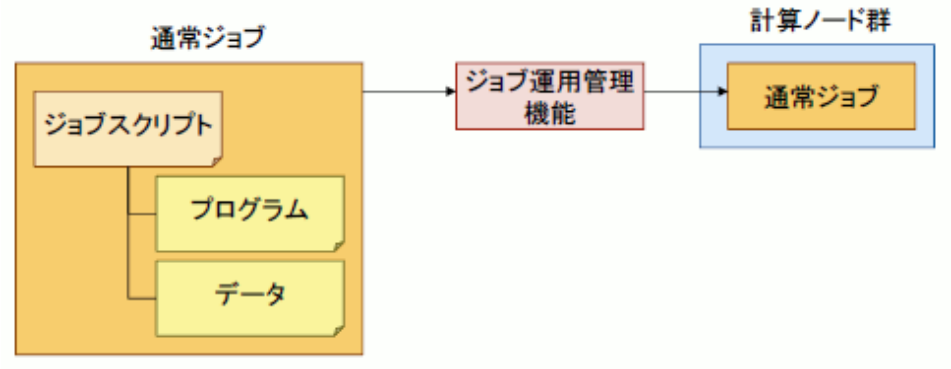
ジョブの構造による分類を「ジョブモデル」と呼びます。ここでは、各ジョブモデルの特徴を説明します。

1.2.1.1 通常ジョブ

通常ジョブは、最もシンプルな構造のジョブです。

1つのジョブスクリプトを実行し、それが終了するとジョブが終了します。

図1.2 通常ジョブ



1.2.1.2 バルクジョブ

バルクジョブは、複数の同じ通常ジョブを同時に投入し、実行するジョブです。

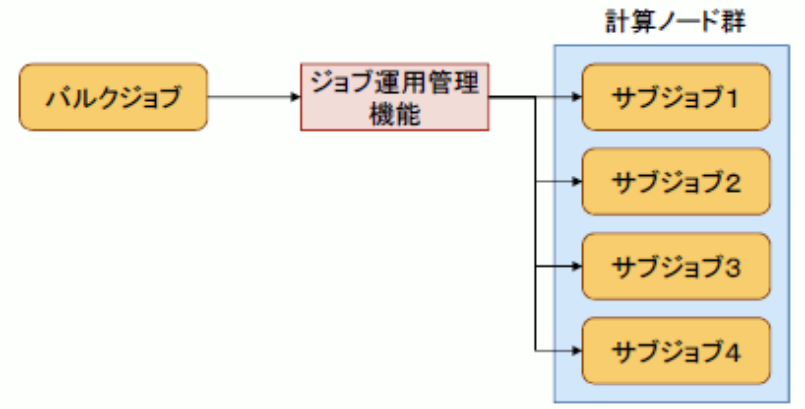
例えば、ジョブのパラメーターを変えて、それぞれの実行結果を確認したい場合、通常ジョブであれば、ユーザーがジョブを1つ1つ投入します。しかし、バルクジョブを使えば、1度に複数パターンの投入ができます。

バルクジョブで同時に実行されるジョブスクリプトの処理単位を「サブジョブ」と呼びます。バルクジョブのサブジョブには、1ジョブ内での通し番号 (0 から 999999) が設定されます。この番号を「バルク番号」と呼びます。

注意

バッチジョブかつ通常ジョブだけ、バルクジョブのサブジョブにできます。

図1.3 バルクジョブ



1.2.1.3 ステップジョブ

ステップジョブは、実行順序や依存関係がある複数のジョブのまとまりです。

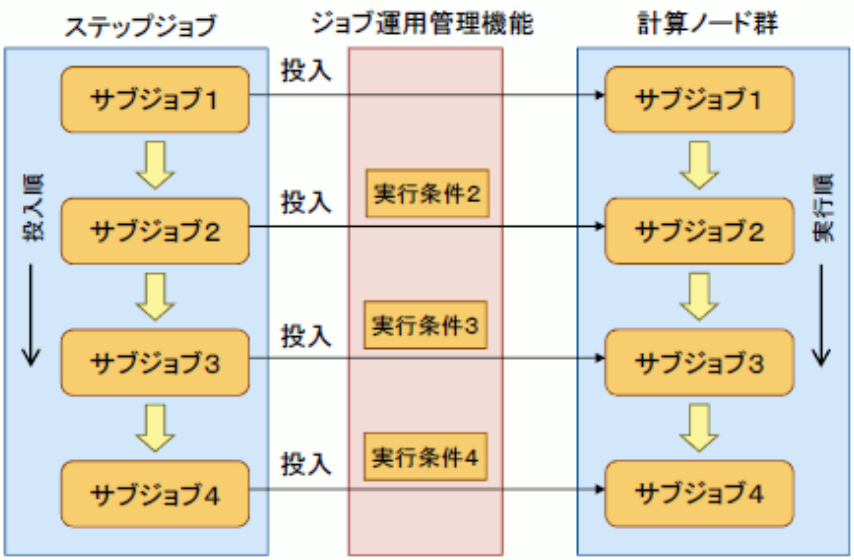
例えば、あるジョブの実行結果に応じて、別のジョブを実行するかどうかが決まる場合、通常ジョブであれば、ユーザーが判断して、ジョブを投入します。しかし、ステップジョブを使えば、ジョブの投入時に実行条件や順序を指定することで、特定のジョブの実行結果に応じて自動的に処理させることができます。

ステップジョブとして関連づけられたそれぞれのジョブは「サブジョブ」と呼びます。ステップジョブのサブジョブは、1ジョブ内での通し番号 (0 から 65534) が設定されます。この番号を「ステップ番号」と呼びます。

注意

バッチジョブかつ通常ジョブだけ、ステップジョブのサブジョブにできます。

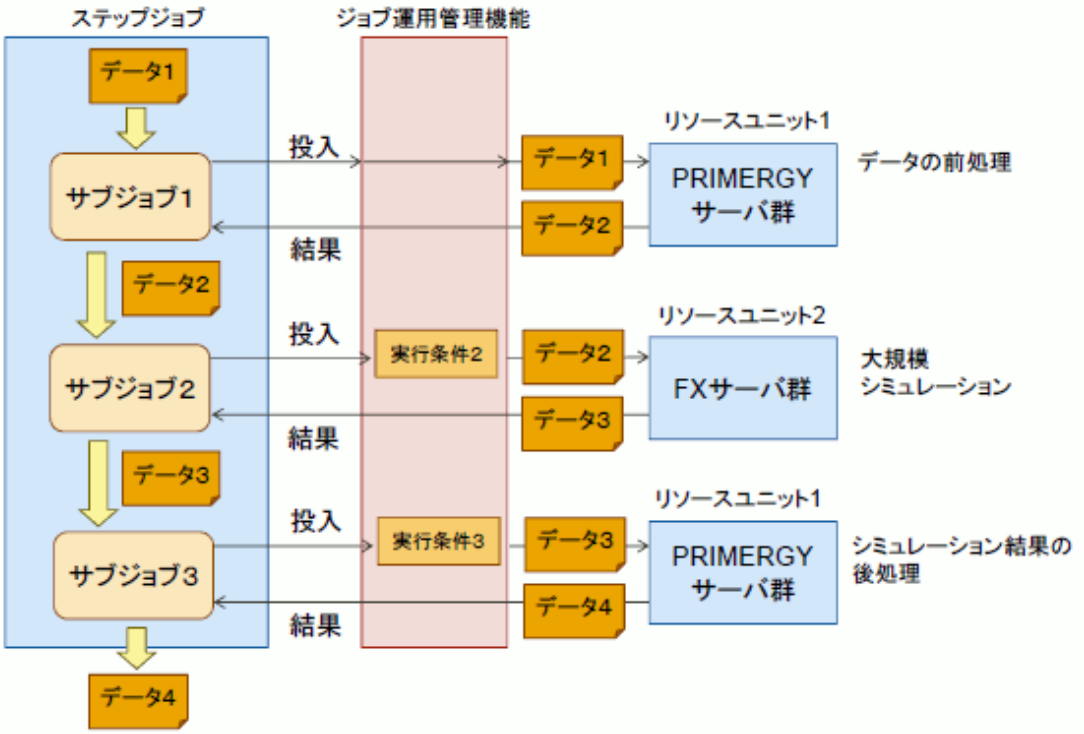
図1.4 ステップジョブ



ステップジョブを投入するリソースユニットやリソースグループは、サブジョブごとに異なるリソースユニットやリソースグループを指定できます。例えば、FXサーバのリソースユニットとPRIMERGYサーバのリソースユニットで構成されたクラスタで、以下のように、各アーキテクチャーに最適化されたアプリケーションをステップジョブのサブジョブとして投入し、一連の処理として実行できます。

1. サブジョブ1
   PRIMERGYサーバ上で大規模シミュレーション用のデータの前処理をする。
2. サブジョブ2
   前処理されたデータを入力として、FXサーバ上で大規模シミュレーション用アプリケーションを実行する。
3. サブジョブ3
   大規模シミュレーションの結果を入力として、PRIMERGYサーバ上で後処理(データ解析/可視化など)をする。

図1.5 ステップジョブのサブジョブを異なるリソースユニットに投入する例



1.2.1.4 ワークフロージョブ

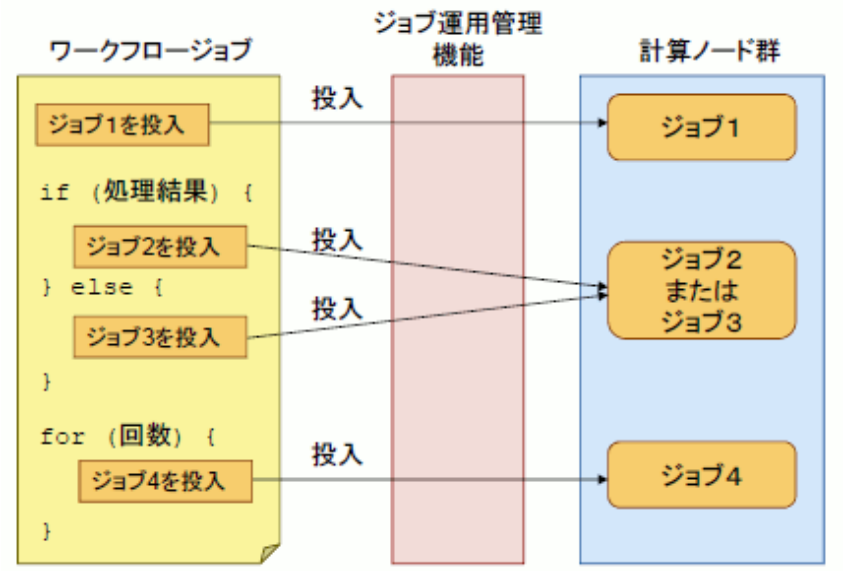
ワークフロージョブは、ジョブ単位での実行制御(条件分岐や繰り返し)をする複数のジョブのまとまりです。

これもステップジョブと同様に、あるジョブの実行結果に応じて、次に実行するジョブが決まるような場合で利用しますが、実行制御をするシェルスクリプトの内容次第で、ステップジョブより自由度が高い制御ができます。

参考

ワークフロージョブは、正確にはジョブの種類ではなく、複数のジョブの投入をシェルスクリプトによってユーザーが制御する方法のことです。

図1.6 ワークフロージョブ



1.2.1.5 マスタ・ワーカ型ジョブ

マスタ・ワーカ型ジョブとは、通常ジョブ、ステップジョブ、バルクジョブと並ぶジョブモデルの1つで、以下の特徴を持つジョブです。

- マスタ・ワーカ型ジョブは、ジョブスクリプトプロセス、マスタプロセスおよびワーカプロセスから構成されます。
  マスタプロセスとワーカプロセスが協調することで計算タスクを実行します (タスク：並列プログラムの処理単位)。
- マスタプロセスは、計算タスク全体を統括し、ワーカプロセスの生成や管理、計算結果を取りまとめます。
  ワーカプロセスは、マスタプロセスから依頼された計算タスクを実行し、結果をマスタプロセスに返します。
- マスタ・ワーカ型ジョブに割り当てられた計算ノードのダウンやプロセスの異常終了が発生しても、ジョブスクリプトプロセスが動作している限り、マスタ・ワーカ型ジョブは継続します。

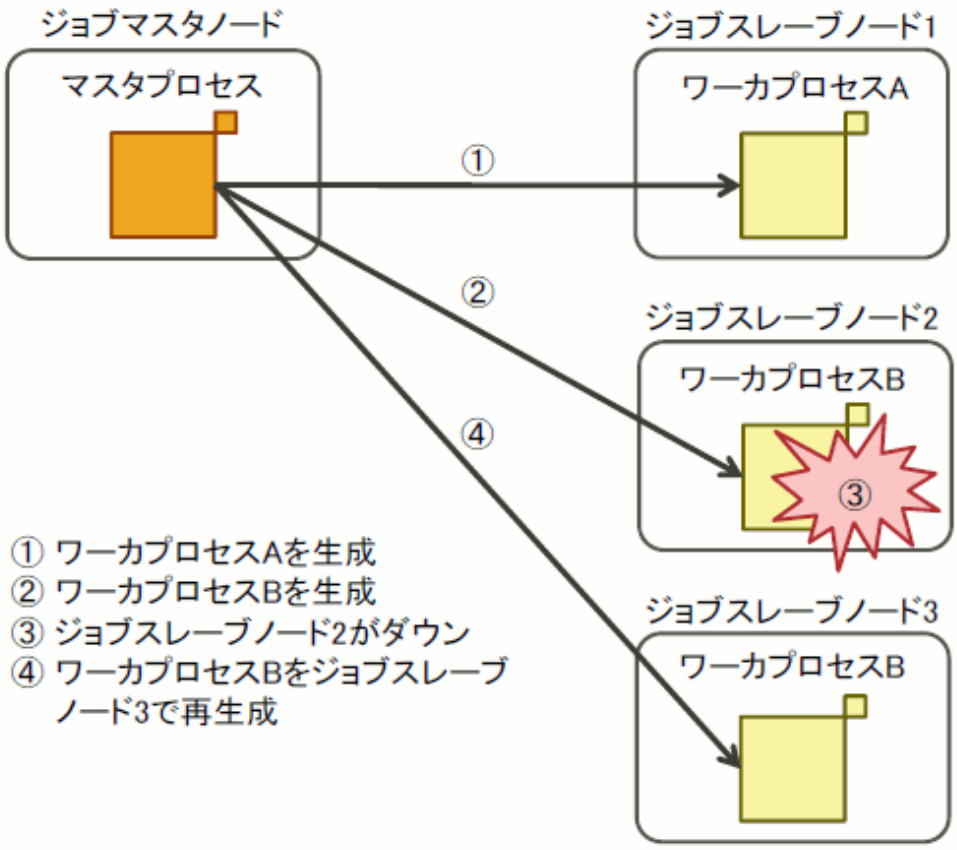
この特徴を利用して、計算ノードダウンやワーカプロセスの異常終了に対し、ワーカプロセスを別のノードで再実行する仕組みをユーザーが作ることで計算タスクを継続できます。

参照

マスタ・ワーカ型ジョブについては、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド マスタ・ワーカ型ジョブ編」を参照してください。

マスタ・ワーカ型ジョブで実行されるユーザープログラムの実行イメージを以下に示します。

図1.7 マスタ・ワーカ型ジョブの実行イメージ



参考

- ジョブスクリプトプロセスが動作するノードを「ジョブマスタノード」、それ以外のノードを「ジョブスレーブノード」と呼びます。マスタプロセスは、ジョブスレーブノードで動作するワーカプロセスを管理することが目的なので、ジョブマスタノードで動作させる必要があります。
- 複数のプロセスを生成するという点で、類似したジョブモデルに「バルクジョブ」があります。バルクジョブは、同一のジョブスクリプトを複数のサブジョブとして投入する方式で、それぞれのサブジョブは独立して動作します。バルクジョブでは、サブジョブの数はジョブ投入時に指定され、実行中に変化しません。
  一方、マスタ・ワーカ型ジョブでは、マスタプロセスと各ワーカプロセスが1つのジョブとしてプロセス間通信を行いながら動作します。また、ワーカプロセスはマスタプロセスによって動的に生成するため、ジョブの実行中に数が変化します。

1.2.2 バッチジョブと会話型ジョブ

ジョブは、その実行形態によってバッチジョブと会話型ジョブに分類されます。これを「ジョブタイプ」と呼びます。

表1.3 バッチジョブと会話型ジョブ

ジョブの種類	説明
バッチジョブ	非対話的に実行されるジョブです。 ジョブの標準出力、標準エラー出力はファイルになります。
会話型ジョブ	ジョブ内で実行されるプログラムに対し、ユーザーが対話的な操作ができるジョブです。 ジョブの標準入出力は、ジョブの実行依頼操作をしたシェルの標準入出力になります。主に、ジョブやプログラムのデバッグをする場合に利用します。

1.2.3 逐次ジョブと並列ジョブ

ジョブは、プログラムの並列度によって、逐次ジョブと並列ジョブに分類されます。

表1.4 逐次ジョブと並列ジョブ

ジョブの種類	説明
逐次ジョブ	逐次プロセス(単一スレッド、かつ、単一プロセス)を実行するジョブです。 例:単一スレッド、単一プロセスのプログラム
並列ジョブ	複数スレッドや複数プロセス、またはその両方を使って並列処理をするプログラムを実行するジョブです。 複数プロセスによる並列処理には、1つのノード内で行う場合と複数ノードにまたがって行う場合があります。 例:スレッド並列プログラム (自動並列化プログラム、OpenMPプログラム)、プロセス並列プログラム(MPIプログラム)、ハイブリッド並列プログラム なお、MPIプログラムを実行するジョブを特に「MPIジョブ」と呼ぶ場合もあります。

参考

プログラムの並列化にはスレッド並列とプロセス並列があり、ジョブ運用ソフトウェアでは以下のように分類しています。

表1.5 プログラムの並列化の種類

並列化の種類	説明
スレッド並列	1つのプロセスを複数のスレッドという単位に分け、複数のCPU(マルチコア CPU の場合は複数のコア)で処理を並行する方法です。 各スレッドは1つのプロセスのメモリ空間を共有して動作するため、プログラミングが比較的容易です。ただし、使える CPU 数の上限はノードに存在する個数になりますので、大きな性能向上は期待できません。 スレッド並列の例として、コンパイラによる自動並列化や OpenMP があります。これらについては、Development Studioが提供するマニュアルを参照してください。
プロセス並列	複数のプロセスで処理を並行する方法です。並列処理のために必要なデータはプロセス間通信によってやり取りします。この方法では、個々のプロセスを異なるノードで実行することもできるため、ノードの数を多くすることで性能向上が期待できます。ただし、高度なプログラミング技術が必要になります。 1ノードあたり、複数のプロセスがそれぞれ単一のスレッドで動作する場合を、フラット並列と呼びます。 プロセス並列の例として、MPI(Message Passing Interface)を利用した MPI プログラムがあります。MPI に関しては、Development Studioが提供するマニュアルを参照してください。
ハイブリッド並列	スレッド並列とプロセス並列の両方を使用する方法です。

1.2.4 シングルノードジョブとマルチノードジョブ

ジョブは、必要とするノード数によって、シングルノードジョブとマルチノードジョブに分類されます。

表1.6 シングルノードジョブとマルチノードジョブ

ジョブの種類	説明
シングルノードジョブ	実行するために1ノードだけ必要とするジョブです。 逐次ジョブはシングルノードジョブになります。 並列ジョブは、ノード内で複数のプロセスまたはスレッドを使うものが該当します。
マルチノードジョブ	実行するために複数ノード必要とするジョブです。 ノードをまたぐプロセス並列プログラムまたはハイブリッド並列プログラムが該当します。

参考

シングルノードジョブに対し、複数のノードを割り当てた場合、ジョブ運用ソフトウェアの中ではマルチノードジョブとして処理されます。

ジョブの並列度による分類と必要なノード数による分類を正確に表現する場合、本書では以下のように記述します。

表1.7 シングルノードジョブとマルチノードジョブ

必要なノード数	並列化手法	ジョブの名称
1ノード	逐次ジョブ	シングルノード(逐次)ジョブ
	スレッド並列	シングルノード(スレッド並列)ジョブ
	プロセス並列	シングルノード(プロセス並列)ジョブ
	ハイブリッド並列	シングルノード(ハイブリッド並列)ジョブ
複数ノード	プロセス並列	マルチノード(プロセス並列)ジョブ
	ハイブリッド並列	マルチノード(ハイブリッド並列)ジョブ

1.2.5 ノード資源の割り当て方による分類

ジョブは、割り当てられるノード資源の単位によって、ノード割り当てジョブと仮想ノード割り当てジョブに分類されます。

表1.8 ノード割り当てジョブと仮想ノード割り当てジョブ

ジョブの種類	説明
ノード割り当てジョブ	物理的なノードを単位として、ノード資源がジョブに割り当てられます。 この場合は、1つのノード内には1つのジョブだけ存在するため、「ノード専有ジョブ」とも呼びます。 ただし、資源の割り当て方や上限値の設定によっては、ジョブはノード内の資源をすべて利用できるわけではありません。
仮想ノード割り当てジョブ	ノード内のCPUコアとメモリをセットにした仮想ノードと呼ぶ単位で、ノード資源がジョブに割り当てられます。 仮想ノードは、割り当てたジョブ専用のノード資源になります。 ノード割り当てよりもノード資源を無駄なく利用ができるのが特徴です。 仮想ノードは、1つのノード内に複数存在できるため、1つのノードを複数のジョブで共有する場合は、「ノード共有ジョブ」と呼びます。 PRIMERGYサーバでジョブを実行する場合、仮想ノードの割り当て方によって、ノード専有ジョブになったり、ノード共有ジョブになったりします。 FXサーバでは仮想ノード割り当てジョブはサポートしていません。

参照

ノードと仮想ノードの詳細やノード資源の割り当て方については、"1.6 ノード資源の割り当て" を参照してください。

1.2.6 I/Oノードの割り当て方による分類 [FX]

ジョブの入出力を処理するI/Oノード、特にストレージI/Oノードの割り当て方によって、I/O専有ジョブとI/O共有ジョブに分類されます。

表1.9 I/O専有ジョブとI/O共有ジョブ

ジョブの種類	説明
I/O専有ジョブ	1つのジョブがその入出力処理のためにストレージI/Oノードを専有します。
I/O共有ジョブ	ジョブの入出力処理のために使うストレージI/Oノードをほかのジョブと共有します。

参照

I/O専有ジョブとI/O共有ジョブの詳細については、"2.3.2.11 I/Oノードを意識した資源の割り当て [FX]"の"[I/Oノードをジョブに専有させる割り当て]"を参照してください。

1.3 ジョブの状態

ジョブには以下のような状態があり、ジョブの実行依頼から終了まで、状態は変化します。

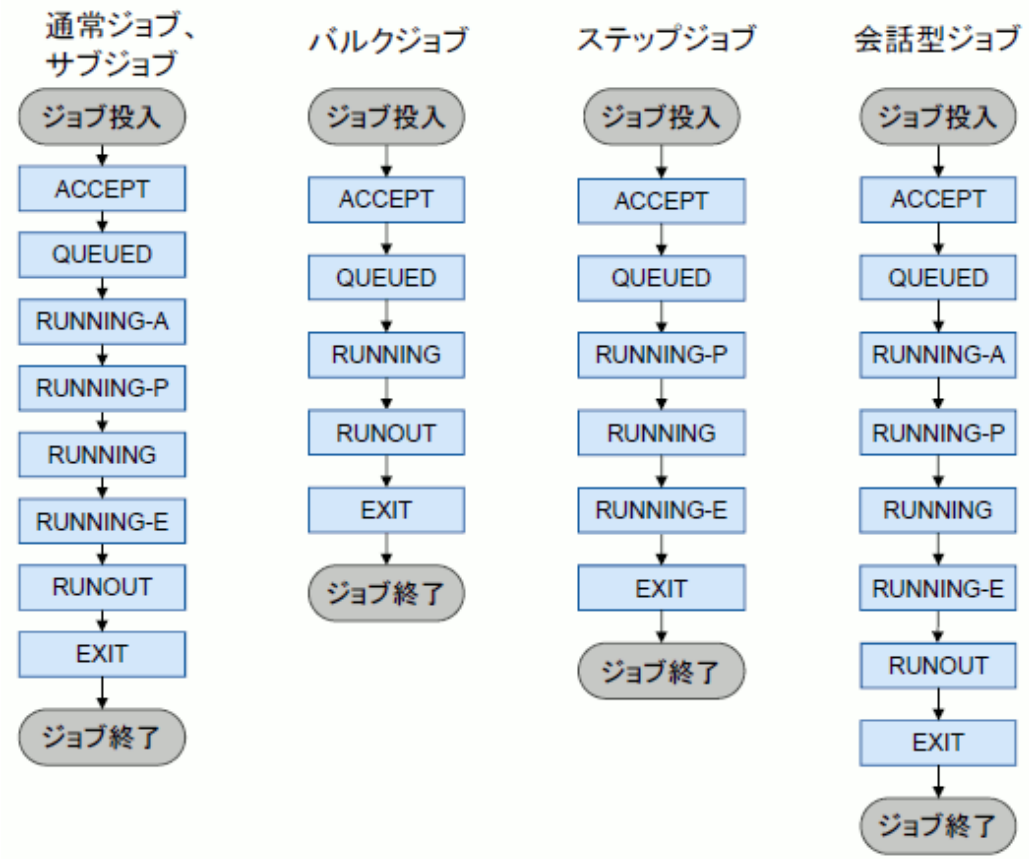
表1.10 ジョブの状態

状態名	表記(注1)	説明
ACCEPT	ACC	ジョブが受付け条件(ジョブACL機能による制限項目)を満たしているか確認している状態
QUEUED	QUE	ジョブが受け付けられ、実行の順番が来るのを待っている状態
RUNNING-A	RNA	ジョブの実行に必要な資源を獲得している状態
RUNNING-P	RNP	プロローグ処理が実行されている状態 (注2)
RUNNING	RUN	ジョブが実行されている状態
RUNNING-E	RNE	エピローグ処理が実行されている状態 (注2)
RUNOUT	RNO	ジョブの終了処理をしている状態
EXIT	EXT	ジョブが終了した状態
REJECT	RJT	ジョブの受付けが拒否された状態
CANCEL	CCL	ジョブ投入者または管理者からの指示で、ジョブが中止された状態
HOLD	HLD	ジョブの実行を中止し、投入済みの状態で固定された状態
ERROR	ERR	ジョブ運用管理機能で検出したエラーにより、投入済みの状態を保ったままジョブが中止された状態
SUSPEND	SPP	サスペンド処理をしている状態
SUSPENDED	SPD	サスペンド済みの状態
RESUME	RSM	リジューム処理をしている状態

(注1) pjsubコマンドやpjstatコマンドなどの表示における、状態を表す文字列です。
(注2) 「プロローグ」および「エピローグ」については "1.12 プロローグ・エピローグ機能" を参照してください。

ジョブの基本的な状態遷移は、ジョブの種類によって異なり、以下のようになります。

図1.8 ジョブの基本的な状態遷移



注意

- バルクジョブとステップジョブには、1つのジョブとしての状態とそのサブジョブの状態があります。「バルクジョブの状態」または「ステップジョブの状態」は、前者を指します。
  バルクジョブやステップジョブの状態遷移と、そのサブジョブの状態遷移は上の図に示すように異なります。
- ステップジョブの状態は、実行中のサブジョブ(実行中のサブジョブがない場合は次に実行される予定のサブジョブ)の状態と同じになります。
  ただし、サブジョブが状態RUNNING-Aの場合、ステップジョブは状態QUEUEDになります。また、サブジョブが状態RUNOUTの場合、ステップジョブは状態RUNNINGまたはRUNNING-Eになります。

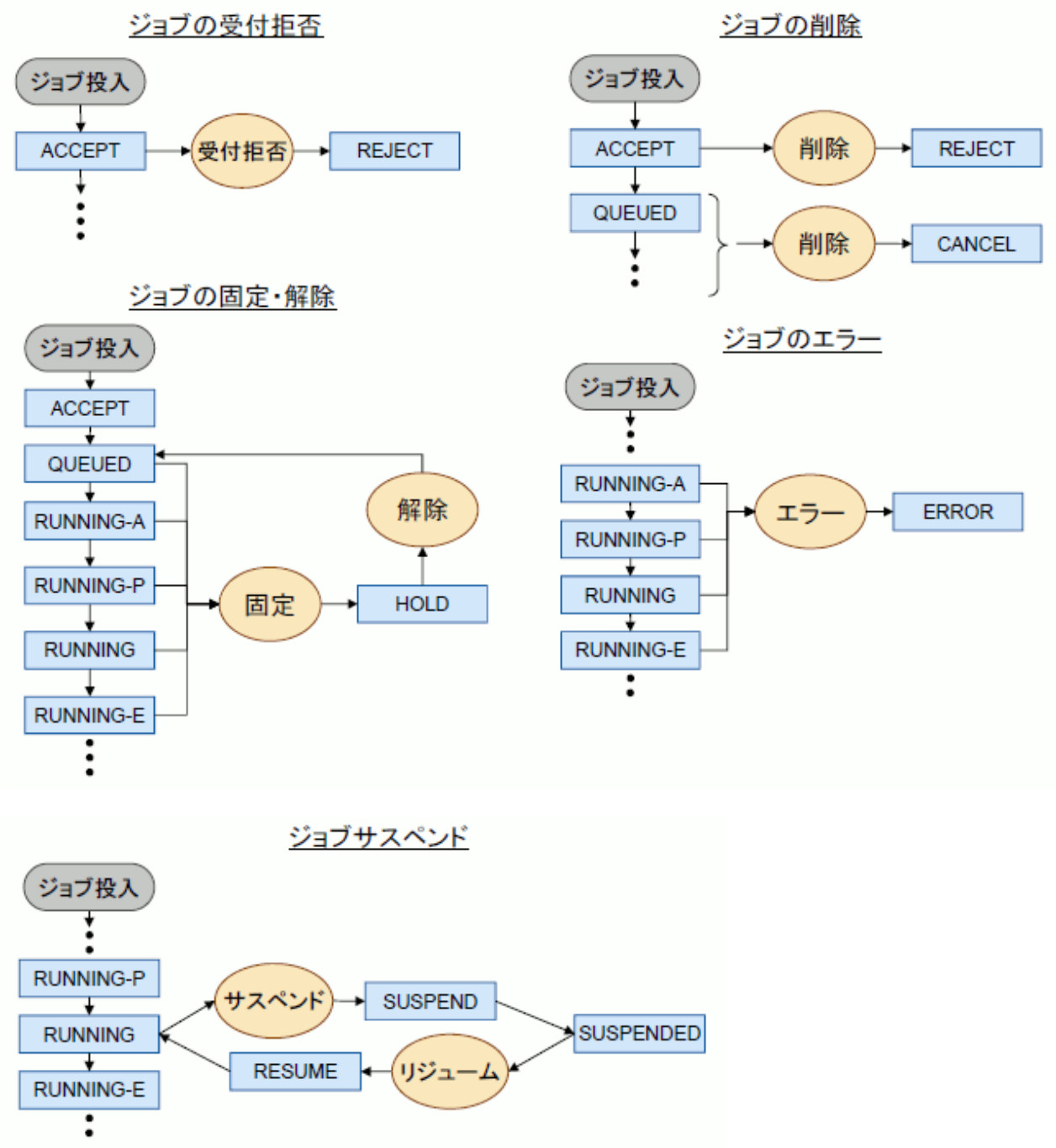
なお、ジョブを投入後に発生する事象に伴って、それぞれ以下のようにジョブは状態遷移をします。

表1.11 事象とジョブの状態遷移

事象	ジョブの状態遷移
ジョブの受付け拒否	ユーザーが投入したジョブは仮受付けの状態 ACCEPT になります。受付け条件(ジョブACL)を満たしていない場合、受付けが拒否された状態 REJECT になります。 ジョブACL については、"1.10 ジョブACL機能" を参照してください。
ジョブの削除	投入したジョブの取り消し操作を「ジョブの削除」と呼びます。ACCEPT 状態のジョブを削除すると、REJECT 状態になります。それ以外の状態のジョブを削除すると、CANCEL 状態になります。ジョブが実行中の場合は、中止されます。 ジョブの削除操作については "2.5.1 ジョブの削除" を参照してください。
ジョブの固定・固定解除	ジョブの実行を中止し、状態が変わらないようにする操作を「ジョブの固定」と呼びます。ジョブは QUEUED、RUNNING-A、RUNNING-P、RUNNING または RUNNING-E 状態で固定でき、その結果、HOLD 状態になります。HOLD 状態を解除すると、再び QUEUED 状態となります。 ジョブの固定・固定解除操作については、"2.5.3 ジョブの固定と固定解除" を参照してください。

事象	ジョブの状態遷移
ジョブのエラー	ジョブが RUNNING-P 状態から RUNNING-E 状態の間で、ジョブ運用管理機能にエラーが発生した場合は、ERROR 状態になります。
ジョブのサスペンド	実行中のジョブを一時的に停止することをサスペンドと呼びます。サスペンドしたジョブを再び実行させることをリジュームと呼びます。これは管理者が操作できます。 RUNNING 状態のジョブに対してサスペンドが発生すると、ジョブの状態は SUSPEND 状態になります。サスペンド処理が完了すると SUSPENDED 状態になります。 サスペンド済み (SUSPENDED) のジョブのリジューム処理が始まると、ジョブの状態は RESUME 状態になります。リジューム処理が完了すると、RUNNING 状態になり、ジョブの実行が継続されます。

図1.9 ジョブの受付け拒否やエラー発生などでの状態遷移



1.4 ジョブIDとサブジョブID

ジョブには、システム内で一意に識別する番号「ジョブID」が自動的に設定されます。ジョブIDは 0 から 2147483647 の値です。

ジョブのうち、バルクジョブとステップジョブには、サブジョブを一意に識別するための番号「サブジョブID」も設定されます。サブジョブIDは、ジョブIDにバルク番号またはステップ番号を付加した文字列で表現します。バルク番号は 0 から 999999 の範囲、ステップ番号は 0 から 65534 の範囲です。

バルクジョブのサブジョブIDは "*ジョブID*[*バルク番号*]"、ステップジョブのサブジョブIDは"*ジョブID*_*ステップ番号*"のように表記します。

```
[例]
ジョブID                    : 123456
バルクジョブのサブジョブID   : 123456[1234]     ジョブID 123456、バルク番号 1234 のサブジョブ
ステップジョブのサブジョブID : 123456_1234      ジョブID 123456、ステップ番号 1234 のサブジョブ
```

1.5 ジョブの出力

ジョブの標準出力および標準エラー出力は、ジョブ投入時のカレントディレクトリにそれぞれファイルとして出力されます。

これらのファイル名については、"2.3.2.10 バッチジョブの標準出力、標準エラー出力ファイルの指定" を参照してください。

1.6 ノード資源の割り当て

ジョブを実行するためには、それを実行するノード資源(CPUコア、メモリ)を割り当てる必要があります。

システム内のノードは、システムの運用単位であるクラスタ、その中のジョブ運用の単位であるリソースユニットで階層的にグルーピングされます。さらに、リソースユニットはリソースグループと呼ぶ資源割り当ての単位に分けられ、ジョブ運用の方針に応じて、複数のリソースグループが用意されます(マニュアル「ジョブ運用ソフトウェア 概説書」を参照)。

ジョブはリソースグループ内で実行されます。管理者がジョブACL機能によってデフォルトのリソースグループを設定していない場合は、ジョブ投入時にリソースグループを指定する必要があります。ジョブのプロセスは、割り当てられたCPUコアを専有します。また、Development Studioを利用することで、CPUコアへのバインドを制御できます。

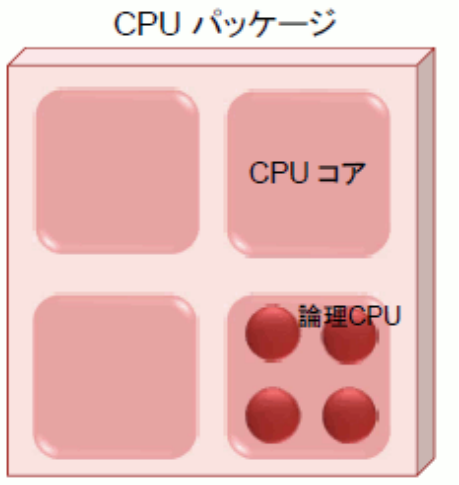
ユーザーはジョブを投入する際に、リソースグループ内のノード資源をどのように割り当てるかを指定します。FXサーバとPRIMERGYサーバとではアーキテクチャーが異なるため、ノード資源の割り当てに関する考え方も異なります。

以降では、FXサーバおよびPRIMERGYサーバにおけるノード資源の割り当ての考え方についてそれぞれ説明します。

1.6.1 CPU 資源の階層構造について

ジョブ運用ソフトウェアでは、CPU 資源を以下に示す階層構造として扱います。

図1.10 CPU 資源の階層構造

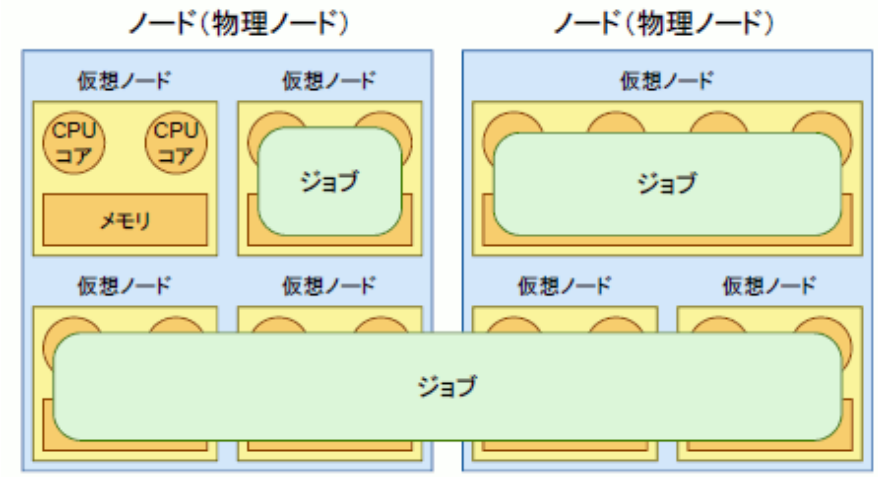


名称	説明
CPU パッケージ	物理的な CPU の単位を表します。
CPU コア	CPU パッケージ内に1つ以上存在します。CPU コアには OS によって CPU コア ID が設定されます。 ジョブ運用ソフトウェアがジョブに割り当てる CPU 資源は、CPU コアを基本単位とします。
論理 CPU	インテル®ハイパースレッディング・テクノロジーのような SMT (*Simultaneous Multithreading*) 機能が有効な場合に、CPU コア内に存在する擬似的な CPU です。
CPU	OS 上のプロセスが認識する CPU 資源の単位です。SMT 機能が有効な場合は論理 CPU、無効な場合は CPU コアに相当します。CPU には OS によって CPU ID が設定されます。

1.6.2 ノードと仮想ノード

ジョブ運用ソフトウェアが管理するノード資源の概念には、物理的なノードと、CPUコアとメモリをセットにした仮想ノードがあります。
仮想ノードは、物理的なノードからユーザーが必要に応じた資源量を切り出して定義するものです。このため、システムのノード資源を無駄なく利用ができ、システムのスループットや稼働率を高めることができます。

図1.11 仮想ノード



参考

- 1つの仮想ノードの資源量(コア数、メモリ量)は、物理的な1つのノード内に収まっている必要があります。複数の物理的なノードを連結して1つの仮想ノードとすることはできません。
- ジョブ運用ソフトウェアでは、単に「ノード」と表現する場合は物理的なノードを指します。

1.6.3 ノード単位での割り当て

ノード資源をノード単位で割り当てる場合、計算ノードの機種によって考え方が異なります。以下では、FXサーバおよびPRIMERGYサーバの割り当て方について説明します。

参照

ジョブ投入時のノードの指定方法については、"2.3.2.2 ノード資源の指定" を参照してください。

1.6.3.1 FXサーバのノード単位での割り当て

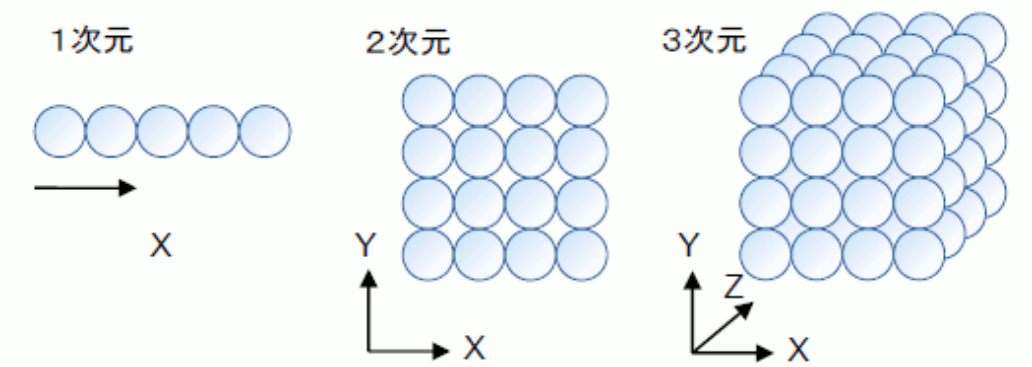
FXサーバをノード単位でジョブに割り当てる場合は、以下を指定できます。

- 割り当てるノードの形状とノード数

- Tofu 座標におけるノードの配置方法
- MPIプログラムを実行する場合、ノード割り当てのルール (ランク)

割り当てるノードは、仮想的な 1次元、2次元、または 3次元の空間に配置される形状として指定します。

図1.12 ノードの形状 (イメージ)



この形状のノードを、FXサーバの Tofu 座標にどのように配置するかによって、トーラスモード、メッシュモード、および離散割り当ての3種類の配置方法があります。

表1.12 Tofu座標におけるノードの配置方法

ノードの配置方法	説明
トーラスモード	ノードの最小割り当て単位は Tofu 単位 (12ノード) です。 割り当てられるノードは、Tofu 座標上で隣接するノードが選択されます。
メッシュモード	ノードの最小割り当て単位は1ノードです。 割り当てられるノードは、Tofu 座標上で隣接するノードが選択されます。
離散割り当て	ノードの最小割り当て単位は1ノードです。 割り当てられるノードは、できるだけTofu座標上で隣接するように選択されます。 以下の場合は隣接しないノードが選択されます。 ・隣接する空きノードがない場合 ・隣接しないノードを選択することでジョブの実行開始を早められる場合

これらの配置方法の特徴は以下のとおりです。

- ジョブの通信性能
  ジョブの通信性能の点では、トーラスモードが有利です。
  トーラスモードとメッシュモードはジョブごとの通信は干渉しません。しかし、メッシュモードはノードの配置結果によってはノード間の通信経路の長さが変わるため、通信性能が常に同じにならない可能性があります。
  離散割り当てでは、割り当てるノードが隣接するとは限らないため、各ジョブの通信が干渉しやすくなります。

- ノードの割り当てやすさ
  ノードの割り当てやすさの点では、ノードの最小割り当て単位が1ノードのメッシュモードや離散割り当てが有利です。
  すなわち、メッシュモードや離散割り当てのほうが、トーラスモードよりもジョブの実行開始が早くなる可能性があります。
  また、メッシュモードは1ノード単位の割り当てですが、割り当てるノードの形状に条件があります("2.2.1 リソースユニット、リソースグループの確認"の"表2.4 割り当て可能なノードのサイズ [FX]"を参照)。このため、ノードの形状に縛られない離散割り当てのほうがメッシュモードよりノードを割り当てやすくなります。

- 通信経路故障時の影響
  通信経路が故障した場合の影響の点では、トーラスモードやメッシュモードが有利です。
  離散割り当てでは、できるだけTofu座標上で隣接するようにノードを配置しようとしますが、必ずしも隣接するとは限りません。ジョブに割り当てられていないノードであっても、複数のジョブの通信経路としてTofuインターコネクトが使用される場合があります。このため、通信経路として使用されているノードのTofuインターコネクトが故障した場合、複数のジョブが影響を受けやすくなります。

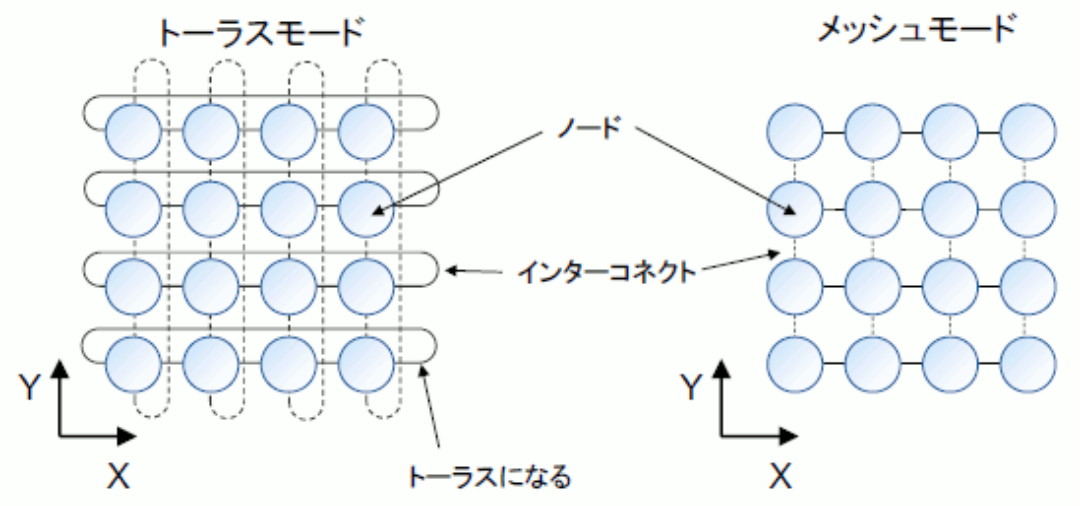
注意

- トーラスモードで、1次元形状のシングルノードジョブを投入した場合は、割り当てられるノードは1ノードになります。すなわち、1つのTofu単位内で、1次元形状の複数のシングルノードジョブを実行できます。
  管理者が行うリソースグループの設定によっては、割り当てるノード数が11ノード以下のマルチノードジョブも1つのTofu単位内で複数実行できます。
  トーラスモードで、シングルノードジョブを2次元形状(1x1)または3次元形状(1x1x1)として投入した場合は、割り当てられるノードはTofu単位に切り上げられます。
- メッシュモードで 2次元形状のノードを指定した場合は、割り当てられるノード形状は3x1単位に切り上げられます。
- リソースユニット内で、トーラスモードまたはメッシュモードのジョブと離散割り当てジョブを混在して実行した場合、離散割り当てジョブがトーラスモードまたはメッシュモードのジョブの通信性能を劣化させる可能性があります。
  なお、ほかのユーザーのジョブも含め、リソースユニット内で、これらのジョブが混在して実行される可能性があるかどうかは、管理者にお問い合わせください。

トーラスモードでは、ノードを接続するインターコネクトは、各軸方向のトーラスを構成します。トーラスとはノードを接続する形状(トポロジ)の1つで、ノード形状の1つの軸で、両端のノードが隣り合うノードとして接続され、ループを描くようにインターコネクトで接続される構造のことです。

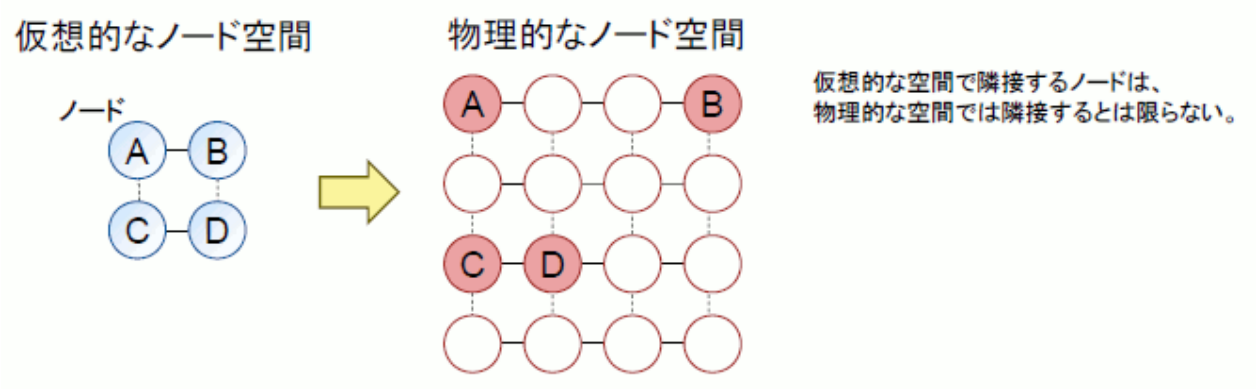
- 1次元形状では、X 軸方向のトーラス
- 2次元形状では、X、Y軸方向のトーラス
- 3次元形状では、X、Y、Z軸方向のトーラス

図1.13 ノード間の接続 (例:2次元形状)



トーラスモードとメッシュモードでは、割り当てられるノードは物理的な空間で隣接するように配置されますが、離散割り当てでは、ノードが物理的な空間で隣り合うことは保証されません。
つまり、離散割り当てではジョブ内の仮想的なノード空間におけるノードの距離と、物理的なノードの距離は一致しない場合があります。

図1.14 離散割り当てにおけるノードの配置のイメージ



トーラスモードとメッシュモードでは、指定できる最大のノード形状はノードの物理的な構成に関係し、リソースユニットやリソースグループと呼ぶジョブ運用の単位ごとに管理者によって設定されます。
この制限を超えるノード形状を指定した場合には、ノード資源不足となり、ジョブは実行されません。ユーザーはジョブ投入の前に、最大のノード形状について確認してから投入してください。確認方法については "2.2.1 リソースユニット、リソースグループの確認" を参照してください。

トーラスモードとメッシュモードで、指定するノード形状が2次元または3次元の場合、そのままでは最大形状に収まらなくても、回転させることで収まる場合は、自動的に調整されます。
例えば、最大のノード形状 XxYxZ = 2x3x32 のとき、ノード形状 6x3x2 は収まりませんが指定できます。これは、ノード形状を回転して 2x3x6 の形状にすることで最大形状 2x3x32 に収まるためです。
一方、ノード形状 4x4x4 はどのように回転させても最大形状 2x3x32 に収まらないので、ジョブはノード資源不足となり、実行されません。

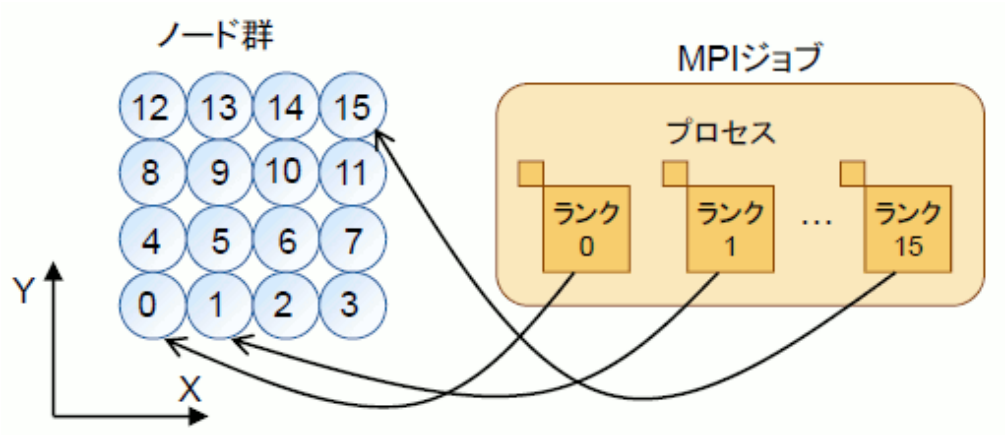
形状の自動的な回転は抑制できます。詳細は "2.3.2.2 ノード資源の指定" を参照してください。

ノード群の形状やノードの数の決定には、以下の考慮も必要です。

- ノード形状は、並列プログラムにおけるノード間通信のコストなどに影響します。
- 異なる複数のプログラム間で処理をする MPMD (Multiple Program Multiple Data) モデルの MPI プログラムでは、各プログラムが必要とするノードの数を考慮する必要があります。

MPI では、プロセスの管理のために「ランク」と呼ぶ番号をプロセスの生成順に付けます。ジョブ運用ソフトウェアでは、このランクに対し、どのノードを割り当てるかをジョブ投入時に指定できます。

図1.15 ランクとノード割り当ての例



割り当て方には、ルールを示す方法とランクとノードの対応をユーザーが1つ1つ指示する方法があります。詳細は "2.3.6 MPI ジョブの投入" を参照してください。
トーラスモードとメッシュモードのMPIジョブの場合、1次元形状よりも2次元形状または3次元形状を使って、ノードが隣接するようにランクを割り当てる方がランク間の通信距離が短くなり、ジョブの性能向上が期待できます。

注意

- トーラスモードとメッシュモードの場合、空き資源の状態によっては、2次元形状または3次元形状を指定したことで、1次元を指定した場合よりもジョブの実行開始が遅れる場合があります。
- TCP/IP通信を行うジョブは512ノード以下で実行してください。これ以上の規模では通信異常によるジョブの異常終了を引き起こすなど、システムに影響を与える可能性があります。
  なお、一般的にTCP/IP通信はMPIから用いられるRDMA通信より性能が劣るため、ジョブでの利用には向きません。

1.6.3.2 PRIMERGYサーバのノード単位での割り当て

ノード資源にPRIMERGYサーバをノード単位で割り当てる場合は、ノード数で指定します。

参考

- PRIMERGYサーバでは、FXサーバとは違い、割り当てるノードには形状の概念はありません。
- 管理者は、割り当てるノードに優先度を設定できます (割り当てノード優先度制御)。

1.6.4 仮想ノード単位での割り当て

ノード資源を仮想ノード単位で割り当てる場合は、以下を指定できます。

- 割り当てる仮想ノードの数
- 1つの仮想ノード当たりの CPU コア数とメモリ量
- CPU コアごとのメモリ量
- ノード資源割り当ての考え方 (ノード選択ポリシー) [PG]

注意

FXサーバでは仮想ノードの割り当てはサポートしていません。

PRIMERGYサーバでは、ノード資源の割り当てに関する考え方のノード選択ポリシーをジョブ投入時に指定できます。
ノード選択ポリシーには以下があります。

表1.13 ノード選択ポリシー

名称		説明
ノードと仮想ノードの関係	仮想ノード配置ポリシー	ノードに対する仮想ノードの配置の考え方
	ランクマップ	ノードに配置された仮想ノードに対する仮想ノードIDの設定の考え方
ジョブとノードの関係	ノード選択方式	ジョブが使うノードを選択する考え方
	割り当てノード優先度制御	ノードに設定した優先度に従ってノードを選択する考え方
ほかのジョブとの関係	実行モードポリシー	ノードの専有に関する考え方

以降で、これらの考え方について説明します。

仮想ノード配置ポリシー

仮想ノード配置ポリシーは、選択したノードに対し、仮想ノードをどのように配置するかを示します。

これには以下があります。

表1.14 仮想ノード配置ポリシーの種類

方式	説明
Absolutely PACK	すべての仮想ノードを1つのノードに配置します。配置ができない場合は、ジョブは実行されません。
PACK	仮想ノードをできるだけ少ない数のノードに配置します。
Absolutely UNPACK	1つのノードには1つの仮想ノードだけ配置します。仮想ノード数がノード数より多い場合は、ジョブは実行されません。
UNPACK	できるだけ1つのノードに1つの仮想ノードだけ配置するようにします。仮想ノード数がジョブに割り当てられたノード数より多い場合は、1つのノードに複数の仮想ノードが配置されます。

ランクマップ

ランクマップは、仮想ノード配置ポリシーに従って配置された仮想ノードに対し、仮想ノードIDを設定するルールを示します。これはMPIにおけるランクの指定に相当します。

ランクマップには以下の2つの方式があります。

表1.15 ランクマップによる割り当て方式の種類

方式	説明
rank-map-bynode	仮想ノードIDは、ノードのノードID順に1つずつ (ラウンドロビン方式) で設定されます。
rank-map-bychip	仮想ノードIDは、同じノード内の仮想ノードから順に設定されます。

ノード選択方式

ノード選択方式は、ジョブに割り当てるノードを、分散させるか、少数のノードに集中させるかを指定します。ただし、ノード選択方式は、ユーザーは指定できません。管理者がジョブACL機能で設定した内容が適用されます。

選択するノードを分散させる場合は、使われていないノード (CPUコアの空き数が多いノード) が優先して選択されます。少数のノードに集中させる場合は、すでに使われているノード (CPU コアの空き数が少ないノード) から選びます。

割り当てノード優先度制御

割り当てノード優先度制御は、管理者が各ノードに対して設定した割り当て優先度に従って、優先度が高いノードから選択します。

実行モードポリシー

ジョブによる、ノードの専有について指定する方式です。それぞれ以下のように呼びます。

表1.16 実行モードポリシーの種類

方式	説明
SIMPLEX	ノードを1つのジョブで専有します (ノード専有ジョブ)。 SIMPLEX が指定されたジョブが動作するノードは、CPU コアの空きがあっても、ほかのジョブには割り当てられません。
SHARE	ノードのほかの SHARE 指定のジョブと共有します (ノード共有ジョブ)。

参照

- 仮想ノードの指定方法については、"2.3 ジョブの投入" を参照してください。
- ノード選択ポリシーの指定方法については、"2.3.5 ノード選択ポリシーの指定 [PG]" を参照してください。

1.6.5 NUMA割り当てポリシー

NUMA アーキテクチャーの計算ノードは複数の CPUコアで共有するメモリへのアクセスコストが均一になりません。
この場合、ジョブに割り当てるCPUコアによっては実行性能のぶれや劣化の原因になります。そのため、メモリへのアクセスコストが同じ CPU コアのグループ (NUMA ノード) に対するジョブの割り当てルール (NUMA割り当てポリシー) を管理者が選択できます。

表1.17 NUMA割り当てポリシー

ポリシー	説明
pack	ジョブをできるだけNUMAノードに収まるように割り当てます。(デフォルト)
unpack	ジョブをNUMAノードに跨るように割り当てます。

エンドユーザはNUMA割り当てポリシーを選択できませんが、利用するシステムにどちらのポリシーが設定されているかは、ジョブACL機能("1.10 ジョブACL機能"参照)の設定内容で確認できます。

参考

NUMA割り当てポリシーが意味を持つのは、ジョブに仮想ノードを割り当てる場合です。

1.7 ジョブの実行順序

投入されたジョブの実行順序は、以下に示す様々な要素を使って評価した結果で決まります。これら実行順序を決めるための要素や基準は管理者が決めています。これを「ジョブ選択ポリシー」と呼びます。

表1.18 ジョブの実行順序を決める要素

要素	説明
特権的ジョブ	再実行ジョブは、優先的に実行されます。
投入時刻	ジョブの投入した時刻が早い順に実行します。これを fcfs (First-Come First-Serve)と呼びます。
要求ノード数 [FX]	ジョブが要求するノード数の大小で評価します。
グループやユーザーごとの優先度	グループやユーザーごとに設定されたジョブの優先度で評価します。
リソースグループの優先度	リソースユニット内のリソースグループの優先度に基づいて評価します。
同一ユーザーのジョブに対する優先度	同一ユーザーの複数のジョブにおける優先度で評価します。
リソースユニット内でのジョブの優先度	リソースユニット内のジョブの優先度に基づいて評価します。
ジョブ実行の実績による優先度	過去のジョブ実行の実績により、ユーザー間のシステム利用が公平になるように評価されます。これを「フェアシェア機能」と呼びます。
経過時間制限値	ジョブの経過時間制限値の大小で評価します。
経過時間制限値×要求ノード数 [FX]	ジョブの経過時間制限値と要求ノード数の積の大小で評価します。
実行開始時刻の指定	ジョブ投入時にジョブの実行開始時刻を指定しているかどうかで評価します。
会話型ジョブの指定	ジョブが会話型ジョブとして投入されたかどうかで評価します。

注意

上記の要素でジョブの実行順序が決まりますが、資源の空きがある場合は、その実行順序を飛び越して、先に実行できます。これを「バックフィル機能」と呼び、管理者が有効/無効を設定します。

以降では、ジョブの実行順序を決める主な要素について詳細を説明します。

1.7.1 グループやユーザー、リソースグループの優先度

OSにおけるグループやユーザーに対し、ジョブ実行の優先度が設定されている場合があります。この優先度は、ジョブACL機能("1.10 ジョブACL機能" 参照)により、管理者だけが設定できます。
グループやユーザーに対する優先度は、pjacl コマンドで確認できます("2.2.2 制限情報の確認"参照)。

また、ジョブ運用ソフトウェアのリソースグループに対してもジョブ実行の優先度が設定されている場合があります。
これも管理者だけが設定できます。リソースグループの優先度については、管理者にお問い合わせください。

1.7.2 ジョブ優先度

ユーザーは、自分のジョブについてだけ、ジョブ間の実行優先度を設定できます。優先度は最高が 255、最低が 0 の数値で表現します。

ジョブの優先度は、ジョブ投入時に pjsub コマンドの -p オプションで指定します。ジョブの投入後は、pjalter コマンドの -p オプションで変更できます。

参照

ジョブ投入時の優先度指定方法については "2.3.2.9 ジョブ優先度の指定" を参照してください。

ジョブの優先度には、ユーザーのジョブにおける優先度以外に、リソースユニット内のすべてのジョブにおける優先度もあります。リソースユニット内のジョブの優先度は、管理者だけが設定、変更できます。

1.7.3 フェアシェア機能

フェアシェア機能は、各ユーザーや各グループのシステム利用が公平になるように、実行したジョブの計算機資源利用実績によって実行優先度を決める機能です。
フェアシェア機能が有効になっている場合、ジョブを実行した直後は、優先度が下がり、時間の経過と共に一定の割合で回復します。
このため、大規模なジョブを実行した場合、または頻繁にジョブを実行した場合は、以降のジョブ実行がすぐに始まらない場合があります。
ご利用のシステムでフェアシェア機能が有効かどうかは、管理者にお問い合わせください。

1.7.4 ジョブの実行開始時刻

ユーザーはジョブの投入時に実行開始時刻を指定できます。ジョブの実行開始時刻は、ジョブ投入時に pjsub コマンドの --at オプションで指定します。ただし、実行開始時刻は、会話型ジョブに対しては指定できません。

注意

計算機資源の空き状況やジョブ選択ポリシーによっては、必ずしも指定した時刻にジョブの実行が開始されるとは限りません。指定時刻に開始できない場合は、それ以降で可能な限り早く計算機資源を確保できる時刻に実行されます。

参照

ジョブの実行開始時刻の指定方法については "2.3.2.8 実行開始時刻の指定" を参照してください。

1.8 カスタム資源

ジョブ運用ソフトウェアでは、ソフトウェアライセンスや電力などの任意の資源を定義できます。この任意の資源を「カスタム資源」と呼びます。
例えば、ソフトウェアライセンスをカスタム資源として定義し、ジョブ投入時に必要なソフトウェアライセンス数を要求すると、ジョブで使用するソフトウェアライセンス数が足りている時間にジョブが実行するようにスケジューリングされます。このように、カスタム資源を利用することで、計算ノードの CPU やメモリなどのハードウェア資源によるジョブのスケジューリングだけでなく、様々な利用用途に応じた汎用的な資源によるジョブのスケジューリングができます。

カスタム資源は管理者だけ定義できます。

ジョブ投入時に要求するカスタム資源は、pjsub コマンドの -L オプションで指定します。

なお、カスタム資源は、以下の 2つのタイプで定義されています。ジョブ投入時に要求するカスタム資源は、このタイプに従って指定します。

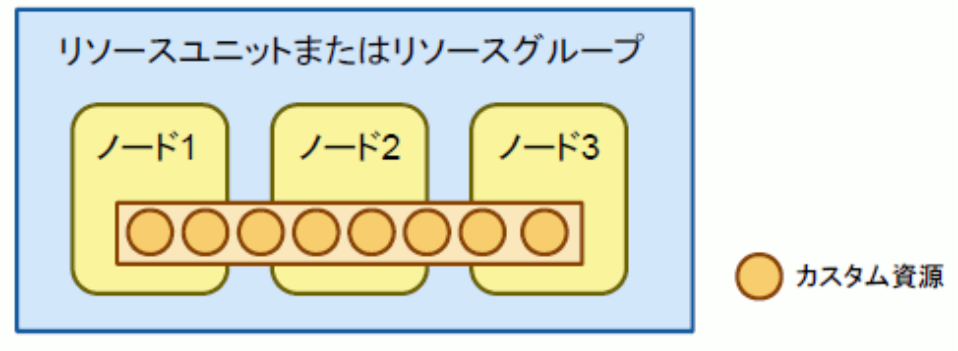
- 「数値」で定義されているカスタム資源
  ジョブ投入時には、ジョブで使用したい資源の数 (要求量) を指定します。
  例えば、数値で定義されているカスタム資源で、資源の数が8と定義されている場合、この8を超えない数を指定します。

・「種別」で定義されているカスタム資源
ジョブ投入時には、カスタム資源に定義されているいくつかの資源の種別うち、ジョブで使用したい種別(要求種別)を指定します。
例えば、種別で定義されているカスタム資源で、種別aとbが定義されている場合、このaとbの種別のうちのどちらかを指定します。

また、カスタム資源は、2つのタイプに加えて、以下の 2つの単位で定義されています。ジョブ投入時には、どちらの単位でカスタム資源が定義されているかも考慮して、資源の要求量や要求種別を指定する必要があります。

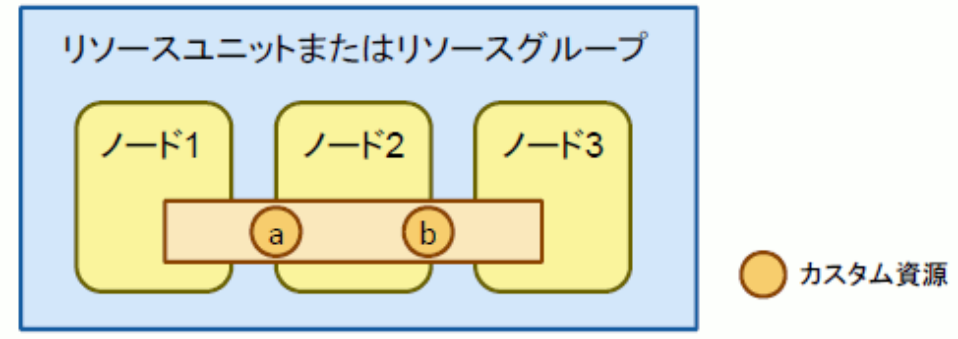
・リソースユニットまたはリソースグループ単位のカスタム資源
例えば、数値で定義されているカスタム資源で、カスタム資源の数が8と設定されている場合、リソースユニットまたはリソースグループ内の計算ノード全体で8つの資源を利用できます。

図1.16 リソースユニットまたはリソースグループ単位のカスタム資源 (数値で定義されているカスタム資源の場合)



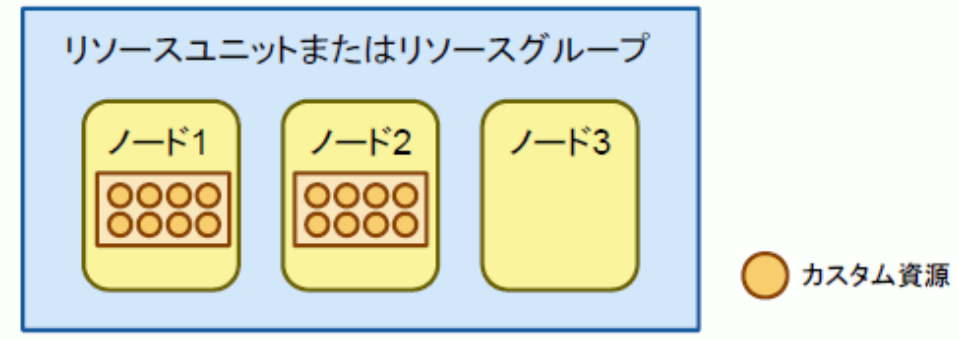
種別で定義されているカスタム資源で、資源の種別aとbが設定されている場合、リソースユニットまたはリソースグループ内の計算ノード全体で、この2つの種別をジョブの資源として利用できます。

図1.17 リソースユニットまたはリソースグループ単位のカスタム資源 (種別で定義されているカスタム資源の場合)



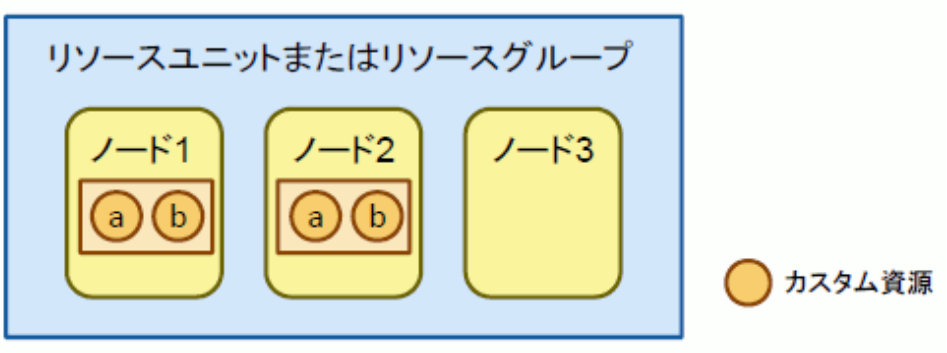
・ノード単位のカスタム資源
数値で定義されているカスタム資源で、リソースユニットまたはリソースグループ内の指定された計算ノード(ノード1とノード2)にカスタム資源の数が8と設定されている場合、指定されたノードそれぞれで 8つの資源を利用できます。

図1.18 ノード単位のカスタム資源 (数値で定義されているカスタム資源の場合)



資源の種別aとbが定義されているカスタム資源で、ノード1とノード2に種別aとbが設定されている場合、指定されたノードそれぞれでaとbの種別をジョブの資源として利用できます。

図1.19 ノード単位のカスタム資源 (種別で定義されているカスタム資源の場合)



参照

カスタム資源を指定したジョブの投入方法は、"2.3.2.4 カスタム資源の指定" を参照してください。設定されているカスタム資源の確認方法は、"2.2.2 制限情報の確認" を参照してください。

1.9 ジョブ実行環境

ジョブ運用ソフトウェアでは、ジョブプログラムを実行するソフトウェア環境 (ジョブ実行環境) をユーザーの指定に応じて明示的に切り替える「ジョブ実行環境カスタマイズ機能」を備えています。この機能により、ユーザーはシステムに配備されたジョブ実行環境の中から自身のジョブを実行するのに適した環境を選択し、ジョブに特化した環境を使用できます。さらに、ユーザー自身が用意した実行環境を使用することも可能です。

ジョブ実行環境カスタマイズ機能は、ジョブ実行環境として以下の環境 (実行モード) を利用できます。

通常モード

デフォルトのジョブ実行環境モードです。システムにインストールされた標準のOS環境(Linux)上でジョブプログラムを実行します。このモードではジョブ実行環境の明示指定は不要です。

Dockerモード

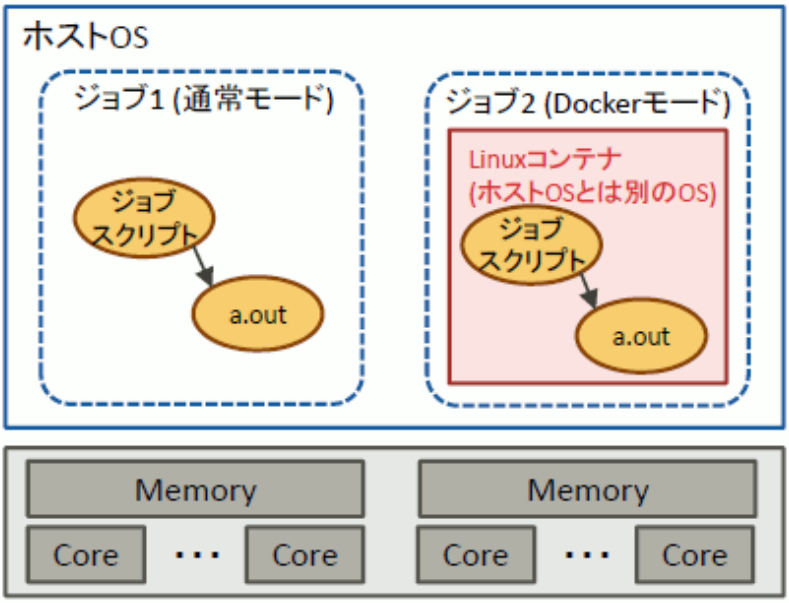
Dockerとは、Linuxコンテナを使用したOSレベル仮想化環境を提供するオープンソースソフトウェアです。ホストOS上にLinuxコンテナというホストOSから隔離された環境を用意し、ホストOSと異なるOS(コンテナイメージ)上でプログラムを動作させることができます。

参照

Docker の詳細は、https://www.docker.com/を参照してください。

Dockerモードは、FXサーバおよびPRIMERGYサーバで利用できます。

図1.20 Dockerモード



Dockerモードでは、ユーザーのジョブスクリプトを含むすべてのジョブプログラムをジョブ投入時に指定されたコンテナイメージから起動したコンテナ上で実行します。Dockerモードを利用することで、Dockerイメージとしてパッケージングされ たOSをそのまま利用してジョブを実行することができ、アプリケーション実行に必要なソフトウェア環境の導入が容易となります。

Dockerモードでは、管理者によってシステムに配備されたジョブ実行環境を利用するだけでなく、ユーザー自身が用意したジョブ実行環境も利用できます。これら2つの利用方式を、以下のように呼びます。

- SDI (System Deployed Image) 指定

  システムに配備されたジョブ実行環境を選択して利用する方式です。

- UDI (User Defined Image) 指定

  ユーザーが用意したジョブ実行環境を利用する方式で、指定するコンテナイメージを自身で切り替えてジョブを投入できます。

注意

UDI指定が可能かどうかはシステムの設定に依存します。UDI指定を使用する場合は、指定方法についてシステム管理者にお問い合わせください。

McKernel モード

McKernel は、Linux と協調して動作する軽量 OS であるLight Weight Kernel(LWK)の1つで、オープンソースソフトウェアとして公開されています。McKernel モードでは、ユーザーのジョブプログラムをMcKernel 上で実行します。McKernel 上ではジョブプログラムだけが動作し、ホストOSから隔離された環境でプログラムを実行するため、特にシステムノイズに影響を受けやすいアプリケーションに対して性能向上効果が期待できます。

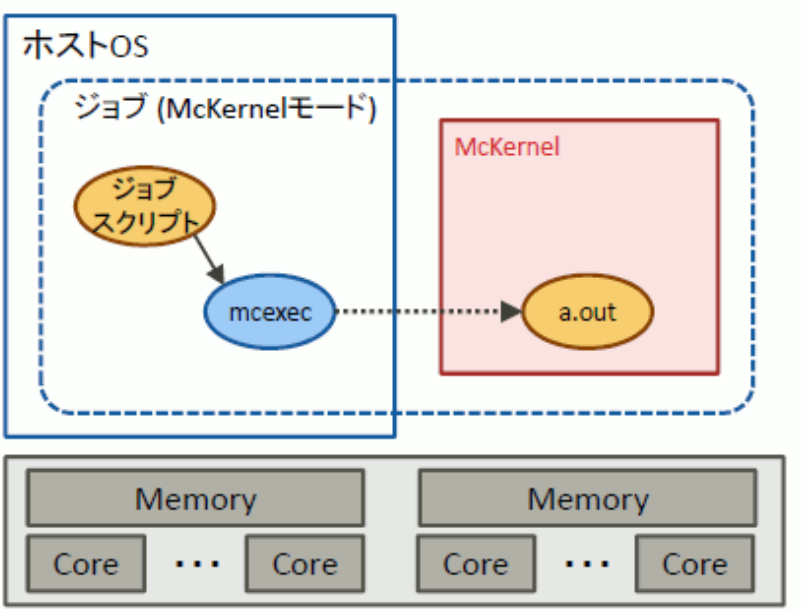
参照

McKernelの詳細は、https://ihkmckernel.readthedocs.io を参照してください。

McKernelモードは、FXサーバだけで利用できます。

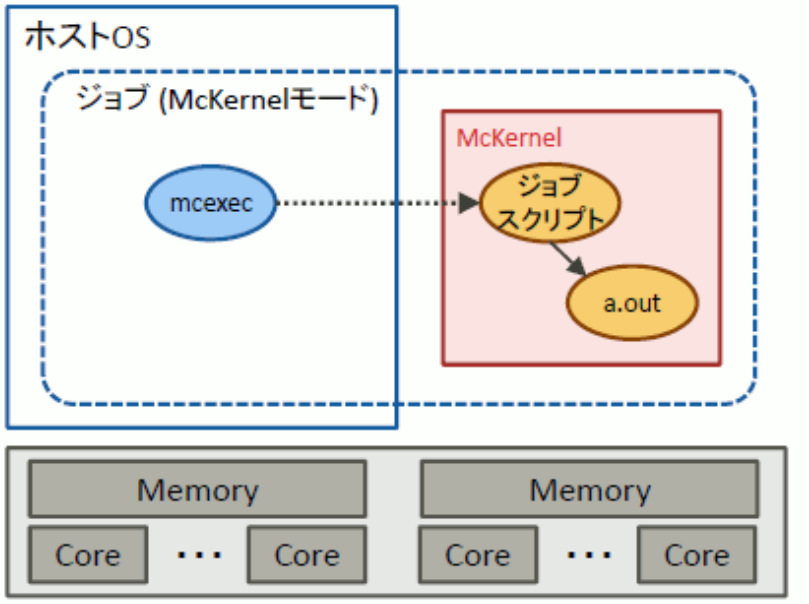
以下にMcKernelモードの概要を示します。McKernelは、システムコールなど重いOS処理をホストOSに移譲することでOSの軽量化を図り、プログラムの性能向上を実現しています。この移譲処理を受け持つプロセスをmcexecと呼び、ホストOS上に起動されます。

図1.21 McKernelモード(1)



McKernelモードでは、デフォルトではユーザーのジョブプログラムをMcKernel上で実行し、ジョブスクリプトはホストOS上に起動しますが、必要な場合はジョブ投入時の指定によりジョブスクリプトを含めすべてのユーザープロセスをMcKernel上に起動することも可能です。指定方法については "2.3.8 ジョブの実行環境の指定" を参照してください。

図1.22 McKernelモード(2)



McKernel モードでは、ジョブに対して次のように資源を割り当てます。

- CPU
  - McKernel モードでは、計算ノード内のすべてのCPUコアを使用します。
  - ホストOS上で起動されるプログラムおよびmcexecコマンドはアシスタントコア上で動作します。
- メモリ

  ジョブ投入時に指定したメモリ量をMcKernel用とホストOS上で動作するプログラム向けに分割して割り当てます。
  - ジョブ投入時のオプションで指定したメモリ量のうち、128MiBはホストOS上で起動されるプログラム向けに割り当てられます。それ以外のメモリは、すべてMcKernelに割り当てられます。

- ホストOS上で起動されるプログラム用に割り当てるメモリ量は、環境変数PJM_JOBENV_MCKERNEL_JOBMEMで変更できます。設定方法については"2.3.8 ジョブの実行環境の指定"を参照してください。環境変数PJM_JOBENV_MCKERNEL_JOBMEMを設定しない場合は、管理者が設定したデフォルト値が適用されます。

McKernelモードでは、UDI指定とSDI指定が利用できます。UDI指定とSDI指定については、Dockerモードの説明を参照してください。

KVMモード

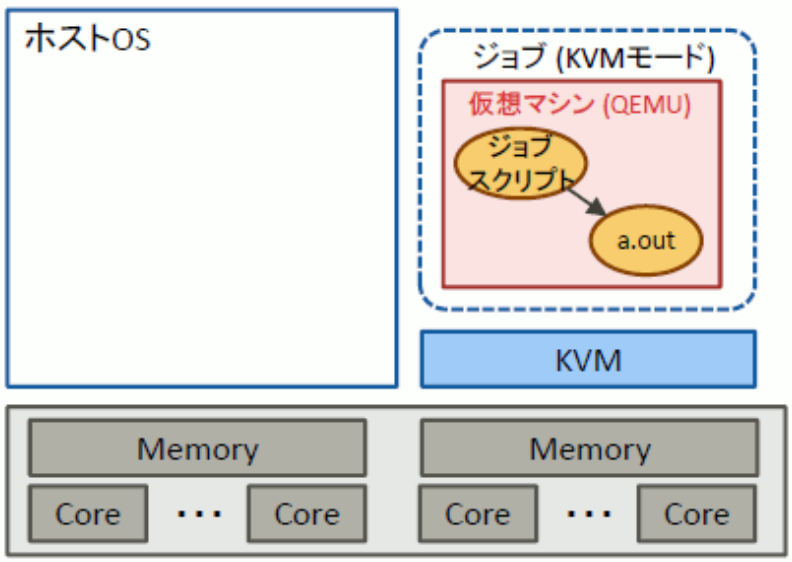
KVM (Kernel Virtual Machine) は、Linux 上の準仮想化機能です。KVM モードでは、ユーザーのジョブプログラムをこの KVM 上で実行します。KVM モードでは、仮想マシン管理のためのオープンソースソフトウェアlibvirt、QEMUを使用してKVMを利用します。ハードウェアの仮想化支援を利用することで高速なハードウェア仮想化環境上でのジョブ実行が可能となります。カーネルモジュールなどOSカーネル層のシステム開発者などプログラム実行に特権を必要とするユーザーに対して有効です。

参照

KVM の詳細は、https://www.linux-kvm.org/page/Main_Pageを参照してください。

KVMモードは、FXサーバおよびPRIMERGYサーバで利用できます。

図1.23 KVMモード



KVMモードでは、UDI指定とSDI指定が利用できます。UDI指定とSDI指定については、Dockerモードの説明を参照してください。

注意

KVMモードは、ユーザーのホームディレクトリがログインノードと計算ノードで共有されているシステムで利用できます。また、UDI指定が可能かどうかはシステムの設定に依存します。UDI指定を使用する場合は、指定方法についてシステム管理者にお問い合わせください。

参照

ジョブ実行環境を指定したジョブの投入方法については、"2.3.8 ジョブの実行環境の指定" を参照してください。

ジョブ実行環境を自分で用意する場合は、Linuxコンテナなどの仮想化技術の知識が必要になります。本書では、ジョブ実行環境のイメージファイルの作成方法についてだけ、"E.1 ジョブ実行環境のイメージファイルの作成"で説明しています。

1.10 ジョブACL機能

ジョブが使用するメモリ量やCPU時間に上限を設けたり、ユーザーが利用できる機能をジョブ運用の方針に合わせて限定したりするために、ジョブ運用ソフトウェアは「ジョブ ACL (Access Control List)機能」を備えています。

ジョブ ACL 機能は、クラスタやリソースユニット、リソースグループごと、またはグループ、ユーザーごとに設定ができ、ジョブ運用に合わせて管理者が設定します。ユーザーはジョブACL機能による制限の範囲内で、システムを利用できます。例えば、ジョブ ACL 機能による制限を超える資源や機能を必要とするジョブ投入は拒否され、ジョブ実行時間が上限に達した場合はジョブが強制的に終了させられます。

ジョブ ACL 機能による制限内容の詳細は pjacl コマンドで確認できます。

参照

pjacl コマンドでジョブACL機能の制限内容を確認する方法、および制限される項目については、"2.2.2 制限情報の確認" を参照してください。

1.11 ジョブ統計情報

ジョブが使用した資源量や各種制限値などの情報を「ジョブ統計情報」と呼びます。ジョブ運用ソフトウェアは、このジョブ統計情報を出力する機能を備えています。ジョブ統計情報を参照することで、ジョブの実行の様子を後から分析できます。

ユーザーは、ジョブ投入時に pjsub コマンドで、ジョブの出力結果としてジョブ統計情報を得られます。また、pjstat コマンドで実行中のジョブの統計情報を確認できます。

参照

pjstat コマンドでジョブ統計情報を確認する方法は "2.4.2 ジョブ統計情報の表示"、pjsub コマンドでジョブの実行結果としてジョブ統計情報を出力する方法は "2.3.2.6 ジョブ統計情報出力の指定" を参照してください。

ジョブ統計情報に含まれる項目と意味については"付録A ジョブの情報"やmanマニュアルpjstatsinfo(7)を参照してください。

1.12 プロローグ・エピローグ機能

ジョブ運用の方針に応じて、各ジョブの実行開始直前および実行終了直後に特定のスクリプトを実行するように管理者は設定できます。例えば、すべてのジョブに共通の処理として、特定の環境変数の設定や、作業ディレクトリをジョブ実行前に用意し、ジョブ終了時に削除するような処理です。

これを「プロローグ・エピローグ機能」と呼びます。

プロローグ・エピローグ機能の処理は、ユーザーは変更できませんが、ジョブの一部として実行されるため、以下に注意してください。

- プロローグ・エピローグ処理の標準出力、標準エラー出力はジョブの実行結果として出力されます。
- ジョブに対する資源制限値は、プロローグ・エピローグ処理にも適用されます。
- ジョブ統計情報は、プロローグ・エピローグ処理が加算された内容となります。
  ただし、ジョブの実行経過時間にプロローグ・エピローグ処理が含まれるかどうかはシステムの設定によります。

ご利用のシステムにおけるプロローグ・エピローグ処理については、管理者にお問い合わせください。

1.13 階層化ストレージ

ジョブ運用ソフトウェアは、「階層化ストレージ」でのジョブ実行をサポートしています。階層化ストレージとは、以下に示す第1階層ストレージと第2階層ストレージによる階層構造を持つファイルシステムです。

注意

現在、階層化ストレージは、計算ノードがFXサーバのシステムで利用できます。

- 第1階層ストレージ

  第1階層ストレージはLightweight Layered IO-Accelerator(LLIO)技術を利用した高速ファイルシステムです。第1階層ストレージはLLIOと呼ぶ場合もあります。階層化ストレージでは、計算ノードから直接アクセスできるのはこの第1階層ストレージになります。

  第1階層ストレージには以下の3種類の領域があります。

  ー ノード内テンポラリ領域

    ジョブに割り当てられたそれぞれの計算ノードで利用できるローカルな領域です。

  ー 共有テンポラリ領域

    ジョブに割り当てられたノード間で共有できる領域です。同じジョブのプロセスであれば、どの計算ノードからでもアクセスできます。

  ー 第2階層ストレージのキャッシュ

    ジョブからは第2階層ストレージとして見えますが、内部的には、第2階層ストレージを直接アクセスするのではなく、第1階層ストレージ上にある第2階層ストレージのキャッシュをアクセスします。

  ノード内テンポラリ領域と共有テンポラリ領域は"表1.19 ノード内テンポラリ領域と共有テンポラリ領域が確保される実行単位"に示す実行単位ごとに確保され、その実行単位でだけ使用できる高速な作業用ファイルシステムです。これらの領域は実行単位の開始時に確保され、終了時に削除されます。

表1.19 ノード内テンポラリ領域と共有テンポラリ領域が確保される実行単位

ジョブの種類		実行単位
バッチジョブ	通常ジョブ	ジョブごと
	ステップジョブ	サブジョブごと
	バルクジョブ	サブジョブごと
	マスタ・ワーカ型ジョブ	ジョブごと
会話型ジョブ		ジョブごと

参照

第1階層ストレージについての詳細は、マニュアル「LLIOユーザーズガイド」を参照してください。

- 第2階層ストレージ

  第2階層ストレージは、分散ファイルシステムの FEFS を利用し、ログインノードおよび各計算ノードで共有されます。
  ユーザーはジョブを投入する前に、ジョブの実行に必要なジョブスクリプトなどのファイルをこのファイルシステムに配置します。
  計算ノード上のジョブから第2階層ストレージへのアクセスは、内部的には第1階層ストレージ上のキャッシュに対するアクセスになります。

参照

FEFSについての詳細は、マニュアル「FEFSユーザーズガイド」を参照してください。

1.14 コマンドAPI

ジョブ運用で求められるユーザーインターフェースは運用システムごとに様々です。利用者が望むユーザーインターフェースを持ったコマンドの作成を支援するために、ジョブ運用管理機能はそれが提供するコマンドと同等の機能(ジョブの操作や情報取得)をプログラムから呼び出すためのインターフェースを提供します。これをコマンドAPI(Application Programming Interface)と呼びます。

コマンドAPIが提供する機能は以下のとおりです。

表1.20 コマンドAPIが提供する機能

対象利用者	分類	APIの種類
エンドユーザ および管理者	ジョブ操作API	ジョブの投入 (pjsub コマンド相当)
		ジョブの削除 (pjdel コマンド相当)

対象利用者	分類	APIの種類
		ジョブの固定 (pjhold コマンド相当)
		ジョブの固定解除 (pjrls コマンド相当)
		ジョブへのシグナル送信 (pjsig コマンド相当)
		ジョブの終了待ち合わせ (pjwait コマンド相当)
		ジョブのパラメーター変更 (pjalter コマンド相当)
	情報取得API	ジョブ情報の取得 (pjstat コマンド、pjstat コマンドの -s/-S オプション相当)
		資源情報の取得 (pjstat コマンドの --rsc オプション相当)
		制限値情報の取得 (pjstat コマンドの --limit オプション相当)
		ジョブACL情報の取得 (pjacl コマンド相当)
		資源状態の取得 (pjshowrsc コマンド相当)
管理者	システム操作API	ジョブ投入・実行可否の変更 (pmpjmopt コマンドの --set-rsc-ug および --show-rsc-ug オプション相当)
	ジョブ操作API	ジョブのパラメーター変更 (pmalter コマンド相当)

コマンドAPIの詳細については、マニュアル「ジョブ運用ソフトウェア APIユーザーズガイド コマンドAPI編」を参照してください。

第2章 ジョブの操作方法

ここでは、ジョブの作成から実行結果確認までの操作について説明します。

ユーザーは、以下のような流れでジョブの作成、投入し、実行結果を確認します。

1. ジョブの作成
2. ジョブの制御情報の確認
3. ジョブの投入
4. ジョブの情報の表示
5. ジョブの削除 (ジョブを中断する場合)
6. ジョブの結果の確認

注意

特に断りがない限り、以降の操作例は、ログインノードで実行するものとします。
管理者が実行する場合、ログインノード以外でも実行できるコマンドもあります。詳細は各コマンドのmanマニュアルを参照してください。

2.1 ジョブの作成方法

ここではジョブの作成と置き場所に関して説明します。

2.1.1 ジョブの作成方法

ジョブスクリプトの実体はシェルスクリプトです。

以下にジョブスクリプトの記述例を示します。

```
#!/bin/bash                    ← bash で実行されるジョブスクリプト
#PJM -L "node=2"               ← #PJM 以降の空白で区切られたフィールドは pjsub コマンドの引数と同じ扱い
# comment                      ← #PJM で始まらない場合は、単なるシェルのコメントと判断
# comment
export PATH=<dirname>:$PATH    ← 環境変数の設定など
...
#PJM -L "elapse=86400"         ← 一度コメント行以外の行が現れると、単なるコメント行と判断

./a.out                        ← プログラム a.out を実行
```

ジョブスクリプトを記述する際、以下のことに注意してください。

- ジョブスクリプトを実行するシェルは、ジョブスクリプトの1 行目に"#!"でシェルが指定されていなければ、ユーザーのログインシェルになります。
- ジョブスクリプト内で "#PJM " で始まる行には、ジョブを投入するときの pjsubコマンドの引数を記述できます。ジョブスクリプト内での指定より、pjsub コマンドの引数による指定の方が優先されます。
- 一度、コメント行以外が現れると、以降の "#PJM " は単にコメント行として無視します。
- ジョブスクリプトにはジョブを投入するユーザーに対する読込み権が必要です。実行権はなくてもかまいません。
- ジョブスクリプト内で、/dev/stdout または /dev/stderr にリダイレクトを行わないでください。リダイレクトした場合、標準出力ファイルまたは標準エラー出力ファイルが先頭から上書きされます。

以降では、ジョブを作成する際に、ユーザーが考慮すべき点について説明します。

参考

"2.3 ジョブの投入" で説明する、ジョブ投入をする pjsub コマンドのオプションは、ジョブスクリプト内にシェルのコメント行として記述できます。このため、以降では、pjsub コマンドのオプションも説明しているところがあります。

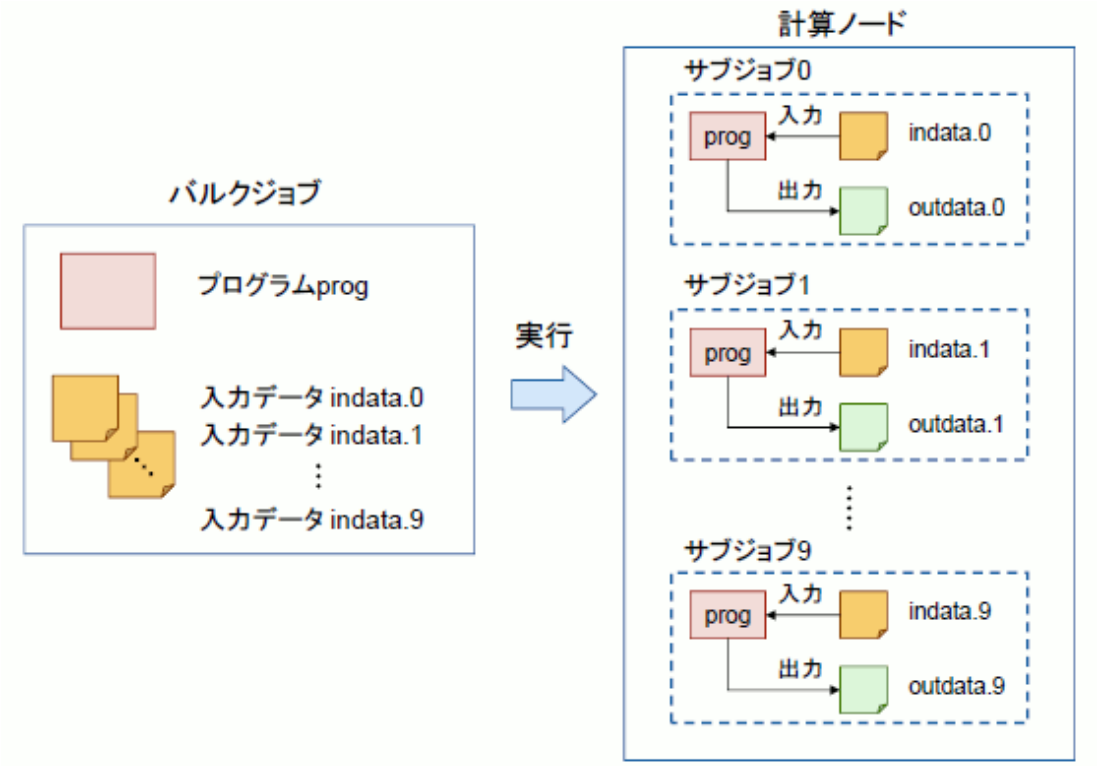
2.1.1.1 バルクジョブの作成

バルクジョブでは、サブジョブごとにジョブの入力パラメーターを変えられるようにジョブスクリプトを作成する必要があります。

このために、サブジョブごとに設定されるバルク番号を使います。バルク番号はサブジョブ内で、環境変数 PJM_BULKNUM に設定されています。

以下は、バルクジョブの例です。

図2.1 バルクジョブの例



上記をジョブスクリプトで記述した例が以下です。

```
#!/bin/sh
IN_DATA=./indata.${PJM_BULKNUM}        ← バルク番号を使って入力データファイル名を決定
OUT_DATA=./outdata.${PJM_BULKNUM}      ← バルク番号を使って出力データファイル名を決定

prog -i ${IN_DATA} -o ${OUT_DATA}      ← プログラム prog の引数で入出力データファイルを指定し、実行

※ ここでは、説明を簡単にするために、ジョブ投入時のオプション(例: 資源量の指定)は省略しています。
```

2.1.1.2 ステップジョブの作成

ステップジョブでは、先に実行したサブジョブの終了コードに基づいて次のサブジョブの動作を変えます。このため、必要に応じて複数のサブジョブの終了コードを設けるようにします。

ステップジョブの投入時は、サブジョブの終了コードに応じた次のサブジョブの実行条件を指定します。

指定方法の詳細は "2.3.4.2 ステップジョブの投入方法" を参照してください。

2.1.1.3 ワークフロージョブの作成

ワークフロージョブでは、ジョブの投入をユーザーがシェルスクリプトで制御します。

例えば、特定のジョブの終了を待ってから別のジョブを投入したり、あるジョブの結果によって次に投入するジョブを選択したりということをシェルスクリプトで自動的に行う方法です。

このために、ユーザーはジョブ運用ソフトウェアが提供する pjwait コマンドを利用して、ジョブの投入制御ができます。

- ジョブの終了待ち合わせ

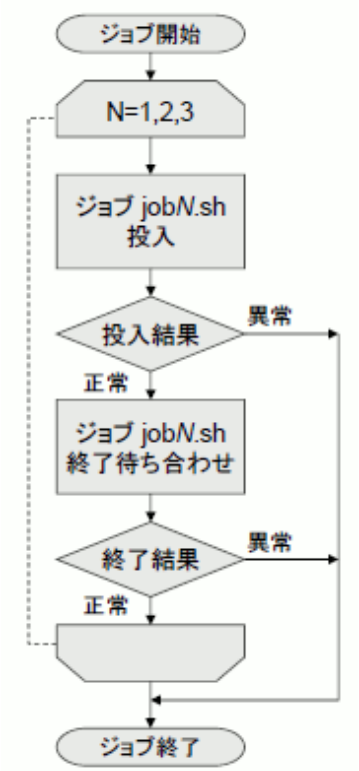
  pjwait コマンドを使用して、1つ以上のジョブの終了をシェルスクリプト内で待ち合わせることができます。

- ジョブの実行結果の取得

  pjwaitコマンドを使用して、ジョブ終了コード(ジョブマネージャーの処理結果)、ジョブスクリプトの終了ステータス、ジョブスクリプト終了時のシグナル番号を得ることができます。

以下は、ワークフロージョブの例です。

図2.2 ワークフロージョブの例



このワークフロージョブをシェルスクリプトで記述すると以下のようになります。

```
#!/bin/sh

for no in 1 2 3                    ← ジョブスクリプト job1.sh、job2.sh、job3.sh を順番に投入
do
    JID=`pjsub -z jid job${no}.sh` ← -z オプションでジョブID を表示させ、シェル変数 JID に代入
    if [ $? -ne 0 ]; then          ← ジョブ投入結果が 0 以外の場合、ワークフロージョブ終了
        exit 1
    fi
    set -- `pjwait $JID`                ← ジョブの終了を待ち合わせ、出力結果を位置パラメーターに代入 (注)
    if [ $2 != "0" -o $3 != "0" ]; then ← ジョブ終了コード($2)とジョブスクリプトの終了ステータス($3)の
        exit 1                            どちらかが 0 でない場合は、ワークフロージョブを終了
    fi
done                               ← 次のジョブの実行へ
```

(注) ジョブの種類、ジョブIDの指定の仕方によっては、出力結果が複数行になる場合があるため、工夫が必要です。出力についてはpjwaitコマンドの man マニュアルを参照してください。

2.1.1.4 マスタ・ワーカ型ジョブの作成

マスタ・ワーカ型ジョブの作成については、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド マスタ・ワーカ型ジョブ編」を参照してください。

ジョブ運用ソフトウェアは、マスタ・ワーカ型ジョブの実装方式として、ワーカプロセスの生成方法の観点で、以下の3つの方式をサポートします。

- MPI の動的プロセス生成によるワーカプロセス生成
  この方式は、マスタプロセス、ワーカプロセスが共に MPI プログラムの場合に利用します。すなわち、ワーカプロセスの生成や通信は MPI の仕組みを利用します。
  ワーカプロセスを生成するノードの選択はジョブ運用ソフトウェアが行います。

- Agent プロセスによるワーカプロセス生成
  この方式は、ワーカプロセスが MPI プログラムでない場合に利用します。
  プロセス間の通信はソケットなどの通信機構を利用し、ワーカプロセスの生成やそれを生成するノードの選択はユーザーが制御・管理します。

- pjaexe コマンドによるワーカプロセス生成
  前項と同様に、この方式もワーカプロセスが MPI プログラムでない場合に利用します。
  この方式でもプロセス間の通信はソケットなどの通信機構を利用し、ワーカプロセスを生成するノードの選択もユーザーが制御・管理します。ただし、ワーカプロセスの生成は、ジョブ運用ソフトウェアが提供するpjaexe コマンドを利用します。

ユーザーは、ジョブの用途に合った方式を選択し、マスタ・ワーカ型ジョブを作成する必要があります。

2.1.1.5 MPIプログラムを実行するジョブの作成

MPI プログラムをジョブとして実行する場合は、MPI プログラムが必要とするノード資源を正しく割り当てる必要があります。このため、ジョブスクリプトにその指定を記述する場合は考慮が必要です。

詳細については、"2.3.6 MPI ジョブの投入" を参照してください。

一般的にDevelopment Studio以外の MPI プログラムは、rsh や ssh コマンドを使用して複数のノードでプロセスをリモート実行します。しかし、ジョブ運用ソフトウェア上で MPI プログラムを実行するには、これらの替わりに pjrsh コマンドを使用してください。
pjrsh コマンドを使用する方法は、"付録C Development Studio以外の MPI 処理系の実行について" を参照してください。

また、MPI プログラム固有の環境変数を設定しなければいけない場合もあります。これについては MPI プログラムを作成するDevelopment Studioのマニュアル「MPI使用手引書」を参照してください。

2.1.1.6 仮想ノードにおける逐次プログラムの一斉実行 [PG]

複数の仮想ノードで逐次プログラムを一斉に実行するには pjexe コマンドを利用できます。

以下は、逐次プログラム prog を、8 個の仮想ノードで実行する例です。

```
pjexe --vnode 8 prog
```

2.1.1.7 PAPI ライブラリを使用するジョブの作成 [FX]

FXサーバに投入するジョブ内で、以下のようなプログラムを実行する場合、xospastop コマンド (/usr/bin/xospastop) をそのプログラムより先に実行する必要があります。

- オープンソースソフトウェアの PAPI (Performance Application Programming Interface) ライブラリをリンクしたプログラム
  (オープンソースソフトウェアの TAU や Scalasca などを利用したプログラムも含む)

[記述例]

```
#!/bin/bash
xospastop            ← 追加する記述
./a.out
```

xospastop コマンドを実行しない場合、上記のようなプログラムは正常に動作しない可能性があります。

なお、Development Studioの以下の機能を使用する場合は、xospastop コマンドを実行する必要はありません。

- プロファイラ
- 実行時情報出力機能

参考

xospastop コマンドは FXサーバ用に提供されるコマンドです。
本コマンドは、当該ジョブにおけるPMUカウンタ(Performance Monitoring Unit Counter)の採取を停止します。オプションやメッセージ出力はありません。終了ステータスは常に0です。

2.1.1.8 ジョブ実行性能への外乱の対処

ジョブの実行性能に影響する外乱については、それぞれ以下のように対処してください。

- mpiexecコマンドの標準出力/標準エラー出力による外乱 [FX]

  mpiexecコマンドの標準出力や標準エラー出力を それぞれ1つのファイルにまとめる場合、ジョブの実行性能に影響を与えることがあります。この性能影響を抑止するためには、mpiexecコマンドの標準出力や標準エラー出力はランクごとに別のファイルに出力するようにしてください。詳細は"2.3.6.9 mpiexecコマンドの標準出力/標準エラー出力 [FX]"を参照してください。

- ジョブ資源管理機能のジョブ統計情報の収集処理による外乱 [FX]

  ジョブ資源管理機能によるジョブ統計情報の定期収集処理がジョブ実行性能に影響することがあります。ジョブ内でxospastopコマンドを実行することでこの外乱を避けることができます。詳細は"2.1.1.7 PAPI ライブラリを使用するジョブの作成 [FX]"を参照してください。

2.1.2 ジョブ内での環境変数

ジョブ内では、ジョブ運用ソフトウェアによって、環境変数が設定されています。

表2.1 ジョブ内で参照可能な環境変数

環境変数名	詳細
PJM_ADAPTIVE_ELAPSED_TIME_MAX [FX]	pjsub コマンドの -L elapse オプションで指定したジョブの経過時間制限の最大値(秒) "-L elapse=30:00-1:00:00" のように範囲で指定した場合に、最大値(1:00:00)が秒数 (3600) で設定されます。 -L elapse オプションが指定されなかった場合は、ジョブACL機能で定義されているデフォルト値が設定されます。 "-L elapse=30:00" のように指定した場合は、この環境変数自体が設定されません。
PJM_ADAPTIVE_ELAPSED_TIME_MIN [FX]	pjsub コマンドの -L elapse オプションで指定した、ジョブの実行可能時間の最小値(秒) "-L elapse=30:00-1:00:00" のように実行可能時間を最小値と最大値で指定した場合に、最小値(30:00) が秒数 (1800) で設定されます。 -L elapse オプションが指定されなかった場合は、ジョブACL機能で定義されているデフォルト値が設定されます。 "-L elapse=30:00" のように指定した場合は、この環境変数自体が設定されません。
PJM_APPNAME	pjsub コマンドの --appname オプションで指定した文字列 オプションが指定されなかった場合、この環境変数は設定されません。
PJM_ASSIGN_LOGICAL_CPU [PG]	ジョブプロセスが動作する論理CPUの範囲 「Development Studioで作成した C/C++/Fortran プログラム」以外のジョブプロセスで有効です。 job: ジョブプロセスは、割り当てられたCPUコア内のジョブ用論理CPUだ

環境変数名	詳細
	けを使用できます。 all: ジョブプロセスは、割り当てられたCPUコア内のすべての論理CPUを使用できます。 ジョブ用論理CPUの詳細は、"C.5 ジョブで利用できる CPU 資源の範囲の設定 [PG]" を参照してください。
PJM_ASSIGN_ONLINE_NODE [FX]	pjsub コマンドの --mpi assign-online-node オプションが指定された場合は 1が設定されます。 オプションが指定されなかった場合、この環境変数は設定されません。
PJM_AT	pjsubコマンドの--atオプションで指定したジョブの実行開始時刻 *YYYY*/*MM*/*DD hh*:*mm*:*ss* (*YYYY*:年、*MM*:月、*DD*:日、*hh*:時、*mm*:分、*ss*:秒) オプションが指定されなかった場合、この環境変数は設定されません。
PJM_BULKNUM	バルク番号 (バルクジョブだけ設定)
PJM_COMMENT	pjsub コマンドの --comment オプションで指定した文字列 オプションが指定されなかった場合、この環境変数は設定されません。
PJM_CORE_MEM_LIMIT [PG]	pjsub コマンドの -L core-mem オプションで指定した、CPUコア当たりのメモリ量 (バイト) オプションが指定されなかった場合、この環境変数は設定されません。
PJM_CUSTOM_RESOURCES	pjsub コマンドの -L オプションで指定したカスタム資源名とその要求量 (または要求種別) 例: *customrscname*=1 (*customrscname* には定義されているカスタム資源名が表示されます) オプションが指定されなかった場合は、ジョブACL機能で定義されているデフォルト値が設定されます。 ジョブACL機能にデフォルト値が定義されていない場合は、要求量は 0 が設定されます。この場合、要求種別は設定されません。 複数のカスタム資源を要求した場合は、資源の情報をコンマ "," で区切ります。 ノード単位のカスタム資源を要求した場合は、カスタム資源名に "/n" が付加されます。
PJM_DPREFIX	ジョブスクリプトで、ジョブ運用ソフトウェアへの指示行を示す接頭語 デフォルトでは"#PJM"が設定されますが、pjsubコマンドのオプション-Cまたは--dir-prefixで明示的に指定した場合、その値が設定されます。
PJM_ELAPSED_TIME_MODE [FX]	pjsub コマンドの -L elapse オプションで指定したジョブの実行可能時間の指定方式 以下のどちらかが設定されます。 ・ "fixed" "-L elapse=30:00" のように実行可能時間を指定した場合に設定されます。 ・ "adaptive" "-L elapse=30:00-" または "-L elapse=30:00-1:00:00" のように実行可能時間の最小値と最大値を指定した場合に設定されます。 -L elapseオプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_ELAPSE_LIMIT	pjsub コマンドの -L elapse オプションで指定したジョブの最大実行可能時間 (秒)

環境変数名	詳細
	オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_ENVIRONMENT	バッチジョブでは"BATCH"、会話型ジョブでは"INTERACT"が設定されます。
PJM_EXEC_POLICY [PG]	pjsubコマンドの -P exec-policy オプションで指定した実行モードポリシー simplex : SIMPLEX (ノードを専有) share : SHARE (ノードを共有) オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。ノード割り当てジョブの場合は設定されません。
PJM_FSNAME	pjsub コマンドの --fs オプションで指定した文字列 オプションが指定されなかった場合、この環境変数は設定されません。
PJM_JOBDIR	ジョブスクリプト実行開始時のカレントディレクトリのパス
PJM_JOBID	ジョブID
PJM_JOBNAME	ジョブ名
PJM_LLIO_ASYNC_CLOSE [FX]	第1階層ストレージおよび第2階層ストレージ上のファイルのクローズを非同期クローズにするか否かの動作 on: 非同期クローズ off: 同期クローズ on(非同期クローズ)が設定されている場合、ファイルのクローズ時に書出し完了は保証されません。 off(同期クローズ)が設定されている場合、ファイルのクローズ時に書出し完了は保証されます。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_AUTO_READAHEAD [FX]	ジョブが複数回続けて第1階層ストレージまたは第2階層ストレージ上の連続した領域を読み込もうとした場合、自動的に先読みをするか否かの動作 on: 先読みをします。 off: 先読みをしません。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_CN_CACHED_WRITE_SIZE [FX]	第1階層ストレージへの書込み時にキャッシュするか否かのしきい値(バイト) 第1階層ストレージへの書込みの際に、書込みサイズがこの値以下の場合は、すぐにはストレージに書き出さず、一時的に計算ノード内キャッシュにキャッシュします。 小さなデータについてはまとめて書き出すことで、第1階層ストレージへの転送回数を減らし、性能を向上させるためのパラメーターです。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_CN_CACHE_SIZE [FX]	第1階層ストレージ上の計算ノード内キャッシュのサイズ(バイト) 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_CN_READ_CACHE [FX]	第1階層ストレージまたは第2階層ストレージから読み込んだファイルを計算ノード内キャッシュにキャッシュするか否かの動作

環境変数名	詳細
	on: キャッシュします。 off: キャッシュしません。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_LOCALTMP_SIZE [FX]	第1階層ストレージ上のノード内テンポラリ領域のサイズ(バイト) 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。 なお、ノード内テンポラリ領域のパスは環境変数PJM_LOCALTMPに設定されます。
PJM_LLIO_PERF [FX]	LLIO性能情報ファイルを出力するか否かの動作 on : 出力します。 off : 出力しません。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_PERF_PATH [FX]	LLIO性能情報ファイルのパス LLIO性能情報ファイルを出力する場合だけ設定されます。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_SHAREDTMP_SIZE [FX]	第1階層ストレージ上の共有テンポラリ領域のサイズ(バイト) 注) 1つのジョブが利用できる共有テンポラリ領域のサイズは、このサイズに、割り当てられた計算ノード数を乗じた値です。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。 なお、共有テンポラリ領域のパスは環境変数PJM_SHAREDTMPに設定されます。
PJM_LLIO_SIO_READ_CACHE [FX]	第2階層ストレージから計算ノードへ読み込んだファイルを第1階層ストレージにキャッシュするか否かの動作 on: キャッシュします。 off: キャッシュしません。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_STRIPE_COUNT [FX]	第1階層ストレージにファイルを分散配置する際のファイル当たりのストライプ数 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_STRIPE_SIZE [FX]	第1階層ストレージにファイルを分散配置する際のストライプサイズ(バイト) 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_LLIO_UNCOMPLETED_FILEINFO_PATH [FX]	未書出しファイル情報の出力先パス ジョブ終了時に、第2階層ストレージへの書出しが完了していないファイルがキャッシュに残っていた場合に、ファイル名の一覧を出力するファイルのパスです。 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。

環境変数名	詳細
PJM_LOCALTMP [FX]	第1階層ストレージ上のノード内テンポラリ領域のパス 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_MAILOPTION	pjsubコマンドの-mオプションで指定した、メール通知についての動作 以下の値の論理和が16進数表記で設定されます。 0x1 : ジョブの実行開始時に通知します。(-m b) 0x2 : ジョブの終了時に通知します。(-m e) 0x4 : ジョブの再実行時に通知します。(-m r) 0x8 : ジョブ統計情報を通知に含めます。(-m s) 0x10 : ジョブ統計情報とノード統計情報を通知に含めます。(-m S) 例: PJM_MAILOPTION=0x3 オプションが指定されなかった場合、この環境変数は設定されません。
PJM_MAILSENDTO	pjsub コマンドの --mail-list オプションで指定したメール送信先ユーザー オプションが指定されなかった場合、この環境変数は設定されません。
PJM_MPI_PROC	pjsub コマンドの --mpi proc オプションで指定した、MPI プログラムの起動時に生成される最大プロセス数
PJM_MPI_SHAPE_X PJM_MPI_SHAPE_Y PJM_MPI_SHAPE_Z [FX]	pjsub コマンドの --mpi shape オプションで指定したプロセスの形状における、X、Y、および Z 軸方向のノード数 オプションが指定されなかった場合、pjsub コマンドの -L node オプションで指定されるノード形状でのノード数になります。 また、形状の次元数に対応しない環境変数は設定されません。例えば、1次元形状の場合は、環境変数 PJM_MPI_SHAPE_Y と PJM_MPI_SHAPE_Z は設定されません。
PJM_NET_ROUTE [FX]	ジョブが通信経路として使用しているTofuインターコネクトのリンクダウンが発生したときの動作 dynamic : 通信経路を変更して、ジョブの実行を継続します。 static : 通信経路は変更しません。ジョブは異常終了します。
PJM_NODE	pjsub コマンドの -L node オプションで指定された値 (FXサーバの場合は、指定されたノード形状から求まるノード数) オプションが指定されなかった場合、ジョブACL機能で設定されるデフォルト値になります ("2.2.2 制限情報の確認" に示される pjacl コマンドの表示項目 "pjsub option parameter" の "(node=)")。 この環境変数は物理ノードを割り当てる場合だけ設定され、仮想ノードを割り当てる場合は設定されません。
PJM_NODE_ALLOCATION_IO_MODE [FX]	pjsub コマンドの -L node オプションで指定した、I/O専有に関するノードの割り当て方法 io-exclusive : I/O専有モード no-io-exclusive : I/O共有モード オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_NODE_ALLOCATION_MODE [FX]	pjsub コマンドの -L node オプションで指定した、ノードの割り当て方法 torus : トーラスモード mesh : メッシュモード noncont : 離散割り当て オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。

環境変数名	詳細
PJM_NODE_MEM_LIMIT [FX]	pjsub コマンドの -L node-mem オプションで指定した、1つのノードにおけるメモリ使用量の上限 (バイト) オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_NODE_X PJM_NODE_Y PJM_NODE_Z [FX]	pjsub コマンドの -L node オプションで指定した形状における、X 、Y、および Z軸方向のノード数 形状の次元数に対応しない環境変数は設定されません。例えば、1次元形状の場合は、環境変数 PJM_NODE_Y と PJM_NODE_Z は設定されません。
PJM_NORESTART	pjsub コマンドの --norestart オプションが指定された場合は1が設定されます。 オプションが指定されなかった場合、この環境変数は設定されません。
PJM_O_HOME	pjsub コマンドを実行したユーザーの環境変数 HOME
PJM_O_HOST	pjsub コマンドを実行したホスト名
PJM_O_LANG	pjsub コマンドを実行したユーザーの環境変数 LANG
PJM_O_LOGNAME	pjsub コマンドを実行したユーザーの環境変数 LOGNAME
PJM_O_MAIL	pjsub コマンドを実行したユーザーの環境変数 MAIL
PJM_O_NODEINF [PG]	割り当てられたノードリストファイルのパス 仮想ノードを割り当てるジョブの場合は、仮想ノードが配置される物理ノードのIPアドレスが1行に1つ記述されます。
PJM_O_PATH	pjsub コマンドを実行したユーザーの環境変数 PATH
PJM_O_SHELL	pjsub コマンドを実行したユーザーの環境変数 SHELL
PJM_O_TZ	pjsub コマンドを実行したユーザーの環境変数 TZ
PJM_O_WORKDIR	pjsub コマンドを実行したときのカレントディレクトリ
PJM_PROC_BY_NODE	MPI プログラムが1つの物理ノードまたは1つの仮想ノードに生成する最大プロセス数 ただし、物理ノードを1つ(node=1)または仮想ノードを1つ(vnode=1)割り当てている状態で、pjsubコマンドの--mpiオプションが指定されなかった場合、この環境変数は設定されません。
PJM_RANK_MAP_BYCHIP	pjsub コマンドの --mpi rank-map-bychip オプションで指定した、ランクの割り当てルール ・ FXサーバ XY、YX、XYZ、XZY、YXZ、YZX、ZXY、または ZYX の文字列が設定されます。 ・ PRIMERGYサーバ 1つの物理ノードに割り当てる仮想ノード数が設定されます。 オプションが指定されなかった場合、この環境変数は設定されません。オプションの引数が指定されなかった場合は、環境変数に DEF と表示されます。
PJM_RANK_MAP_BYNODE	pjsub コマンドの --mpi rank-map-bynode オプションで指定した、ランクの割り当てルール ・ FXサーバ XY、YX、XYZ、XZY、YXZ、YZX、ZXY、または ZYX の文字列が設定されます。

環境変数名	詳細
	・ PRIMERGYサーバ オプションが指定された場合は1が設定されます。 オプションが指定されなかった場合、この環境変数は設定されません。 オプションの引数が指定されなかった場合は、環境変数に DEF と表示されます。
PJM_RSCGRP	pjsub コマンドの -L rscgrp または -L rg オプションで指定したリソースグループ名 オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_RSCUNIT	pjsub コマンドの -L rscunit オプションまたは -L ru オプションで指定したリソースユニット名 オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_SHAREDTMP [FX]	第1階層ストレージ上の共有テンポラリ領域のパス 階層化ストレージを利用しないシステムでは、この環境変数は設定されません。
PJM_SHELL	ジョブを実行するシェルのパス
PJM_STDERR_PATH	pjsub コマンドの -e オプションで指定した、ジョブの標準エラー出力ファイルのパス オプションが指定されなかった場合、デフォルトのファイル名 "*ジョブ名*.e.*ジョブID*"が設定されます。ただし、バルクジョブとステップジョブでは、"*ジョブID*" はサブジョブIDになります。 pjsub コマンドの -j オプションを指定した場合は、環境変数 PJM_STDOUT_PATH の値と同じになります。
PJM_STDOUT_PATH	pjsub コマンドの -o オプションで指定した、ジョブの標準出力ファイルのパス オプションが指定されなかった場合、デフォルトのファイル名 "*ジョブ名*.o.*ジョブID*"が設定されます。ただし、バルクジョブとステップジョブでは、"*ジョブID*" はサブジョブIDになります。
PJM_STEPNUM	ステップ番号 (ステップジョブだけ設定)
PJM_SUBJOBID	サブジョブID 通常ジョブの場合は、ジョブIDが設定されます。
PJM_VNODE	pjsub コマンドの -L vnode オプションで指定された値 オプションが指定されなかった場合、ジョブACL機能の項目で設定されるデフォルト値になります ("2.2.2 制限情報の確認" に示される pjacl コマンドの表示項目 "pjsub option parameter" の "(vnode=)")。 この環境変数は仮想ノードを割り当てる場合だけ設定され、物理ノードを割り当てる場合は設定されません。
PJM_VNODE_CORE	pjsub コマンドの -L vnode-core オプションまたは -L vnode=(core=) オプションで指定された値 オプションが指定されなかった場合、ジョブACL機能の項目で設定されるデフォルト値になります ("2.2.2 制限情報の確認" に示される pjacl コマンドの表示項目 "pjsub option parameter" の "(vnode-core=)")。 この環境変数は仮想ノードを割り当てる場合だけ設定され、物理ノードを割り当てる場合は設定されません。
PJM_VNODE_MEM_LIMIT [PG]	pjsub コマンドの -L vnode-mem オプションまたは -L mem オプションで指定した仮想ノード当たりのメモリ量 (バイト)

環境変数名	詳細
	オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
PJM_VN_POLICY [PG]	pjsub コマンドの -P vn-policy オプションで指定した仮想ノード配置ポリシー abs-pack : Absolutely PACK pack : PACK abs-unpack : Absolutely UNPACK unpack : UNPACK オプションが指定されなかった場合、ジョブACL機能で定義されているデフォルト値が設定されます。
そのほか	pjsubコマンドの-xまたは-Xオプションで指定された環境変数

注意

- "表2.1 ジョブ内で参照可能な環境変数"で示した環境変数と同名の環境変数をpjsubコマンドの-xオプションに指定しても、その値は無視され、"表2.1 ジョブ内で参照可能な環境変数"で示した値になります。
- pjsubコマンドを実行するシェル上で設定されている環境変数は、指示がない限り、ジョブ内には引き継がれません。ジョブ内に環境変数を引き継ぐためには、pjsub コマンドの-Xオプションを指定してください。
  ただし、"表2.1 ジョブ内で参照可能な環境変数"で示した環境変数と同名の環境変数はpjsubコマンドの-Xオプションを指定しても、その値は無視され、"表2.1 ジョブ内で参照可能な環境変数"で示した値になります。
- エンドユーザは"PJM_PK_"で始まる環境変数を使用しないでください。"PJM_PK_"で始まる環境変数は、パワーノブ操作機能が使用します。パワーノブ操作機能については、マニュアル「ジョブ運用ソフトウェア APIユーザーズガイド Power API編」を参照してください。
- "PJM_LLIO_" で始まる環境変数は、LLIO の機能で使用します。詳細は、マニュアル「LLIO ユーザーズガイド」を参照してください。
  エンドユーザは "表2.1 ジョブ内で参照可能な環境変数" に記載されていない "PJM_LLIO_" で始まる環境変数を使用しないでください。
- ジョブ内での環境変数LD_LIBRARY_PATHについて
  プログラムの実行に必要な共有ライブラリをカレントディレクトリに配置しない場合は、環境変数LD_LIBRARY_PATHにカレントディレクトリを含めないでください。
  FEFSファイルシステム上にあるカレントディレクトリを検索パスに含めると、ファイルへの不必要なアクセスが増え、FEFSファイルシステムが高負荷になることがあります。

  以下にカレントディレクトリが検索対象になる設定例を示します。

```
a) LD_LIBRARY_PATH=/usr/local/lib:
b) LD_LIBRARY_PATH=/usr/local/lib:.
c) LD_LIBRARY_PATH=/usr/local/lib:.:/usr/lib
d) LD_LIBRARY_PATH=:/usr/local/lib
e) LD_LIBRARY_PATH=/usr/local/lib::/usr/lib
```

  なお、a、d、およびeの場合は、明にカレントディレクトリを指定していませんが、検索対象になります。
  このような場合は、

```
a)、d) LD_LIBRARY_PATH=/usr/local/lib
e) LD_LIBRARY_PATH=/usr/local/lib:/usr/lib
```

  と設定すれば、カレントディレクトリは検索対象にはなりません。

2.1.3 ユーザープログラムの作成方法

プログラムの作成方法については、Development Studioが提供するマニュアルを参照してください。ジョブ内で実行する MPI プログラムやコンパイラによる自動並列化プログラムなどの作成方法についても同様です。

2.1.4 ジョブが利用できるファイルシステム

ジョブの実行に必要なジョブスクリプト、ユーザーが作成したプログラム、およびデータファイルはジョブを投入するログインノード上に配置します。ただし、ジョブを実行する計算ノードでもこれらのファイルにアクセスできる必要があるため、ログインノードと計算ノードで同じパスでアクセスできる共有ファイルシステム上に配置してください。

ジョブの実行結果としてファイルを作成する場合、この共有ファイルシステム上に出力するようにします。これにより、ジョブが計算ノード上で出力したファイルをログインノード上で参照できます。共有ファイルシステムのパスは、システムごとに異なりますので、管理者にお問い合わせください。

[階層化ストレージについて]

階層化ストレージが利用できるシステムでは、第1階層ストレージのノード内テンポラリ領域と共有テンポラリ領域をジョブ用の高速なストレージとして使用できます。これらの領域はジョブごとに確保され、そのジョブだけが使用できます。ノード内テンポラリ領域と共有テンポラリ領域のパスは、それぞれジョブ内で設定されている環境変数PJM_LOCALTMPとPJM_SHAREDTMPで知ることができます。

以下は、ノード内テンポラリ領域を一時的なデータの格納先として利用する簡単なジョブの例です。

1. プログラムprog1がノード内テンポラリ領域${PJM_LOCALTMP}に計算結果out.dataを出力します。

```
prog1 -o ${PJM_LOCALTMP}/out.data
```

2. ファイルout.dataをプログラムprog2が読み込んで、ディレクトリ${PJM_LOCALTMP}に最終的な計算結果result.dataを出力します。

```
prog2 -i ${PJM_LOCALTMP}/out.data -o ${PJM_LOCALTMP}/result.data
```

3. ジョブの終了前にファイルresult.dataをジョブ実行時のディレクトリ${PJM_JOBDIR}(グローバルファイルシステム上)に、ジョブID ${PJM_JOBID}をつけた名前に変えて退避します。

```
cp ${PJM_LOCALTMP}/result.data ${PJM_JOBDIR}/result_${PJM_JOBID}.data
```

注意

ノード内テンポラリ領域と共有テンポラリ領域に出力したファイルは必要であればジョブ終了前に第2階層ストレージ上へ退避してください。ノード内テンポラリ領域と共有テンポラリ領域は一時的な領域であり、ジョブ終了後に削除されます。

ジョブ内に第2階層ストレージ上のファイルにアクセスするプログラムがある場合、ジョブ内でプログラムの実行前にLLIOが提供するllio_transferコマンドでそのファイルを第2階層ストレージのキャッシュにコピーすることで、ファイルに対するアクセスの性能向上が期待できます。llio_transferコマンドの詳細は、マニュアル「LLIOユーザーズガイド」の付録"リファレンス"を参照してください。

以下に、ジョブ内でllio_transferコマンドを使用する例を示します。

```
#!/bin/bash
#PJM -L "node=2"
#PJM --llio sharedtmp-size=5Mi,localtmp-size=10Gi
llio_transfer /gfs1/user1/a.out                 ← ファイルa.outを第2階層ストレージのキャッシュへコピー
mpiexec /gfs1/user1/a.out                       ← プログラム a.out を実行
llio_transfer --purge /gfs1/user1/a.out         ← ファイルa.outを第2階層ストレージのキャッシュから削除
```

注意

llio_transferコマンドで第2階層ストレージのキャッシュへコピーしたファイルを共通ファイルと呼びます。共通ファイルに関して、以下に注意してください。

- 共通ファイルに対する第2階層ストレージのキャッシュ領域経由の削除、ファイルデータ更新やファイル属性更新の操作はエラーで失敗します。
- 共通ファイルに対するファイルロックは未サポートです。
- ジョブ実行中は第2階層ストレージにある共通ファイルのコピー元を変更したり削除したりしないでください。これを変更した場合は、第2階層ストレージのキャッシュにコピーした共通ファイルの内容が不定になる可能性があります。また、これを削除した場合は、共通ファ

イルをllio_transferコマンドの--purgeオプションで削除できなくなり、ジョブが終了するまで第2階層ストレージのキャッシュ領域が共通ファイルに占有されたままになります。

- ジョブスクリプト内で共通ファイルを削除する場合は、llio_transferコマンドに--purgeオプションを指定して削除してください。rmコマンドはエラーで失敗し、削除できません。
- llio_transfer --purgeで第2階層ストレージのキャッシュから共通ファイルを削除した直後は、ジョブからの共通ファイルのコピー元ファイルの削除、ファイルデータ更新やファイル属性更新の操作がエラーで失敗する場合があります。その場合は、60秒待ってから、操作を再実行してください。
- 共通ファイルのコピー先である、第1階層ストレージの"第2階層ストレージのキャッシュ領域"には共通ファイル全体を格納できるだけの容量が必要です。第2階層ストレージのキャッシュ領域にコピーされた共通ファイルは、この領域が一杯になっても削除されることはありません。第2階層ストレージのキャッシュ領域の空き容量が必要な場合には、ジョブの途中で、llio_transferコマンドに--purgeオプションを指定して削除してください。なお、第2階層ストレージのキャッシュ領域の共通ファイルは、ジョブ終了後には自動的に削除されます。
- コマンドが共通ファイルの領域を確保する前に、ジョブ内で転送元ファイルのキャッシュデータが第1階層ストレージに存在する場合、llio_transferコマンドはエラーになります。ファイルオープンをした場合にキャッシュデータが作成される場合があるため、llio_transferコマンド実行前にジョブ内でファイルオープンをしないでください。以下のコマンドも、ファイルオープンする場合があるため、使用しないでください。
  - lfs setstripe
  - lfs getstripe

  なお、llio_transferコマンドがエラーになった場合にジョブを続行すると、共通ファイルが配布されていない状態でジョブは実行されます。
- llio_transferコマンドに指定できるファイルは以下の条件を満たす必要があります。
  - 本コマンド実行ユーザーがファイルにアクセス可能、かつ、
  - 第2階層ストレージのファイル、かつ、
  - ファイルの種類が通常ファイル、かつ、
  - ファイルサイズが1Byte以上のファイル

2.2 ジョブに関する情報の確認

ここでは、ジョブを投入する前にユーザーが確認すべき情報について説明します。

2.2.1 リソースユニット、リソースグループの確認

ユーザーは、ジョブを投入するリソースグループやリソースユニットの大きさや形状が、投入するジョブに対して十分かどうかを確認します。また、リソースユニットやリソースグループは管理者によってジョブの投入が抑制されている場合がありますので、ユーザーはこの状況についても確認します。
pjstat コマンドの --rsc オプションを使用すると、リソースユニットやリソースグループの情報を確認できます。

```
$ pjstat --rsc
RSCUNIT                    RSCUNIT_SIZE  RSCGRP                   RSCGRP_SIZE
unit1[ENABLE,START]        20x9x16       group1[ENABLE,START]     20x3x16
unit1[ENABLE,START]        20x9x16       group2[ENABLE,STOP]      20x6x16
unit2[ENABLE,STOP]         20x9x16       group1[ENABLE,START]     1500
unit3[DISABLE,STOP]        4x18x16       group1[ENABLE,START]     1152
```

表2.2 pjstat コマンドの表示項目(1)

項目	説明
RSCUNIT	リソースユニット名とその状態 書式: *リソースユニット名*[*状態*,...] 「*状態*」には以下があります。 ENABLE: ジョブの投入は可能 DISABLE: ジョブの投入は不可

項目	説明
	START : ジョブは実行可能 STOP : ジョブは実行不可
RSCUNIT_SIZE	リソースユニットのサイズ ・ FXサーバ Tofu 単位の数を1辺の長さとする X、Y、Z軸方向の直方体として表現されます。 書式: *X*x*Y*x*Z* ・ PRIMERGYサーバ リソースユニットを構成するノード数 *N* が表示されます。
RSCGRP	リソースグループ名とその状態 書式: *リソースグループ名*[*状態*,...] 「*状態*」の意味は、前述の項目 RSCUNIT と同様です。
RSCGRP_SIZE	リソースグループのサイズ ・ FXサーバ ノード数を1辺の長さとする、直方体、またはノード数で表現されます。 書式: *X*x*Y*x*Z* (直方体) または *N* (ノード数) ・ PRIMERGYサーバ リソースグループを構成するノード数 *N* が表示されます。

リソースユニットやリソースグループにおける計算ノードの機種については、管理者にお問い合わせください。

ジョブに割り当てるノード数やノード形状は、リソースグループのサイズ(上記表示の RSCGRP_SIZE)に収まる必要があります。このサイズを超えた場合、ノード不足となり、ジョブは実行されません。

ただし、FXサーバのリソースグループでは、その構成によっては、項目 RSCGRP_SIZE が示すすべてのノードをジョブに割り当てられるわけではありません。
ジョブに割り当てることができるノードの最大形状は、pjstat コマンドの --shape オプションで表示される項目 MAX_SIZE から"表2.4 割り当て可能なノードのサイズ [FX]" のように求められます。

```
$ pjstat --rsc --shape
RSCUNIT                        RSCUNIT_SIZE  RSCGRP                      RSCGRP_SIZE   MAX_SIZE
rscunit000[ENABLE,START]       3x2x9         rscgroup1[ENABLE,START]     540           4x6x18
rscunit000[ENABLE,START]       3x2x9         rscgroup2[ENABLE,START]     2x3x18        2x3x18
rscunit000[ENABLE,START]       3x2x9         rscgroup3[ENABLE,START]     2x3x18        2x3x18
rscunit000[ENABLE,START]       3x2x9         rscgroup4[ENABLE,START]     540           4x6x18

※上記は FXサーバのリソースグループの例です。
```

表2.3 pjstat コマンドの表示項目(2)

項目	説明
MAX_SIZE	ジョブに割り当てることができるノードの最大サイズ ・ FXサーバ ノード数を1辺の長さとする、直方体 *X*x*Y*x*Z* で表現されます。 最大形状が複数存在する場合は、コンマで区切ってすべて表示されます。 ・ PRIMERGYサーバ 項目 RSCGRP_SIZE と同じノード数 *N* が表示されます。

表2.4 割り当て可能なノードのサイズ [FX]

ノードの形状	割り当て可能なノードのサイズ
1次元形状	・ トーラスモードまたはメッシュモードの場合 ノード数は、項目MAX_SIZEが示す形状の積 *X* Y* Z* 以下であること。 項目RSCGRP_SIZEがノード数 *N* で表示されている場合は、さらに、ノード数が *N* 以下であること。

ノードの形状	割り当て可能なノードのサイズ
	・ 離散割り当ての場合 ノード数は、項目 RSCGRP_SIZE が示す形状の積 *X** *Y***Z* またはノード数 *N* 以下であること。 離散割り当ての場合は、項目MAX_SIZEは関係ありません。
2次元形状	・ トーラスモードまたはメッシュモードの場合 項目MAX_SIZE(*X*x *Y*x*Z*)から求められる形状 (*X**3)x(*Y***Z*/3)、またはそれを回転させた形状に収まること。 項目 RSCGRP_SIZE がノード数 *N* で表示されている場合は、さらに、ノード数が *N* 以下であること。 ・ 離散割り当ての場合 ノード数は、項目 RSCGRP_SIZE が示す形状の積 *X** *Y***Z* またはノード数 *N* 以下であること。 離散割り当ての場合は、項目MAX_SIZEは関係ありません。
3次元形状	・ トーラスモードまたはメッシュモードの場合 項目MAX_SIZEが示す形状 *X*x *Y*x*Z*、またはそれを回転させた形状に収まること。 項目 RSCGRP_SIZE がノード数 *N* で表示されている場合は、さらに、ノード数が *N* 以下であること。 ・ 離散割り当ての場合 ノード数は、項目 RSCGRP_SIZE が示す形状の積 *X** *Y***Z* またはノード数 *N* 以下であること。 離散割り当ての場合は、項目MAX_SIZEは関係ありません。

[例]

前述の表示例では、リソースグループ rscgroup1 はノード数が 540 ですが、項目 MAX_SIZE が 4x6x18 なので、割り当て可能な最大ノード数は以下になります。

－ 1次元形状の場合
トーラスモード、メッシュモードの場合は共に、ノード数は RSCGRP_SIZE=540ノード以下、かつ MAX_SIZE=4x6x18=432ノード以下であること。すなわち、最大ノード数は432ノードになります。
離散割り当ての場合は、RSCGRP_SIZE=540ノード以下であること。

－ 2次元形状の場合
トーラスモード、メッシュモードの場合は共に、最大形状は(4*3)x(6*18/3)=12x36、またはそれを回転させた形状36x12に収まり、さらにそのノード数は RSCGRP_SIZE=540ノード以下であること。すなわち、最大ノード数は最大形状に制限され、12x36=432ノードになります。
離散割り当ての場合は、RSCGRP_SIZE=540ノード以下であること。

－ 3次元形状の場合
トーラスモード、メッシュモードの場合は共に、最大形状は4x6x18、またはそれを回転させた形状4x18x6、 6x4x18、6x18x4、18x6x4、18x4x6のどれかに収まり(":strict"を指定している場合は4x6x18だけ)、さらにそのノード数は RSCGRP_SIZE=540ノード以下であること。すなわち、最大ノード数は最大形状に制限され、4x6x18=432ノードになります。
離散割り当ての場合は、RSCGRP_SIZE=540ノード以下であること。

2.2.2 制限情報の確認

ユーザーのシステム利用は、ジョブACL機能によって制限されます("1.10 ジョブACL機能" 参照)。ユーザーはその制限の範囲内でジョブを投入、実行しなければいけません。

ユーザーに適用される制限情報は、pjacl コマンドで確認できます。

```
$ pjacl
#
# JOBACL information
#
user    user1                                   ← (1)
group   group1                                  ← (2)

defines                                         ← (3)
    default rscunit             rscunit1        ← (4)
    default rscgroup            rscgroup1       ← (5)
    default rscunit(interact)   rscunit1        ← (6)
    default rscgroup(interact)  rscgroup1       ← (7)
```

```
    default alloc granularity   node            ← (8)
    default elapsed time mode   fixed           ← (9)
    user priority               127             ← (10)
    fairshare init              0               ← (11)
    fairshare recovery          100             ← (12)
    ingroup priority            127             ← (13)
    ingroup fairshare init      0               ← (14)
    ingroup fairshare recovery  100             ← (15)
    node-priority               31              ← (16)
    numa-policy                 pack            ← (17)
    default allocation mode     torus           ← (18)
    default allocation-io-mode  no-io-exclusive ← (19)
    default strict-mode         nostrict        ← (20)
    net-route                   dynamic         ← (21)
    load-policy                 balancing       ← (22)
    vn-policy                   pack            ← (23)
    exec-policy                 share           ← (24)
    assign-logical-cpu          job             ← (25)
    pjstat display mode         anonymous       ← (26)
    mpiexec-stdouterr-unit      mpiexec         ← (27)
    mpiexec-stdout              %o              ← (28)
    mpiexec-stderr              %e              ← (29)
    mpiexec-stdout(interact)    noset           ← (30)
    mpiexec-stderr(interact)    noset           ← (31)
    mpiexec-std-emptyfile       on              ← (32)
    default llio-async-close    on              ← (33)
    default llio-auto-readahead on              ← (34)
    default llio-cn-read-cache  on              ← (35)
    default llio-sio-read-cache off             ← (36)
    default llio-uncompleted-fileinfo-path  %e  ← (37)
    default llio-perf           off             ← (38)
    default llio-perf-path      %n.%J.llio_perf ← (39)

groups                                          ← (40)
    group priority              127             ← (41)
    group fairshare init        0               ← (42)
    group fairshare recovery    100             ← (43)

pjsub option parameters                                                 ← (44)
    (-L/--rsc-list)                 lower     upper          default    ← (45)
        (elapse=)                   1         24:00:00       01:00:00   ← (46)
        (adaptive elapsed time min) 00:00:01  24:00:00       01:00:00   ← (47)
        (adaptive elapsed time max) 00:00:02  48:00:00       02:00:00   ← (48)
        (node elapse)               1         unlimited      -          ← (49) (*1)
        (adaptive node elapse min)  00:00:01  unlimited      -          ← (50) (*1)
        (adaptive node elapse max)  00:00:02  unlimited      -          ← (51) (*1)
        (total cores elapse)        1         unlimited      -          ← (52) (*1)
        (total cores)               1         unlimited      -          ← (53) (*1)
        (node=)                     1         2147483647     2          ← (54)
        (node-mem=)                 1         unlimited      unlimited  ← (55)
        (vnode=)                    1         2147483647     1          ← (56)
        (vnode-core=)               1         2147483647     1          ← (57)
        (vnode-mem=)                1         unlimited      unlimited  ← (58)
        (proc-core=)                -         unlimited      0          ← (59)
        (proc-cpu=)                 -         unlimited      unlimited  ← (60)
        (proc-crproc=)              -         512            512        ← (61)
        (proc-data=)                -         unlimited      unlimited  ← (62)
        (proc-lockm=)               -         unlimited      unlimited  ← (63)
        (proc-msgq=)                -         unlimited      unlimited  ← (64)
        (proc-openfd=)              -         unlimited      1024       ← (65)
        (proc-psig=)                -         unlimited      unlimited  ← (66)
        (proc-filesz=)              -         unlimited      unlimited  ← (67)
        (proc-stack=)               -         unlimited      unlimited  ← (68)
```

```
        (proc-vmem=)                    -           unlimited       unlimited       ← (69)
        (customrscname1=)               0           unlimited       0               ← (70) (*2)
        (custom-total-customrscname2=)  0           unlimited       -               ← (71) (*2)
        (custom-node-customrscname2=)   0           unlimited       0               ← (72) (*2)
        (custom-vnode-customrscname2=)  0           unlimited       0               ← (73) (*2)

    (-L/--rsc-list)                     select      default                         ← (74)
        (customrscname3=)               a,b,c       -                               ← (75) (*2)
        (customrscname4=)               a,b,c       -                               ← (*2)

    (--interact -L)                     lower       upper           default         ← (76)
        (elapse=)                       1           24:00:00        01:00:00
        (adaptive elapsed time min)     00:00:01    24:00:00        01:00:00
        (adaptive elapsed time max)     00:00:02    48:00:00        02:00:00
        (node elapse)                   1           unlimited       -               ← (*1)
        (adaptive node elapse min)      00:00:01    unlimited       -               ← (*1)
        (adaptive node elapse max)      00:00:02    unlimited       -               ← (*1)
        (total cores elapse)            1           unlimited       -               ← (*1)
        (total cores)                   1           unlimited       -               ← (*1)
        (node=)                         1           2147483647      1
        (node-mem=)                     1           unlimited       unlimited
        (vnode=)                        1           2147483647      1
        (vnode-core=)                   1           2147483647      1
        (vnode-mem=)                    1           unlimited       unlimited
        (proc-core=)                    -           unlimited       0
        (proc-cpu=)                     -           unlimited       unlimited
        (proc-crproc=)                  -           512             512
        (proc-data=)                    -           unlimited       unlimited
        (proc-lockm=)                   -           unlimited       unlimited
        (proc-msgq=)                    -           unlimited       unlimited
        (proc-openfd=)                  -           unlimited       1024
        (proc-psig=)                    -           unlimited       unlimited
        (proc-filesz=)                  -           unlimited       unlimited
        (proc-stack=)                   -           unlimited       unlimited
        (proc-vmem=)                    -           unlimited       unlimited
        (customrscname1=)               0           unlimited       0               ← (*2)
        (custom-total-customrscname2=)  0           unlimited       -               ← (*2)
        (custom-node-customrscname2=)   0           unlimited       0               ← (*2)
        (custom-node-customrscname2=)   0           unlimited       0               ← (*2)

    (-p)                                lower       upper           default         ← (77)
                                        -           255             127

    (--bulk --sparam)                   lower       upper           default         ← (78)
        (number of subjob)              -           unlimited       -

    (--llio)                            lower       upper           default         ← (79)
        (sharedtmp-size=)               0Mi         2147483647Mi    0Mi             ← (80)
        (localtmp-size=)                0Mi         2147483647Mi    0Mi             ← (81)
        (cn-cached-write-size=)         0Mi         2147483647Mi    4Mi             ← (82)
        (cn-cache-size=)                4Mi         2147483647Mi    128Mi           ← (83)
        (stripe-count=)                 1           2147483647      2147483647      ← (84)
        (stripe-size=)                  64Ki        4194240Ki       2048Ki          ← (85)

execute                                         ← (86)
    pjsub                       enable          ← (87)
    pjsub(--at)                 enable          ← (88)
    pjsub(batch)                enable          ← (89)
    pjsub(--interact)           enable          ← (90)
    pjsub(normal)               enable          ← (91)
    pjsub(--step)               enable          ← (92)
    pjsub(--bulk)               enable          ← (93)
    pjsub(--mswk)               disable         ← (94)
```

```
    pjsub(node)                     enable          ← (95)
    pjsub(vnode)                    disable         ← (96)
    pjstat                          enable          ← (97)
    pjdel                           enable          ← (98)
    pjdel(--enforce)                disable         ← (99)
    pjdel(--no-history)             disable         ← (100)
    pjhold                          enable          ← (101)
    pjhold(--enforce)               disable         ← (102)
    pjrls                           enable          ← (103)
    pjwait                          enable          ← (104)
    pjalter                         enable          ← (105)
    pjsig                           enable          ← (106)
    pjacl                           enable          ← (107)
    mpiexec(--std)                  enable          ← (108)
    mpiexec(--stdout)               enable          ← (109)
    mpiexec(--stderr)               enable          ← (110)
    mpiexec(--std)(interact)        enable          ← (111)
    mpiexec(--stdout)(interact)     enable          ← (112)
    mpiexec(--stderr)(interact)     enable          ← (113)
    pjsub(torus)                    enable          ← (114)
    pjsub(mesh)                     enable          ← (115)
    pjsub(noncont)                  enable          ← (116)
    pjsub(nostrict)                 enable          ← (117)
    pjsub(strict)                   enable          ← (118)
    pjsub(strict-io)                disable         ← (119)
    pjsub(no-io-exclusive)          enable          ← (120)
    pjsub(io-exclusive)             enable          ← (121)
    pjsub(-P vn-policy)             enable          ← (122)
    pjsub(-P exec-policy)           enable          ← (123)
    pjsub(fixed elapsed time)       enable          ← (124)
    pjsub(adaptive elapsed time)    disable         ← (125)
    pjsub(--net-route)              disable         ← (126)
    command-api                     disable         ← (127)

permit                                              ← (128)
    pjsub           allow own                       ← (129)
    pjstat          allow own                       ← (130)
    pjdel           allow own                       ← (131)
    pjhold          allow own                       ← (132)
    pjrls           allow own                       ← (133)
    pjwait          allow own                       ← (134)
    pjalter         allow own                       ← (135)
    pjsig           allow own                       ← (136)
    pjacl           allow own                       ← (137)
    pmerls          deny all                        ← (138)
    pmalter         deny all                        ← (139)
    pjshowrsc       allow own                       ← (140)

limit in rscunit (each users)                       ← (141)
    acceptable job                  unlimited       ← (142)
    acceptable all-subjob           unlimited       ← (143)
    acceptable bulk-subjob          unlimited       ← (144)
    acceptable step-subjob          unlimited       ← (145)
    running job                     unlimited       ← (146)
    running bulk-subjob             unlimited       ← (147)
    use node                        unlimited       ← (148)
    use core                        unlimited       ← (149)
    acceptable job(interact)        unlimited       ← (150)
    running job(interact)           unlimited       ← (151)
    use node(interact)              unlimited       ← (152)
    use core(interact)              unlimited       ← (153)
    customrscname1                  unlimited       ← (154) (*2)
    customrscname1(interact)        unlimited       ← (155) (*2)
```

```
limit in rscunit (total users in samegroup)      ← (156)
    acceptable job               unlimited
    acceptable all-subjob        unlimited
    acceptable bulk-subjob       unlimited
    acceptable step-subjob       unlimited
    running job                  unlimited
    running bulk-subjob          unlimited
    use node                     unlimited
    use core                     unlimited
    acceptable job(interact)     unlimited
    running job(interact)        unlimited
    use node(interact)           unlimited
    use core(interact)           unlimited
    customrscname1               unlimited      ← (*2)
    customrscname1(interact)     unlimited      ← (*2)

limit in rscunit (total all users)               ← (157)
    acceptable job               unlimited
    acceptable all-subjob        unlimited
    acceptable bulk-subjob       unlimited
    acceptable step-subjob       unlimited
    running job                  unlimited
    running bulk-subjob          unlimited
    use node                     unlimited
    use core                     unlimited
    acceptable job(interact)     unlimited
    running job(interact)        unlimited
    use node(interact)           unlimited
    use core(interact)           unlimited
    customrscname1               unlimited      ← (*2)
    customrscname1(interact)     unlimited      ← (*2)

limit in rscgroup (each users)                   ← (158)
    acceptable job               unlimited
    acceptable all-subjob        unlimited
    acceptable bulk-subjob       unlimited
    acceptable step-subjob       unlimited
    running job                  unlimited
    running bulk-subjob          unlimited
    use node                     unlimited
    use core                     unlimited
    acceptable job(interact)     unlimited
    running job(interact)        unlimited
    use node(interact)           unlimited
    use core(interact)           unlimited
    customrscname1               unlimited      ← (*2)
    customrscname1(interact)     unlimited      ← (*2)

limit in rscgroup (total users in samegroup)     ← (159)
    acceptable job               unlimited
    acceptable all-subjob        unlimited
    acceptable bulk-subjob       unlimited
    acceptable step-subjob       unlimited
    running job                  unlimited
    running bulk-subjob          unlimited
    use node                     unlimited
    use core                     unlimited
    acceptable job(interact)     unlimited
    running job(interact)        unlimited
    use node(interact)           unlimited
    use core(interact)           unlimited
    customrscname1               unlimited      ← (*2)
```

```
    customrscname1(interact)      unlimited      ← (*2)

limit in rscgroup (total all users)              ← (160)
    acceptable job                unlimited
    acceptable all-subjob         unlimited
    acceptable bulk-subjob        unlimited
    acceptable step-subjob        unlimited
    running job                   unlimited
    running bulk-subjob           unlimited
    use node                      unlimited
    use core                      unlimited
    acceptable job(interact)      unlimited
    running job(interact)         unlimited
    use node(interact)            unlimited
    use core(interact)            unlimited
    customrscname1                unlimited      ← (*2)
    customrscname1(interact)      unlimited      ← (*2)
```

表示の意味は以下のとおりです。

項番	説明
1	実行ユーザー名
2	実行グループ名
3	ユーザーに関する設定値
4	バッチジョブを投入するデフォルトのリソースユニット名
5	バッチジョブを投入するデフォルトのリソースグループ名
6	会話型ジョブを投入するデフォルトのリソースユニット名
7	会話型ジョブを投入するデフォルトのリソースグループ名
8	ノード資源の割り当て粒度
9	ジョブの実行可能時間の指定形式 ・ fixed : 実行可能時間を指定 実行可能時間は、pjsub option parameters の項目 (elapse=) のデフォルト値が使われます。 ・ adaptive : 実行可能時間の最小値と最大値を指定 実行可能時間の最小値と最大値はそれぞれ、pjsub option parameters の項目 (adaptive elapsed time min)、項目 (adaptive elapsed time max) のデフォルト値が使われます。
10	ユーザーの優先度
11	ユーザーのフェアシェア値の初期値
12	ユーザーのフェアシェア値回復率
13	グループにおけるユーザーの優先度
14	グループにおけるユーザーのフェアシェア値の初期値
15	グループにおけるユーザーのフェアシェア値回復率
16	ノード優先度 [PG]
17	NUMA割り当てポリシー
18	ノードの割り当て方法 [FX]
19	ジョブへのノード割り当てがI/O専有モードか否か [FX]
20	ノード割り当て時の strict モード [FX]
21	ジョブが使用しているTofuインターコネクトのリンクダウン時に通信経路を動的に変更するか否か [FX]
22	ジョブ分散、集中割り当て [PG]

項番	説明
23	仮想ノード配置ポリシー [PG]
24	実行モード(ノード) [PG]
25	ジョブプロセスが使用できる論理CPUの範囲 [PG]
26	pjstatコマンドのジョブ情報表示におけるアクセス制限
27	mpiexecコマンドの標準出力/標準エラー出力の出力単位 [FX]
28	バッチジョブにおけるmpiexecコマンドの標準出力のデフォルト出力先 [FX]
29	バッチジョブにおけるmpiexecコマンドの標準エラー出力のデフォルト出力先 [FX]
30	会話型ジョブにおけるmpiexecコマンドの標準出力のデフォルト出力先 [FX]
31	会話型ジョブにおけるmpiexecコマンドの標準エラー出力のデフォルト出力先 [FX]
32	mpiexecコマンドの標準出力/標準エラー出力がない場合に空ファイルを作成するか否か [FX]
33	階層化ストレージにおける非同期クローズ
34	階層化ストレージにおける自動先読み
35	階層化ストレージから読み込んだデータを計算ノード内にキャッシュするか否か
36	第2階層ストレージから読み込んだデータを第1階層ストレージにキャッシュするか否か
37	第1階層ストレージ上から第2階層ストレージへの書出しが完了できなかったファイル名を格納したファイル %で始まるメタ文字は出力ファイル名の書式で、pjsub コマンドの --llio uncompleted-fileinfo-path で指定できるメタ文字と同じ意味です。
38	LLIO性能情報を出力するか否か。
39	LLIO性能情報の出力先ファイル名 %で始まるメタ文字は出力ファイル名の書式で、pjsub コマンドの --llio perf-path で指定できるメタ文字と同じ意味です。
40	グループに関する設定値
41	グループの優先度
42	グループのフェアシェア値の初期値
43	グループのフェアシェア値回復率
44	pjsub コマンドのオプションで指定できる上限値、下限値、およびデフォルト値
45	バッチジョブにおける上限値、下限値、およびデフォルト値
46	ジョブ単位の実行可能時間
47	ジョブ単位の実行可能時間の最小値
48	ジョブ単位の実行可能時間の最大値
49	ジョブ単位の要求ノード数×実行可能時間 (*1)
50	ジョブ単位の要求ノード数×実行可能時間の最小値 (*1)
51	ジョブ単位の要求ノード数×実行可能時間の最大値 (*1)
52	ジョブ単位の CPU コア数×時間の制限 (*1)
53	ジョブ単位の CPU コア数制限 (*1)
54	ジョブ単位のノード数制限
55	ノード単位の使用メモリ制限
56	ジョブ単位の仮想ノード数制限
57	仮想ノード単位の CPU コア数制限
58	仮想ノード単位の割り当てメモリ量制限
59	プロセス単位のコアファイルサイズ制限

項番	説明
60	プロセス単位のCPU時間制限
61	プロセス単位のプロセス生成数制限
62	プロセス単位のデータセグメントサイズ制限
63	プロセス単位のロックメモリサイズ制限
64	プロセス単位のメッセージキューサイズ制限
65	プロセス単位のファイル記述子数制限
66	プロセス単位のペンディングシグナル数制限
67	プロセス単位のファイルサイズ制限
68	プロセス単位のスタックサイズ制限
69	プロセス単位の仮想メモリサイズ制限
70	リソースユニットまたはリソースグループ単位のカスタム資源の使用制限数 (*2)
71	ノード単位のカスタム資源の使用制限数 (1つのジョブに対する使用制限数) (*2)
72	ノード単位のカスタム資源の使用制限数 (1つのジョブ中の 1ノード当たりの使用制限数) (*2)
73	ノード単位のカスタム資源の使用制限数 (1つのジョブ中の 1仮想ノード当たりの使用制限数) (*2)
74	カスタム資源の種別とデフォルト値
75	カスタム資源の指定可能種別 (*2)
76	会話型ジョブにおける上限値、下限値、およびデフォルト値
77	ジョブ優先度の上限値・下限値とデフォルト値
78	バルクジョブのサブジョブ数制限
79	階層化ストレージに関するパラメーターの上限値、下限値、およびデフォルト値
80	共有テンポラリ領域のサイズ 1つのジョブが利用できる共有テンポラリ領域のサイズは、このサイズに、割り当てられた計算ノード数を乗じた値です。
81	ノード内テンポラリ領域のサイズ
82	第1階層ストレージへの書込み時にキャッシュするか否かのしきい値 第1階層ストレージへの書込みの際に、書込みサイズがこの値以下の場合は、すぐにはストレージに書き出さず、一時的に計算ノード内キャッシュにキャッシュします。
83	計算ノード内キャッシュのサイズ
84	第1階層ストレージにファイルを分散配置する際のファイル当たりのストライプ数
85	第1階層ストレージにファイルを分散配置する際のストライプサイズ
86	実行権限の有無
87	pjsub コマンド実行権限
88	実行開始時刻の指定権限
89	バッチジョブ投入権限
90	会話型ジョブ投入権限
91	通常ジョブ投入権限
92	ステップジョブ投入権限
93	バルクジョブ投入権限
94	マスタ・ワーカ型ジョブ投入権限
95	ノードを割り当てるジョブの投入権限
96	仮想ノードを割り当てるジョブの投入権限

項番	説明
97	pjstat コマンド実行権限
98	pjdelコマンド実行権限
99	pjdelでのプロローグ・エピローグスクリプトキャンセル権限
100	ジョブ削除時にジョブの履歴情報へのジョブ情報出力を抑止する指定の権限
101	pjhold コマンド実行権限
102	pjhold でのプロローグ・エピローグスクリプトキャンセル権限
103	pjrls コマンド実行権限
104	pjwait コマンド実行権限
105	pjalter コマンド実行権限
106	pjsig コマンド実行権限
107	pjacl コマンド実行権限
108	バッチジョブにおけるmpiexecコマンドの標準出力および標準エラー出力の出力先変更権限 [FX]
109	バッチジョブにおけるmpiexecコマンドの標準出力の出力先変更権限 [FX]
110	バッチジョブにおけるmpiexecコマンドの標準エラー出力の出力先変更権限 [FX]
111	会話型ジョブにおけるmpiexecコマンドの標準出力および標準エラー出力の出力先変更権限 [FX]
112	会話型ジョブにおけるmpiexecコマンドの標準出力の出力先変更権限 [FX]
113	会話型ジョブにおけるmpiexecコマンドの標準エラー出力の出力先変更権限 [FX]
114	トーラスモードでのジョブ投入権限 [FX]
115	メッシュモードでのジョブ投入権限 [FX]
116	離散割り当てでのジョブ投入権限 [FX]
117	ノード割り当てのstrictおよびstrict-io指定なしでのジョブ投入権限
118	ノード割り当てのstrict指定ありでのジョブ投入権限
119	ノード割り当てのstrict-io指定ありでのジョブ投入権限
120	I/O共有モード(no-io-exclusive)指定でのジョブ投入権限 [FX]
121	I/O専有モード(io-exclusive)指定でのジョブ投入権限 [FX]
122	仮想ノード配置ポリシーの指定権限 [PG]
123	実行モードポリシーの指定権限 [PG]
124	ジョブの実行可能時間を指定する形式 (pjsub -L elapse=*limit*) でのジョブの投入権限
125	ジョブの実行可能時間の最小値と最大値を指定する形式 (pjsub -L elapse=*min_limit*[-*max_limit*]) でのジョブの投入権限
126	ジョブが使用しているTofuインターコネクトがリンクダウンしたときに通信経路を動的に変更するかどうか(pjsub --net-route)の指定権限 [FX]
127	コマンドAPIの利用権限
128	操作対象の許可
129	pjsub コマンドで実行可能なグループの許可
130	pjstat コマンドの対象ジョブ
131	pjdel コマンドの対象ジョブ
132	pjhold コマンドの対象ジョブ
133	pjrls コマンドの対象ジョブ
134	pjwait コマンドの対象ジョブ

項番	説明
135	pjalter コマンドの対象ジョブ
136	pjsig コマンドの対象ジョブ
137	pjacl コマンドの対象ジョブ
138	pmerls コマンドの対象ジョブ
139	pmalter コマンドの対象ジョブ
140	pjshowrsc コマンドによる情報表示の許可ユーザー、グループ
141	リソースユニット内の制限値(ユーザーごと)
142	バッチジョブの同時受付け数制限
143	バルクジョブとステップジョブのサブジョブ、および通常ジョブ(バッチジョブ)の同時受付制限数
144	バルクジョブのサブジョブの同時受付制限数
145	ステップジョブのサブジョブの同時受付制限数
146	バッチジョブの同時実行数制限
147	バルクジョブのサブジョブ同時実行数制限
148	バッチジョブのノードの同時使用数制限
149	バッチジョブのCPUコアの同時使用数制限
150	会話型ジョブの同時受付け数制限
151	会話型ジョブの同時実行数制限
152	会話型ジョブのノードの同時使用数制限
153	会話型ジョブのCPUコアの同時使用数制限
154	リソースユニットまたはリソースグループ単位のカスタム資源同時使用制限数 (*2)
155	リソースユニットまたはリソースグループ単位の会話型ジョブにおけるカスタム資源同時使用制限数 (*2)
156	リソースユニット内の制限値(グループ内合計値)
157	リソースユニット内の制限値(全ユーザーの合計値)
158	リソースグループ内の制限値(ユーザーごと)
159	リソースグループ内の制限値(グループ内合計値)
160	リソースグループ内の制限値(全ユーザーの合計値)

(*1) これらに対応する pjsub コマンドのオプションはありません。
(*2) *customrscname1* から *customrscname4* には、実際には、定義されたカスタム資源名が表示されます。

上記表示における定義項目"permit"で、"allow" は操作許可、"deny" は操作拒否を示します。"allow" または "deny" に続いて示される操作対象は以下を意味します。

表2.5 pjacl コマンドの表示における操作対象の表記

表記	説明
own	自分自身のジョブだけ
all	すべてのユーザーのジョブ
g(g1, g2, ...)	対象となるグループ名 g1、g2、...
u(u1, u2, ...)	対象となるユーザー名 u1、u2、...

注意

- 本書で説明するジョブ運用ソフトウェアのコマンドは、管理者が行うジョブACL機能の設定により、特定ユーザーの実行が許可されていない場合があります。pjacl コマンドの出力結果を確認してください。
  なお、pjacl コマンドも特定ユーザーによる実行が許可されていない場合があります。
- ノード割り当てジョブはジョブ運用の設定により、ノードの同時使用数で制限される場合とCPUコアの同時使用数で制限される場合があります。ジョブ運用がどちらの設定になっているかは管理者にお問い合わせください。
  CPUコアの同時使用数で制限される設定の場合、ノード割り当てジョブのCPUコアの同時使用数は "ノードに搭載されているCPUコア数×要求ノード数" になります。

pjstat コマンドの --limit オプションでは、pjacl コマンドで表示される内容のうち、ユーザーのジョブ投入における資源の制限値と現在の割り当て量を確認できます。

- リソースユニット内での資源の制限値と割り当て量

```
$ pjstat --limit --rscunit rscunit1
 System Resource Information:
 RSCUNIT: rscunit1
 USER: user1
  LIMIT-NAME                          LIMIT         ALLOC
  ru-accept                         unlimited           5  ← (1)
  ru-accept-allsubjob               unlimited           0  ← (2)
  ru-accept-bulksubjob              unlimited           0  ← (3)
  ru-accept-stepsubjob              unlimited           0  ← (4)
  ru-run-job                        unlimited           3  ← (5)
  ru-run-bulksubjob                 unlimited           0  ← (6)
  ru-use-node                       unlimited        1011  ← (7)
  ru-interact-accept                unlimited           0  ← (8)
  ru-interact-run-job               unlimited           0  ← (9)
  ru-interact-use-node              unlimited           0  ← (10)
  ru-use-core                       unlimited          30  ← (11)
  ru-interact-use-core              unlimited           0  ← (12)
  ru-custom-customresourcename             10           1  ← (13) (*)
  ru-interact-custom-customresourcename     5           1  ← (14) (*)
 GROUP: group1
  LIMIT-NAME                          LIMIT         ALLOC
  ru-accept                         unlimited           5
  ru-accept-allsubjob               unlimited           0
  ru-accept-bulksubjob              unlimited           0
  ru-accept-stepsubjob              unlimited           0
  ru-run-job                        unlimited           3
  ru-run-bulksubjob                 unlimited           0
  ru-use-node                       unlimited        1011
  ru-interact-accept                unlimited           0
  ru-interact-run-job               unlimited           0
  ru-interact-use-node              unlimited           0
  ru-use-core                       unlimited          30
  ru-interact-use-core              unlimited           0
ALL:
  LIMIT-NAME                          LIMIT         ALLOC
  ru-accept                         unlimited           5
  ru-accept-allsubjob               unlimited           0
  ru-accept-bulksubjob              unlimited           0
  ru-accept-stepsubjob              unlimited           0
  ru-run-job                        unlimited           3
  ru-run-bulksubjob                 unlimited           0
  ru-use-node                       unlimited        1011
  ru-interact-accept                unlimited           0
  ru-interact-run-job               unlimited           0
  ru-interact-use-node              unlimited           0
```

```
 ru-use-core                    unlimited            30
 ru-interact-use-core           unlimited             0
```

(1) バッチジョブの同時受付け数
(2) バルクジョブとステップジョブのサブジョブ、および通常ジョブ(バッチジョブ)の同時受付制限数
(3) バルクジョブのサブジョブの同時受付制限数
(4) ステップジョブのサブジョブの同時受付制限数
(5) バッチジョブの同時実行数
(6) バルクジョブのサブジョブの同時実行制限数
(7) バッチジョブのノードの同時使用数
(8) 会話型ジョブの同時受付け数
(9) 会話型ジョブの同時実行数
(10) 会話型ジョブのノードの同時使用数
(11) バッチジョブのCPUコアの同時使用数
(12) 会話型ジョブのCPUコアの同時使用数
(13) カスタム資源の同時使用制限数 (*)
(14) 会話型ジョブのカスタム資源の同時使用制限数 (*)
(*) ru-custom-*customresourcename* および ru-interact-custom-*customresourcename* の *customresourcename* は、実際には、定義されたカスタム資源名が表示されます。

- リソースグループ内での資源の制限値と割り当て量

```
$ pjstat --limit --rscunit rscunit1 --rscgrp grp1
 System Resource Information:
 RSCUNIT: rscunit1
 RSCGRP : grp1
 USER: user1
  LIMIT-NAME                        LIMIT         ALLOC
  rg-accept                     unlimited             5 ← (1)
  rg-accept-allsubjob           unlimited             0 ← (2)
  rg-accept-bulksubjob          unlimited             0 ← (3)
  rg-accept-stepsubjob          unlimited             0 ← (4)
  rg-run-job                    unlimited             3 ← (5)
  rg-run-bulksubjob             unlimited             0 ← (6)
  rg-use-node                   unlimited          1011 ← (7)
  rg-interact-accept            unlimited             0 ← (8)
  rg-interact-run-job           unlimited             0 ← (9)
  rg-interact-use-node          unlimited             0 ← (10)
  rg-use-core                   unlimited            30 ← (11)
  rg-interact-use-core          unlimited             0 ← (12)
  rg-custom-customresourcename             5          1 ← (13) (*)
  rg-interact-custom-customresourcename    3          1 ← (14) (*)
 GROUP: group1
  LIMIT-NAME                        LIMIT         ALLOC
  rg-accept                     unlimited             5
  rg-accept-allsubjob           unlimited             0
  rg-accept-bulksubjob          unlimited             0
  rg-accept-stepsubjob          unlimited             0
  rg-run-job                    unlimited             3
  rg-run-bulksubjob             unlimited             0
  rg-use-node                   unlimited          1011
  rg-interact-accept            unlimited             0
  rg-interact-run-job           unlimited             0
  rg-interact-use-node          unlimited             0
  rg-use-core                   unlimited            30
  rg-interact-use-core          unlimited             0
 ALL:
  LIMIT-NAME                        LIMIT         ALLOC
  rg-accept                     unlimited             5
  rg-accept-allsubjob           unlimited             0
  rg-accept-bulksubjob          unlimited             0
  rg-accept-stepsubjob          unlimited             0
```

```
  rg-run-job                  unlimited            3
  rg-run-bulksubjob           unlimited            0
  rg-use-node                 unlimited         1011
  rg-interact-accept          unlimited            0
  rg-interact-run-job         unlimited            0
  rg-interact-use-node        unlimited            0
  rg-use-core                 unlimited           30
  rg-interact-use-core        unlimited            0
```

(1)バッチジョブの同時受付け数
(2) バルクジョブとステップジョブのサブジョブ、および通常ジョブ(バッチジョブ)の同時受付制限数
(3) バルクジョブのサブジョブの同時受付制限数
(4) ステップジョブのサブジョブの同時受付制限数
(5) バッチジョブの同時実行数
(6) バルクジョブのサブジョブの同時実行制限数
(7) バッチジョブのノードの同時使用数
(8) 会話型ジョブの同時受付け数
(9) 会話型ジョブの同時実行数
(10) 会話型ジョブのノードの同時使用数
(11) バッチジョブのCPUコアの同時使用数
(12) 会話型ジョブのCPUコアの同時使用数
(13) カスタム資源の同時使用制限数 (*)
(14) 会話型ジョブのカスタム資源の同時使用制限数 (*)
(*) rg-custom-*customresourcename* および rg-interact-custom-*customresourcename* の *customresourcename* は、実際には、定義されたカスタム資源名が表示されます。

pjstat コマンドの --limit オプションで表示される項目と内容の意味は以下のとおりです。

項目	説明
LIMIT-NAME	制限値の名称
RSCUNIT	リソースユニット名
GROUP	OSにおけるグループ名
LIMIT	制限値
ALLOC	現在の割り当て値

2.2.3 資源の確認

システムにおける資源の利用状況は pjshowrsc コマンドで確認します。

ジョブが実行されるリソースユニットの資源の利用状況は --rscunit (または --ru) オプションおよびその下の階層を表示する -E オプションで表示できます。

参考

--rscunit オプションについては、ショートオプション -- ru を用意しています。どちらの形式でも指定できます。

以降、リソースユニットの指定するオプションは--rscunit (--ru) オプションと表記します。

```
$ pjshowrsc --rscunit -E
[ CLST: clst ]
[ RSCUNIT: unit1 ]
    NODEID     CPU                    MEM
             TOTAL    FREE  ALLOC   TOTAL    FREE   ALLOC
0x01010010       8       0      8    14Gi       0    14Gi
0x01010011       8       0      8    14Gi       0    14Gi
0x01010012       8       0      8    14Gi       0    14Gi
```

```
0x01010013      8       0       8     14Gi       0     14Gi
0x01010014      8       8       0     14Gi     14Gi       0
```

表示される項目の意味は以下のとおりです。

項目		意味
CPU	TOTAL	ノードに搭載されている CPU コア数
	FREE	ジョブに割り当てられていない CPU コア数
	ALLOC	ジョブに割り当て済みの CPU コア数
MEM	TOTAL	ノードに搭載されているメモリ量 (バイト)
	FREE	ジョブに割り当てられていないメモリ量 (バイト)
	ALLOC	ジョブに割り当て済みのメモリ量 (バイト)

--custom-resource オプションを指定すると、カスタム資源の利用状況を表示できます。

```
$ pjshowrsc --rscunit unit1 --custom-resource
[ CLST: clst ]
RSCUNIT          NODE
                 TOTAL   FREE  ALLOC
unit1               96     96      0

CUSTOM RESOURCE                              ←リソースユニット単位のカスタム資源の利用状況
RSCNAME                                TOTAL       FREE      ALLOC
customrscname1                       1000000     300000     700000
customrscname2/type                unlimited unlimited          2

CUSTOM RESOURCE(PER NODE)                    ← ノード単位のカスタム資源の利用状況
RSCNAME                                TOTAL       FREE      ALLOC      NODE
customrscname3                            10         10          0        10
```

(*1) *customrscname1* および *customrscname3* には、定義されているカスタム資源名が表示されます。また、種別で定義されているカスタム資源 *customrscname2/type* には、カスタム資源名/種別 が表示されます。
(*2) ノード単位のカスタム資源の情報 (CUSTOM RESOURCE(PER NODE)) には、リソースユニット内の計算ノードのカスタム資源の総和が表示されます。

参考

--custom-resource オプションについては、ショートオプション -C を用意しています。どちらの形式でも指定できます。

表示される項目の意味は以下のとおりです。

項目	意味
RSCNAME	カスタム資源名 種別で定義されているカスタム資源の場合は、カスタム資源名/種別 *customrscname/type* の形式で表示されます。
TOTAL	カスタム資源の総数 種別で定義されているカスタム資源の場合は、unlimited が表示されます。
FREE	ジョブに割り当てられていていないカスタム資源の総数 種別で定義されているカスタム資源の場合は、unlimited が表示されます。
ALLOC	ジョブに割り当て済みのカスタム資源の総数 ("2.3.2.4 カスタム資源の指定" を参照) 種別で定義されているカスタム資源については、カスタム資源の数に 1 が指定されたものとして計算します。
NODE	カスタム資源が定義されたノードの数 ノード単位のカスタム資源にだけ出力されます。

pjshowrsc コマンドでは、ユーザーに権限のあるリソースユニットが表示されますが、ユーザーに対するデフォルトのリソースグループは、pjstat コマンドの --rsc オプションで確認してください("2.2.1 リソースユニット、リソースグループの確認" 参照)。

リソースユニット内のリソースグループについて、資源の利用状況を確認するには、--rscgrp (または --rg) オプションを指定してください。

参考

--rscgrp オプションについては、ショートオプション --rg を用意しています。どちらの形式でも指定できます。
以降、リソースグループを指定するオプションは、--rscgrp (--rg) オプションと表記します。

- リソースグループのごとのサマリ表示
  --rscgrp (--rg) オプションを引数なしで指定すると、すべてのリソースグループについて表示します。

```
$ pjshowrsc --rscunit unit1 --rscgrp
[ CLST: clst ]
[ RSCUNIT: unit1 ]
RSCGRP          NODE
                TOTAL   FREE  ALLOC
group1             36     24     12
group2             36     24     12
group3             36     24     12
```

- 特定のリソースグループについてのサマリ表示
  特定のリソースグループについて確認したい場合は、--rscgrp オプションの引数にリソースグループ名を指定します。

```
$ pjshowrsc --rscunit unit1 --rscgrp group2
[ CLST: clst ]
[ RSCUNIT: unit1 ]
RSCGRP          NODE
                TOTAL   FREE  ALLOC
group2             36     24     12
```

- 指定したリソースグループの詳細表示
  -l オプションを指定すると、指定したリソースグループの資源について詳細を表示します。

```
$ pjshowrsc --rscunit unit1 --rscgrp group2 -l
[ CLST: clst ]
[ RSCUNIT: unit1 ]
[ RSCGRP: group2 ]
     RSC    TOTAL    FREE   ALLOC
    node       36      33       3
     cpu     1152    1056      96
     mem     941Gi   856Gi     85Gi
```

- リソースグループ内の資源の一覧表示
  -E オプションを指定すると、指定したリソースグループ内の資源について、ノードごとに一覧表示します。

```
$ pjshowrsc --rscunit unit1 --rscgrp group1 -E
[ CLST: clst ]
[ RSCUNIT: unit1 ]
[ RSCGRP: group1 ]
    NODEID      CPU                    MEM
              TOTAL    FREE  ALLOC   TOTAL    FREE   ALLOC
0xFF030001       32      32      0    29Gi    29Gi       0
0xFF030002       32      32      0    29Gi    29Gi       0
0xFF030003       32      32      0    29Gi    29Gi       0
...
```

・ リソースグループ内の資源の詳細表示
-v オプションを指定すると、指定したリソースグループ内の資源について、ノードごとに詳細な情報を表示します。

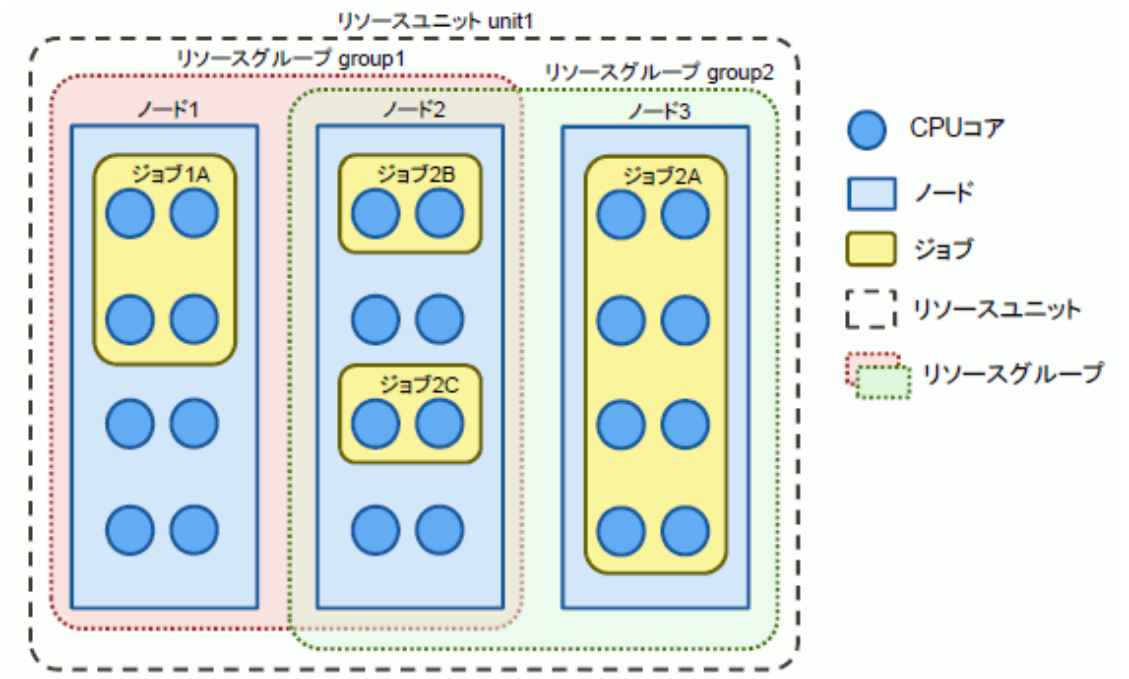
```
$ pjshowrsc --rscunit unit1 --rscgrp group1 -v
[ CLST: clst ]
[ RSCUNIT: unit1 ]
[ RSCGRP: group1 ]
[ NODE: 0xFF030001 ]
     RSC    TOTAL    FREE   ALLOC
     cpu       32      32       0
     mem     29Gi    29Gi       0

[ NODE: 0xFF030002 ]
     RSC    TOTAL    FREE   ALLOC
     cpu       32      32       0
     mem     29Gi    29Gi       0

[ NODE: 0xFF030003 ]
     RSC    TOTAL    FREE   ALLOC
     cpu       32      32       0
     mem     29Gi    29Gi       0
...
```

--rscgrp (--rg) オプションの指定時に表示される項目の意味は以下のとおりです。

項目	意味
RSC	資源の種類 (cpu: CPUコア数、mem: メモリ量)
TOTAL	リソースグループの最大資源量 (注)
FREE	--exclusive オプションの指定有無によって、表示される値が違います。 ・ --exclusive オプションが指定されない場合 当該リソースグループで使用可能な空き資源量が表示されます。 ・ --exclusive オプションが指定された場合 空き資源量は表示されず、"-" が表示されます。 --rscgrpで指定したリソースグループが、資源をほかのリソースグループと共有している場合があるため、自リソースグループの空き資源量は計算できません。
ALLOC	--exclusive オプションの指定有無によって、表示される値が違います。 ・ --exclusive オプションが指定されない場合 当該リソースグループおよび資源を共有するほかのリソースグループが使用する資源量 ・ --exclusive オプションが指定された場合 当該リソースグループだけが使用している資源量

(注)
FXサーバのリソースグループでは、管理者がリソースグループに割り当てる資源をノード数または割合で指定する方法に定義している場合は、リソースユニット内すべての資源が表示されます。ただし、リソースユニット内に、実行モードポリシーを simplex に設定しているリソースグループが存在する場合は、このリソースグループの資源は除かれて表示されます。
リソースグループの資源の定義内容は、管理者に確認してください。

--exclusive オプションは、リソースグループが資源をほかのリソースグループと共有している場合に、当該リソースグループだけが使用している資源量を確認したい場合に使用するオプションです。
例として、以下のような資源の利用状況の場合に、--exclusive オプションの指定有無による表示の違いを説明します。



リソースユニット unit1	ノード当たり8コア、メモリ30GiBの 3 ノード (ノード1、2、3) で構成される。
リソースグループ group1	ノード1、ノード2 で構成される。ノード2はリソースグループ group2 と共有する。
リソースグループ group2	ノード2、ノード3 で構成される。ノード2はリソースグループ group1 と共有する。
ジョブ1A	リソースグループ group1 で実行される。 ノード1のCPUコアを4つ、メモリを20GiB使用している。
ジョブ2A	リソースグループ group2 で実行される。 ノード3のCPUコアを8つ、メモリを20GiB使用している。
ジョブ2B	リソースグループ group2 で実行される。 ノード2のCPUコアを2つ、メモリを10GiB使用している。
ジョブ2C	リソースグループ group2 で実行される。 ノード2のCPUコアを2つ、メモリを10GiB使用している。

このとき、pjshowrsc コマンドで各リソースグループの資源の状態を表示させると、以下のようになります。

- --exclusive オプションを指定しない場合
  リソースグループ group1 の項目 ALLOC には、資源を共有するリソースグループ group2 で使用されている資源量が含まれます。

```
$ pjshowrsc --rscunit unit1 --rscgrp -l
[ CLST: clst000 ]
[ RSCUNIT: unit1 ]
[ RSCGRP: group1 ]
     RSC    TOTAL    FREE   ALLOC
    node        2       0       2
     cpu       16       8       8
     mem      60Gi     20Gi    40Gi
 [ RSCGRP: group2 ]
     RSC    TOTAL    FREE   ALLOC
    node        2       0       2
     cpu       16       4      12
     mem      60Gi     20Gi    40Gi
```

- --exclusive オプションを指定した場合
  項目 ALLOC には、各リソースグループで使用されている資源量だけが表示されます。また、--exclusive オプションを指定した場合、項目 FREE には "-" が表示されます。

```
$ pjshowrsc --rscunit unit1 --rscgrp -l --exclusive
[ CLST: clst ]
[ RSCUNIT: unit1 ]
[ RSCGRP: group1 ]
     RSC    TOTAL    FREE   ALLOC
    node        2       -       1
     cpu       16       -       4
     mem      60Gi       -    20Gi
[ RSCGRP: group2 ]
     RSC    TOTAL    FREE   ALLOC
    node        2       -       2
     cpu       16       -      12
     mem      60Gi       -    40Gi
```

2.3 ジョブの投入

ここでは、ジョブの投入方法について説明します。

2.3.1 ジョブの基本的な投入方法

ジョブの投入は、pjsub コマンドを使用します。

以下は、ジョブスクリプト job.sh を投入する最も簡単な例です。

```
$ pjsub job.sh
```

ジョブの投入時には、ジョブに割り当てる資源量(ノード数、メモリ量など)をpjsubコマンドのオプションで指定できます。以下は、ジョブに割り当てるノード数または仮想ノード数を指定して投入する例です。

```
[ジョブに8つのノードを割り当てる例]
$ pjsub -L "node=8" job.sh

[ジョブに8つの仮想ノードを割り当てる例]
$ pjsub -L "vnode=8" job.sh
```

指定できる資源量は、ジョブACL機能で許されている範囲で指定できます。資源量を指定しない場合は、ジョブACL機能で設定されている値が適用されます。値の確認方法は"2.2.2 制限情報の確認"を参照してください。ジョブの投入時に指定できるオプションについては、"2.3.2 ジョブ投入時のオプション" 以降の説明およびpjsubコマンドのmanマニュアルを参照してください。

注意

- システムによっては、1つのクラスタ内に、ジョブの運用方針や資源量、計算ノードの機種が異なる複数のリソースユニット、リソースグループが存在する場合があります。
  このようなシステムでは、ユーザーは必要に応じてジョブを投入するリソースユニット、リソースグループを指定してください("2.3.2.1 資源の指定" を参照)。
  ジョブが投入されるデフォルトのリソースユニット、リソースグループは、pjacl コマンドで確認できます("2.2.1 リソースユニット、リソースグループの確認" を参照)。
- pjsub コマンドは、PRIMERGYサーバに対し、FXサーバ向けのオプションを指定した場合は無視します。また、FXサーバ対し、PRIMERGYサーバ向けのオプションを指定した場合はエラーになります。

参考

pjsub コマンドのオプションは、コマンドラインではなく、ジョブスクリプトの中に記述できます。ただし、pjsub コマンドのオプションは、ジョブスクリプト内での指定よりもコマンドラインでの指定が優先されます。

ジョブスクリプトを指定しない場合は、pjsub コマンドの標準入力からジョブの内容が読み込まれます。

```
$ pjsub
#!/bin/sh                          ← ユーザーが pjsub の標準入力に対し、ジョブの内容を入力する
...
<Ctrl+d>                           ← 標準入力を <Ctrl+d> キーで閉じる
```

ジョブが正常に受け付けられた場合、以下のメッセージが表示されます。

```
[INFO] PJM 0000 pjsub Job jobid submitted.
```

jobid は、投入したジョブに対して設定されるジョブ ID です。

また、ジョブに対して「ジョブ名」が付けられ、デフォルトではジョブスクリプト名が設定されます。変更したい場合は、ジョブ投入時にpjsubコマンドの-Nまたは--nameオプションで指定できます。

2.3.2 ジョブ投入時のオプション

ここでは、ジョブに割り当てる資源やジョブの動作を指定するための pjsub コマンドのオプションについて説明します。

2.3.2.1 資源の指定

pjsub コマンドの -L オプションまたは --rsc-list オプションで、ジョブに割り当てる資源を指定できます。

```
{-L | --rsc-list } 項目=値
```

指定できる項目と値は以下です。

表2.6 資源の指定書式

書式	説明
FXサーバ: node=*shape*[:strict][:*ノード割り当て方法*] PRIMERGYサーバ: node=*N*	ジョブに割り当てるノード数やノード形状、およびノードの割り当て方法 (ノード割り当てジョブ用) 詳細は、"2.3.2.2 ノード資源の指定" を参照してください。
vnode=*num*[*CPUコア数*・*メモリ量*]	ジョブに割り当てる仮想ノード数、CPU コア数およびメモリ量 (仮想ノード割り当てジョブ用) 詳細は、"2.3.2.2 ノード資源の指定" を参照してください。 なお、FXサーバでは仮想ノードの割り当てはサポートしていません。
elapse=*limit* FXサーバに対しては以下の書式も指定できます。 elapse=*min_limit*- elapse=*min_limit*-*max_limit*	ジョブの経過時間制限値 (実行可能時間) ジョブの経過時間の上限を *limit* で指定します。 FXサーバを割り当てるジョブに対しては、経過時間の上限を最小値*min_limit*と最大値*max_limit*の範囲でも指定できます。 ジョブの経過時間が最小値 *min_limit* に達しても、ノードに空きがあれば、経過時間が最大値 *max_limit* に達するまではジョブの実行を継続します。 なお、最小値 *min_limit* には1秒以上、最大値 *max_limit* には2秒以上の値を指定してください。 経過時間制限値の指定の詳細は、"2.3.2.3 ジョブの経過時間制限値の指定" を参照してください。
node-mem=*limit*	1つのノードにおけるメモリ使用量の上限 (ノード割り当てジョブ用) 最小値は1MiB(Mi=2^{20})です。

書式	説明
rscunit=*name* ru=*name*	ジョブを投入するリソースユニット名 *name* は、63文字以内の文字列です。
rscgrp=*name* rg=*name*	ジョブを投入するリソースグループ名 *name* は、63文字以内の文字列です。
proc-core=*limit*	1つのジョブにおけるプロセスごとの最大コアファイルサイズ
proc-cpu=*limit*	1つのジョブにおけるプロセスごとの最大 CPU 時間 最小値は 1 秒です。
proc-crproc=*limit*	プロセスを実行しているノードおよび実ユーザーIDで生成できる最大プロセス数
proc-data=*limit*	1つのジョブにおけるプロセスごとの最大データセグメントサイズ
proc-lockm=*limit*	1つのジョブにおけるプロセスごとの最大ロックメモリサイズ
proc-msgq=*limit*	プロセスを実行しているノードおよび実ユーザーIDで確保できる最大POSIXメッセージキューサイズ
proc-openfd=*limit*	1つのジョブにおけるプロセスごとの最大ファイルディスクリプタ数
proc-psig=*limit*	プロセスを実行しているノードおよび実ユーザーIDでの最大ペンディングシグナル数
proc-filesz=*limit*	1つのジョブにおけるプロセスごとの最大ファイルサイズ
proc-stack=*limit*	1つのジョブにおけるプロセスごとの最大スタックサイズ
proc-vmem=*limit*	1つのジョブにおけるプロセスごとの最大仮想メモリサイズ
CustomResourceName=*Value*	カスタム資源の要求量または要求種別 *CustomResourceName* はカスタム資源名、*Value* はカスタム資源の数または種別です。 "=*Value*" は省略できます。 詳細は、"2.3.2.4 カスタム資源の指定" を参照してください。

指定する値 *limit* は、以下の表現形式で指定してください。

表2.7 資源量の表現形式

量	表現形式
時間	秒、分:秒、または 時:分:秒 の形式、または 時h分m秒s の形式で指定できます。h、m、s は大文字も使えます。 指定できる時間は特に断りがなければ 1 秒から 2147483647 秒までです。無制限にする場合は unlimited を指定します。 例: elapse=100 (100秒) elapse=1:40 (1分40秒) elapse=6:10:30 (6時間10分30秒) elapse=10h30m50s (10時間30分50秒) elapse=300m (300分) elapse=unlimited (制限なし)
メモリ量 (バイト)	2のべき乗を単位として表現します。 2のべき乗を示す単位は、Ki(キビ=2^{10})、Mi(メビ=2^{20})、Gi(ギビ=2^{30})、Ti(テビ=2^{40})、Pi(ペビ=2^{50})で表現し、省略した場合は Mi が指定されたものとします。 数値と2のべき乗を示す単位の間に空白は挿入してはいけません。 指定できる値は特に断りがなければ0Byteから2147483647MiBまでです。無制限にする場合は unlimited を指定します。 例: node-mem=256Mi (256MiB = 256*2^{20}Byte) node-mem=256 (同上)

量	表現形式
	node-mem=10Gi (10GiB = 10*2^{30}Byte) node-mem=unlimited (制限なし)
ディスク容量 ファイルサイズ (バイト)	10のべき乗を単位として表現します。 10のべき乗を示す単位は、K(キロ=10^3)、M(メガ=10^6)、G(ギガ=10^9)、T(テラ=10^{12})、P(ペタ=10^{15})で表現し、省略した場合はMが指定されたものとします。数値と10のべき乗を示す単位の間に空白は挿入してはいけません。 指定できる値は特に断りがなければ0Byteから2147483647MBまでです。無制限にする場合は unlimited を指定します。 例: node-quota=10M (10MB = 10*10^6Byte) node-quota=10 (同上) node-quota=10G (10GB = 10*10^9Byte) node-quota=unlimited (制限なし)
数値	上記以外の資源量は、数値だけを指定します。 指定できる値は特に断りがなければ 0 から 2147483647 までです。無制限にする場合は unlimited を指定します。

これらの値は、ジョブ ACL 機能で定義されている上限値を超える、または、下限値を下回ることはできません。これらのオプションが指定されない場合は、ジョブ ACL 機能で定義されているデフォルト値が適用されます。ジョブ ACL 機能で定義されている値を確認するには、"2.2.2 制限情報の確認" を参照してください。

ジョブが使用する資源量が、設定されている上限値を超えた場合のジョブの動作については、"2.3.2.5 資源量の上限を超えたジョブの動作について" を参照してください。

以下にジョブ資源の指定例を示します。

- ジョブにノードを 64 個割り当てる例。

```
$ pjsub -L "node=64" job.sh
```

  ジョブは、指定された数のノードで実行されます。指定されたノードの数を超えて使用することはできません。例えば、複数のノードを必要とする MPI プログラムが割り当てられたノードの数以上を要求した場合、MPI プログラムがエラーで終了します。

- ジョブの実行可能時間を86400秒に設定する例。

```
$ pjsub -L "elapse=86400" job.sh
```

  ジョブ実行中に、指定したジョブ実行可能時間を超えた場合、ジョブは強制終了させられます。

- ジョブのノードごとの使用メモリの上限を256MiBに設定する例。

```
$ pjsub -L "node-mem=256Mi" job.sh
```

  ジョブは、ノードごとの使用メモリ上限の中で実行されます。ノード単位の使用メモリ制限値を超える要求はできません。

ここまでに説明したオプションを組み合わせた例を以下に示します。

この例では、MPI プログラム prog を、割り当てノード数 (2つ)、実行可能時間(86400秒)、ノード単位の使用メモリの上限(256MiB)の制限値で投入します。また、制限値の指定は pjsub コマンドの引数ではなく、ジョブスクリプト job.sh 内に記述しています。

```
$ cat job.sh
#!/bin/sh
#PJM -L "node=2"                          ← 割り当てノード数 2
#PJM -L "elapse=86400"                    ← 実行可能時間 86400秒
#PJM -L "node-mem=256Mi"                  ← ノード単位の使用メモリの上限 256MiB
...
mpiexec ./prog                            ← MPIプログラム prog を mpiexec コマンドで実行
$ pjsub job.sh
```

参照

上記例にある mpiexec コマンドの使用方法については、Development Studioのマニュアル「MPI 使用手引書」を参照してください。

ジョブ投入時にはジョブに割り当てる資源だけでなく、ジョブが入出力をする第1階層ストレージに関するパラメーターも指定できます。詳細は、"2.3.3 LLIOに関するパラメーターの指定 [FX]" を参照してください。

2.3.2.2 ノード資源の指定

ジョブに割り当てるノード資源は、pjsub コマンドの -L (--rsc-list) オプションの node または vnode パラメーターで指定します。ノード単位で割り当てる場合は node パラメーター、仮想ノード単位で割り当てる場合は vnode パラメーターを使います。

注意

node および vnode パラメーターのどちらも指定しない場合、ノードまたは仮想ノードのどちらを割り当てるかはジョブACL機能の設定内容に従います。pjacl コマンドで、項目 default alloc granularity の設定値を確認してください。

[ノード単位で割り当てる場合]

ノード資源をノード単位で割り当てるための node パラメーターの指定方法は、計算ノードの機種によって異なります。

- FXサーバを割り当てる場合
  ジョブ投入時にユーザーは、そのノード形状(1、2、または3次元形状)を指定します。ノード形状については、"1.6 ノード資源の割り当て" を参照してください。
  ノード形状は、pjsub コマンドの -L または --rsc-list オプションの node パラメーターで指定し、実行中に存在するプロセスすべてを格納できる大きさでなくてはいけません。

```
{-L | --rsc-list} "node=shape[:strict|:strict-io][:torus|:mesh|:noncont]"
```

表2.8 ノード形状の指定書式 [FX]

書式	説明
shape[:strict|:strict-io] または	*shape* は、ノード形状の次元に応じて、*X*、*X*x*Y*、または *X*x*Y*x*Z* のように各軸方向の大きさを指定します。 ノード形状が3次元の場合、*X*x*Y*x*Z*:strict のように strict 指定ができます。これは、通常、空きノード空間に指定したノード形状が収まるように自動的に回転させて割り当てますが、strict 指定の場合は、回転させずに割り当てを試みます。2次元形状で同様なことをする場合は、3次元形状で表現して strict 指定を使ってください。 node パラメーターを指定しなかった場合、割り当てるノード形状はジョブACL機能で定義されているデフォルト値に従います。pjacl コマンドで、"pjsub option parameters" の項目 (node=) の default 値を確認してください。 計算ノードがFXサーバの場合は、strict-io 指定を使用すると、Tofu座標におけるI/Oノードの位置が常に同じになるように指定できます。strict-io 指定の詳細については、"2.3.2.11 I/Oノードを意識した資源の割り当て [FX]" を参照してください。
:torus または :mesh または :noncont	ジョブがノード割り当てジョブの場合、Tofu座標内でのノードの配置方法(トーラスモード、メッシュモード、離散割り当て)を指定できます。 torus は、Tofu 単位(12ノード)で計算機資源をジョブに割り当てるトーラスモードを意味します。 mesh は、ノード単位で計算機資源をジョブに割り当てるメッシュモードを意味します。 noncont は、ノード単位で計算機資源をジョブに割り当てる離散割り当てを意味します。 ノードの各割り当て方法については、"1.6.3.1 FXサーバのノード単位での割り当て"を参照してください。 省略時は、ジョブACL機能で定義されているデフォルト値に従います。pjacl コマンドで、"defines" の項目 default allocation mode の値を確認してください。

以下は、MPIプログラム prog_A を実行するジョブに対し、3次元のノード形状 (X 軸 4、Y 軸 3、Z 軸 2ノード) をトーラスモードで割り当てる例です。

```
$ cat job.sh                                ← MPIプログラム prog_A を実行するジョブスクリプト job.sh
#!/bin/sh
...
mpiexec ./prog_A
$ pjsub -L "node=4x3x2:torus" job.sh        ← ノード形状 4x3x2 とトーラスモードを指定してジョブ投入
```

注意

- 指定するノード形状は、システムの最大形状を超えて指定できません。最大形状を超えた場合は、ジョブの投入がエラーになります。
  最大形状を知る方法については "2.2.1 リソースユニット、リソースグループの確認" を参照してください。
  なお、ノード形状がそのままでは最大形状をはみ出してしまいますが、回転させることで収まるような場合は、自動的にノード形状の回転が行われます。ユーザーは回転させた形を指定する必要はありません。
- ノードの割り当て方法 ":mesh"、":torus"、":noncont" が指定できるのは、それぞれ、ジョブACL機能の項目 execute pjsub(torus)、execute pjsub(mesh)、execute pjsub(noncont) が "enable"の場合です。
- ノードを割り当てるパラメーター node=*shape* と、仮想ノードを割り当てるパラメーター vnode=*num* (後述) を同時に指定した場合は、ジョブ投入がエラーになります。

- PRIMERGYサーバを割り当てる場合
  ジョブに、PRIMERGYサーバを割り当てる場合、ノード数を指定します。

```
{-L | --rsc-list} "node=num"
```

表2.9 ノード数の指定書式 [PG]

書式	説明
node=*num*	割り当てるノード数を指定します。 node パラメーターを指定しなかった場合、ノード数はジョブACL機能で定義されているデフォルト値に従います。pjaclコマンドで、"pjsub option parameters" の項目 (node=) の default 値を確認してください。 同時にvnodeパラメーターを指定した場合、仮想ノード単位で割り当てられます。

以下は、ジョブに 8 ノードを割り当てる場合の例です。

```
$ pjsub -L "node=8" job.sh
```

[仮想ノード単位で割り当てる場合]

ノード資源を仮想ノード単位で割り当てるための vnode パラメーターは、以下のように指定します。

```
{-L | --rsc-list} "vnode=[num][,vnode-core=num][,{core-mem=size|vnode-mem=size}][,node=num]"
または
{-L | --rsc-list} "vnode=[num][([core=num][:core-mem=size|mem=size])][,node=num]"
```

表2.10 仮想ノードの指定書式

書式	説明
vnode=[*num*]	割り当てる仮想ノードの数です。 *num* を省略した場合、または vnode パラメーターを指定しなかった場合、割り当てる仮想ノード数は、ジョブACL機能で定義されているデフォルト値に従います。pjaclコマンドで、"pjsub option parameters" の項目 (vnode=) の default 値を確認してください。
vnode-core=*num* core=*num*	仮想ノード当たりの CPU コア数です。 省略した場合は、ジョブACL機能の設定に従います。pjaclコマンドで、"pjsub option parameter" の項目 (vnode-core=) の default 値を確認してください。

書式	説明
vnode-mem=*size* mem=*size*	仮想ノード当たりのメモリ量です。パラメーター core-mem とは排他です。 省略した場合は、ジョブACL機能の設定に従います。pjacl コマンドで、"pjsub option parameter" の項目 (vnode-mem=) の default 値を確認してください。
core-mem=*size*	CPU コア当たりのメモリ量です。パラメーター mem および vnode-mem とは排他です。 省略した場合は、仮想ノード当たりの CPU コア数および仮想ノード当たりのメモリ量から決まります。
node=*num* [PG]	PRIMERGYサーバにおける仮想ノード配置ポリシーが UNPACK (-P "vn-policy=unpack") の場合だけ使用でき、ジョブに割り当てるノード数を指定します。UNPACK 以外の場合、このパラメーターは無視されます。 このパラメーターを指定した場合は、必ず指定した数のノードを必要とします。割り当てのイメージは "2.3.5.1 仮想ノード配置ポリシー" の "図2.15 UNPACKとノード数指定 (1)" と "図2.16 UNPACKとノード数指定 (2)" を参照してください。 このパラメーターを省略した場合は、利用できるすべてのノードを割り当てます。

メモリ量の表現形式は "表2.7 資源量の表現形式" を参照してください。

以下は、仮想ノード数が 1、仮想ノード内の CPU コア数が 5、仮想ノード当たりのメモリ量が 30MiBの場合の例です。

```
$ pjsub -L "vnode=1,vnode-core=5,vnode-mem=30Mi" job.sh
または
$ pjsub -L "vnode=1(core=5:mem=30Mi)" job.sh
```

注意

- パラメーター core-mem、vnode-mem、または mem に unlimited を指定することは、そのジョブに対してはジョブ管理機能によるメモリ量制限は行われないことを意味します。このため、ジョブは OS が許す限りメモリを使用できますが、ジョブ実行中にノード内でほかのジョブが動作した場合は、メモリが不足する可能性があります。
  メモリ量制限を unlimited に指定したジョブを投入する場合は、ほかのジョブによってメモリ獲得が失敗しないようにするために、以下のどちらかの方法でジョブがノードを専有してください。
  - ノード割り当て指定 (node パラメーター指定) でジョブを投入する。
  - PRIMERGYサーバで仮想ノード割り当て指定 (vnode パラメーター指定) の場合には、ジョブの実行モードポリシーに SIMPLEX を選択する (-P exec-policy=simplex)。
- FXサーバでは仮想ノード割り当てジョブはサポートしていません。

参照

PRIMERGYサーバでは、上記で説明した割り当てるノード資源の量以外に、ノードの割り当てについての考え方(ノード選択ポリシー)が指定できます。詳細は "2.3.5 ノード選択ポリシーの指定 [PG]" を参照してください。

2.3.2.3 ジョブの経過時間制限値の指定

ジョブを投入するとき、ジョブの経過時間制限値 (実行可能時間) を以下の形式で指定できます。

- 経過時間制限値を指定する。

  ```
  {-L | --rsc-list} elapse=limit
  ```

  ジョブの経過時間が *limit* に達するとジョブは強制終了させられます。

- 経過時間制限値を範囲で指定する。[FX]

  ```
  {-L | --rsc-list} elapse=min_limit-                (*) min_limit の後ろにハイフンが必要
  {-L | --rsc-list} elapse=min_limit-max_limit
  ```

経過時間が *min_limit* に達しても、経過時間が最長で *max_limit* に達するまで実行を継続できます。ただし、経過時間が *min_limit* を超えて実行しているとき、後続のジョブのためにノードが必要になった場合、またはデッドラインスケジューリングによってノードが予約されている期間になると、経過時間が *max_limit* 未満でもジョブは強制終了させられます。

実行を継続できる最長時間を制限しない場合は、*max_limit*にunlimitedを指定してください(例: elapse=600-unlimited)。*max_limit*を指定しない場合(例: elapse=600-)は、*max_limit*にはジョブACL機能で設定されている値が適用されます。

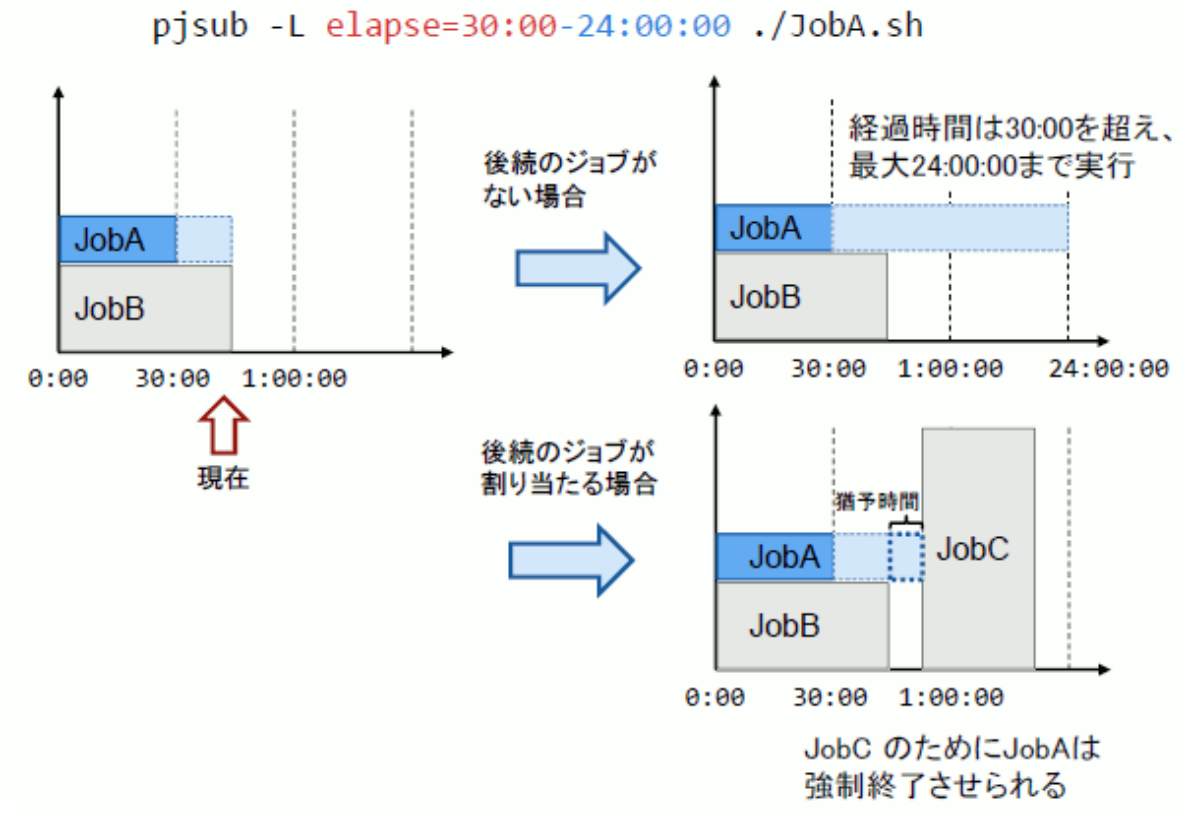
参考

デッドラインスケジューリングとは、保守などのために特定の期間、ノードにジョブを割り当てないようにする機能のことです。これは管理者が設定します。

注意

ジョブの経過時間にプロローグ・エピローグ処理の実行時間を加算するように管理者が設定している場合は、経過時間が *min_limit* に達する前にジョブが強制終了させられる場合があります。

図2.3 ジョブの経過時間制限値の範囲指定



ジョブが経過時間制限値に達して強制終了させられる場合、ジョブが終了処理をできるように猶予期間を設け、事前にシグナルが送信されます。必要に応じて、ジョブ内でシグナルを捕捉して処理をしてください。

- 経過時間の上限値で指定した場合 (elapse=*limit*)

  強制終了させられる10秒前に、ジョブに対してシグナル SIGXCPU が送信されます。

- 経過時間の範囲で指定した場合 (elapse=*min_limit*-*max_limit*)

  ― 経過時間が *max_limit* に達する前に強制終了させられる場合は、その10秒前に、ジョブに対してシグナル SIGTERM が送信されます。なお、この指定形式における猶予期間の長さは管理者が変更できます。

ー 経過時間が*max_limit*を超えた場合は、強制終了させられる10秒前に、ジョブに対してシグナル SIGXCPU が送信されます。

注意

- 経過時間制限値の指定形式は、管理者がジョブACL機能によって利用を制限している場合があります。pjacl コマンドで、項目execute のパラメーター"pjsub(fixed elapsed time)"と"pjsub(adaptive elapsed time)"を確認してください。
- ジョブ投入時にelapseパラメーターを指定しない場合、経過時間制限値はジョブACL機能によって、elapse=*limit*か elapse=*min_limit*-*max_limit* のどちらかの形式が適用されます。pjacl コマンドで、項目 "default elapsed time mode" を確認してください。

2.3.2.4 カスタム資源の指定

ジョブにカスタム資源を割り当てる場合は、pjsub コマンドの -L オプションを使い、以下の書式で指定します。

```
{-L | --rsc-list} CustomResourceName[=Value]
```

CustomResourceName: カスタム資源名
Value: 割り当てるカスタム資源の数または割り当てるカスタム資源の種別

1つのジョブで指定できるカスタム資源の数は、64個までです。

Value の指定は、省略できます。

なお、指定するカスタム資源のタイプ (数値で定義されているカスタム資源、種別で定義されているカスタム資源) とパラメーターの指定方法の違いによって、以下のように、採用されるカスタム資源の要求量または要求種別が異なります。

表2.11 パラメーターの指定方法ごとの採用されるカスタム資源の要求量または要求種別

指定するカスタム資源のタイプ	パラメーターの指定方法		
	CustomResourceName=Value を指定	*CustomResourceName* だけ指定 (*Value* の指定を省略)	指定なし
数値で定義されているカスタム資源を指定する場合	*Value* で指定した資源数が要求量になります。	ジョブ ACL 機能で定義されているデフォルト値が要求量になります。 デフォルト値が定義されていない場合は、下限値が要求量になります。	ジョブ ACL 機能で定義されているデフォルト値が要求量になります。 デフォルト値が定義されていない場合は、当該カスタム資源を利用しないものとして扱われます。
種別で定義されているカスタム資源を指定する場合	*Value* で指定した資源の種別が要求種別になります。	ジョブ ACL 機能で定義されているデフォルトの種別が要求種別になります。 デフォルト値が定義されていない場合は、ジョブを投入してもエラーになります。	ジョブ ACL 機能で定義されているデフォルトの種別が要求種別になります。 デフォルト値が定義されていない場合は、当該カスタム資源を利用しないものとして扱われます。

参照

ジョブ ACL 機能で定義されているカスタム資源の数や種別を確認するには、"2.2.2 制限情報の確認" を参照してください。

"1.8 カスタム資源" で説明したとおり、カスタム資源は「リソースユニットまたはリソースグループ単位のカスタム資源」または「ノード単位のカスタム資源」で定義されています。このため、投入するジョブの種別によって、同じカスタム資源の数を指定していても、割り当てられるカスタム資源の数は異なります。

以下は、リソースユニットまたはリソースグループ単位のカスタム資源を指定した場合の例です。

1. 通常ジョブ
   指定された数だけ、カスタム資源を割り当てます。

```
$ pjsub -L "node=2, customrscname=3" ./job.sh
```

   customrscname: カスタム資源名

   この例の割り当てカスタム資源数は、3 になります。

2. ステップジョブ
   通常ジョブと比べて、ステップジョブはサブジョブ数が 2 以上になることがあるため、その分、割り当てるカスタム資源数が増えます。

```
$ pjsub --step -L "node=2, customrscname=3" ./job1.sh,./job2.sh
```

   この例の割り当てカスタム資源数は、6 (カスタム資源 3個 × 2 サブジョブ) になります。

3. バルクジョブ
   ステップジョブと同様に、サブジョブの数だけ、割り当てるカスタム資源数が増えます。

```
$ pjsub --bulk --sparam 0-2 -L "node=2,customrscname=3" ./job1.sh
```

   この例の割り当てカスタム資源数は、9 (カスタム資源 3個 × 3 サブジョブ) になります。

ノード単位のカスタム資源を指定した場合は、ノード資源の割り当て方法を指定してください。ノード資源の割り当てのため、ノードごと、または仮想ノードごとにカスタム資源の要求量を指定します。それぞれ、パラメーターに node または vnode を指定します。パラメーターの指定方法の詳細は、"2.3.2.2 ノード資源の指定" を参照してください。
以下は、ノード単位のカスタム資源の場合の例です。

1. ノード割り当てジョブ (node を指定)
   node に指定されたカスタム資源の数×ノードの数だけ、カスタム資源を割り当てます。

```
$ pjsub -L "node=2, customrscname=1" ./job.sh
```

   この例の割り当てカスタム資源数は、2 (ノードごと 1 × 2 ノード) になります。

2. 仮想ノード割り当てジョブ (vnode を指定)
   vnode に指定されたカスタム資源の数×仮想ノードの数だけ、カスタム資源を割り当てます。

```
$ pjsub -L "vnode=1,vnode-core=3, customrscname=3" ./job.sh
```

   または

```
$ pjsub -L "vnode=1(core=3; customrscname=3)" ./job.sh
```

   この例の割り当てカスタム資源数は、3 (仮想ノードごと 3 × 1 仮想ノード) になります。

2.3.2.5 資源量の上限を超えたジョブの動作について

ジョブが実行中に使用する資源量が設定されている上限を超えた場合、ジョブの動作は以下のようになります。

表2.12 資源量の上限を超えたジョブの動作

資源	動作
elapse	・ elapse=*limit* の形式で指定した場合 ジョブ内のすべてのプロセスにシグナル SIGXCPU が送信されます。 その後、10 秒経過してもジョブのプロセスが存在する場合、それらにはシグナル SIGKILL が送信され、強制終了させられます。この10秒の待ち合わせは、各プロセスが後処理を実行できるようにするためです。 このため、必要に応じて、ジョブスクリプトやジョブスクリプトから実行されるプロセスそれぞれでシグナル SIGXCPU を捕捉して処理をしてください。 ・ elapse=*min_limit*-*max_limit*の形式で指定した場合 経過時間の最小値 *min_limit* を超えて実行しているジョブが、後続のジョブが実行を開始するために経過時間の最大値 *max_limit* に達する前に強制終了させられる場合、一定時間前

資源	動作
	にジョブ内のすべてのプロセスに SIGTERM が送信されます。デフォルトでは10秒前ですが、この時間は管理者が変更できます。 経過時間の最大値 *max_limit* に達した場合は、上述の elapse=*limit* の場合と同じで、ジョブ内のすべてのプロセスにシグナルSIGXCPU が送信されます。
node-mem vnode-mem mem core-mem	メモリ獲得要求をしたプロセスは OS により強制終了させられます。 その後の動作は、プログラムやジョブスクリプトの作りに依存します。 なお、プロセスの終了だけではなく、ジョブごと強制終了させるように管理者が設定している場合があります。また、この設定になっている場合、ジョブが原因でOSのメモリが枯渇してOOM Killerが動作すると、そのノードを使用しているジョブのうち、原因になったジョブと同じユーザーIDのジョブはすべて強制終了させられます。ご利用のシステムにおける設定については管理者にお問い合わせください。
proc-*	proc-core などの上限値は、ジョブ内の個々のプロセスに対する、OS (システムコール setrlimit()) による資源制限値です。 それぞれ、システムコール setrlimit() で指定できる以下の資源に対応します。 proc-core : RLIMIT_CORE proc-cpu : RLIMIT_CPU proc-crproc : RLIMIT_NPROC proc-data : RLIMIT_DATA proc-lockm : RLIMIT_MEMLOCK proc-msgq : RLIMIT_MSGQUEUE proc-openfd : RLIMIT_NOFILE proc-psig : RLIMIT_SIGPENDING proc-filesz : RLIMIT_FSIZE proc-stack : RLIMIT_STACK proc-vmem : RLIMIT_AS 指定した上限値は、これら資源のソフトリミットおよびハードリミットとして設定されます。プロセスが上限値を超えた場合の動作は OS の仕様に従います。詳細は Linux の man マニュアル setrlimit(2) を参照してください。なお、FXサーバでの動作は Linux の仕様に準じます。

注意

- ジョブの資源制限値は、管理者が設定するプロローグ・エピローグ処理 ("1.12 プロローグ・エピローグ機能" 参照) の実行にも適用されます。
  プロローグ・エピローグ処理はジョブの一部として実行されるため、ユーザーが作成したジョブスクリプト、プログラムが制限値を超えていなくても、プロローグ・エピローグ処理で制限値を超える可能性があります。
  プロローグ・エピローグ処理が必要とする資源量については、管理者にお問い合わせください。
- ジョブ実行環境のKVMモード("2.3.8 ジョブの実行環境の指定"参照)でジョブを実行した場合、ジョブ運用ソフトウェアで設定されたメモリ資源量の超過は検出されません。仮想マシン内で引き起こされたメモリ超過時の動作は、使用した仮想マシンイメージファイルの設定に依存します。

2.3.2.6 ジョブ統計情報出力の指定

pjsub コマンドの -s または -S オプションで、ジョブの実行結果としてジョブ統計情報を出力できます。

```
{ -s | -S } [ --spath 出力先 ]
```

-S オプションは -s オプションよりも詳細な統計情報を出力します。両者の違いは"A.2 ジョブ統計情報の出力"やmanマニュアル pjstatsinfo(7)を参照してください。

参考

-s と -S オプションの代わりに、それぞれロングオプション --stats、--STATS も使用できます。

ジョブ統計情報は、ジョブ投入時のカレントディレクトリ配下のジョブ統計情報ファイル「*ジョブ名*.*ジョブID*.stats」に出力されます。この出力先を指定したい場合は、--spath オプションでファイル名を指定してください。

ファイル名には以下の表記を使用できます。

表2.13 ジョブ統計情報ファイル名の表記方法

書式	意味
%j	ジョブID
%J	サブジョブID
%b	バルク番号
%s	ステップ番号
%n	ジョブ名

また、ファイルではなく、メールによってジョブ統計情報を受け取りたい場合は、-m オプションを使用してください。

```
-m {s | S}
```

この場合の引数 s および S は、それぞれ -s および -S オプションと同じ意味になります。

2.3.2.7 ジョブの自動再実行についての指定

システムダウンなどでジョブの実行が中断された場合、自動的に再実行するかどうかをジョブ投入時に pjsub コマンドの --restart または --norestart オプションで指定できます。

表2.14 再実行可否の指定オプション

オプション	説明
--restart	ジョブが異常終了した場合、自動で再実行します。
--norestart	ジョブが異常終了しても、自動で再実行しません。

注意

- これらのオプションを指定しない場合の動作はシステムの設定で変更できます。ユーザーが利用しているシステムがどちらの動作になるかは、管理者にお問い合わせください。
- 会話型ジョブの自動再実行はできません。これらのオプションを指定した場合は無視されます。

2.3.2.8 実行開始時刻の指定

通常、ユーザーが投入したジョブは、空き資源の状況や優先度に従って最も早く実行できるように順番が決定されますが、ユーザーによる実行開始時刻の指定もできます。

実行開始時刻は pjsub コマンドの--at オプションを使い、以下の書式で指定します。

書式	説明
--at *YYYYMMDD*[*hhmm*]	*YYYY* は年、*MM* は月、*DD* は日を表します。 *hh* は時、*mm* は分を表します。*hhmm* を省略した場合は 0時0分が指定されたものとみなします。 また、秒単位では指定できません。

以下は、実行開始時刻を指定してジョブを投入する例です。

```
$ pjsub --at 201908011511 job.sh    ← 2019年8月1日15:11 に job.sh の実行を開始する
```

注意

- 実行開始時刻を指定したジョブは、資源に空きがあっても、指定時刻より前に実行されることはありません。また、資源の空き状況によっては、指定時刻より後に実行される場合があります。
- 会話型ジョブに対しては、実行開始時刻は指定できません。指定した場合は無視されます。

2.3.2.9 ジョブ優先度の指定

ユーザーは、自分が投入したジョブについてだけ、ジョブ間の実行優先度を設定できます。ジョブの優先度は pjsub コマンドの -p オプションを指定し、0 から 255 の間の整数を指定します。

```
$ pjsub -p 優先度 job.sh
```

優先度は 0 が最低、255 が最高となります。

参照

ユーザーの各ジョブの優先度は pjstat コマンドの -v オプションで確認できます。詳細は "2.4.1 ジョブの一覧表示" を参照してください。

参考

- 優先度が同じジョブがある場合は、投入順となります。
- -p オプションが指定されない場合の優先度は、管理者が設定している内容に従い決定されます。

2.3.2.10 バッチジョブの標準出力、標準エラー出力ファイルの指定

バッチジョブの標準出力、標準エラー出力は、ファイルに出力されます。
ファイル名は以下になります。詳細は "2.6.1 ジョブ実行結果の参照" を参照してください。

- 標準出力: *ジョブ名*.*ジョブID*.out
- 標準エラー出力: *ジョブ名*.*ジョブID*.err

このファイル名をユーザーが指定したい場合は、pjsub コマンドの -o や -e オプションを使用します。また、標準エラー出力を標準出力に向ける場合は -j オプションを指定します。

表2.15 バッチジョブの標準出力、標準エラー出力ファイルの指定

書式	説明
-o *filename*	ジョブの標準出力をファイル *filename* に出力します。
-e *filename*	ジョブの標準エラー出力をファイル *filename* に出力します。
-j	ジョブの標準エラー出力を標準出力に向けます。

参考

-o と -e オプションの代わりに、それぞれロングオプション --out、--err オプションも使用できます。

出力ファイル名の指定では、以下の表記を使用できます。

表記	意味
%j	ジョブID
%J	サブジョブID
%b	バルク番号
%s	ステップ番号
%n	ジョブ名

以下は、この表記を使用した出力ファイル名の指定例です。

```
$ pjsub -o '%j[%b].stdout' -e '%j[%b].stderr'
```

この例では、ジョブID 100 のバルクジョブで、バルク番号 10 のサブジョブの標準出力ファイル、標準エラー出力ファイルはファイル名 100[10].stdout と 100[10].stderr になります。

注意

- バルクジョブやステップジョブで、各サブジョブの出力先を同じファイルにした場合、ファイル内で各サブジョブの出力が混在します。このため、出力結果からサブジョブごとの出力を読み取れない可能性があります。
- 会話型ジョブでは、pjsub オプションの -o や -e オプションの指定はできません。
- FXサーバ上で実行するジョブ内でmpiexecコマンドを実行すると、その標準出力および標準エラー出力は、mpiexecコマンドのオプションで指定しなければジョブACL機能の項目mpiexec-*("2.2.2 制限情報の確認"参照)で定義されるファイルへ出力されます。これにより、ジョブの標準出力および標準エラー出力ファイルは、pjsubコマンドの-oや-eオプションで指定したファイルのほかにも作成されることがあります。
  mpiexecコマンドの標準出力および標準エラー出力については"2.3.6.9 mpiexecコマンドの標準出力/標準エラー出力 [FX]"やDevelopment Studioのマニュアル「MPI使用手引書」を参照してください。

2.3.2.11 I/Oノードを意識した資源の割り当て [FX]

ジョブの入出力処理をするI/Oノード(ストレージI/Oノードまたはグローバル I/Oノード)のTofu座標上の位置を意識して計算ノードを割り当てると、ジョブのI/O性能の変動の抑止や向上が見込めます。

I/Oノードを意識した割り当てには、以下があります。

- I/Oノードと計算ノードの距離を毎回同じにする割り当て

  この割り当て方法は、ジョブを実行するたびにI/O性能が変動することを抑える効果があります。

- I/Oノードをジョブに専有させる割り当て

  この割り当て方法は、1つのジョブの入出力処理のためにI/Oノードを専有させることでほかのジョブの入出力の影響を受けないようにする効果があります。

以下ではそれぞれについて説明します。

[I/Oノードと計算ノードの距離を毎回同じにする割り当て]

ジョブを実行する計算ノードとその入出力を処理するI/OノードのTofu座標上の距離(座標の差)はジョブのI/O性能に影響します。
ジョブ投入時にパラメーターstrict-ioを指定すると、形状の回転をせずにノードがTofu座標上で割り当てられます。また、割り当てるノードの位置は常にFXサーバの本体装置ラックの原点(ラック内の最小の座標)が始点になります。これにより、各計算ノードに対するI/Oノードの位置は毎回同じになり、ジョブを実行するたびにI/O性能が変動することを抑えられます。

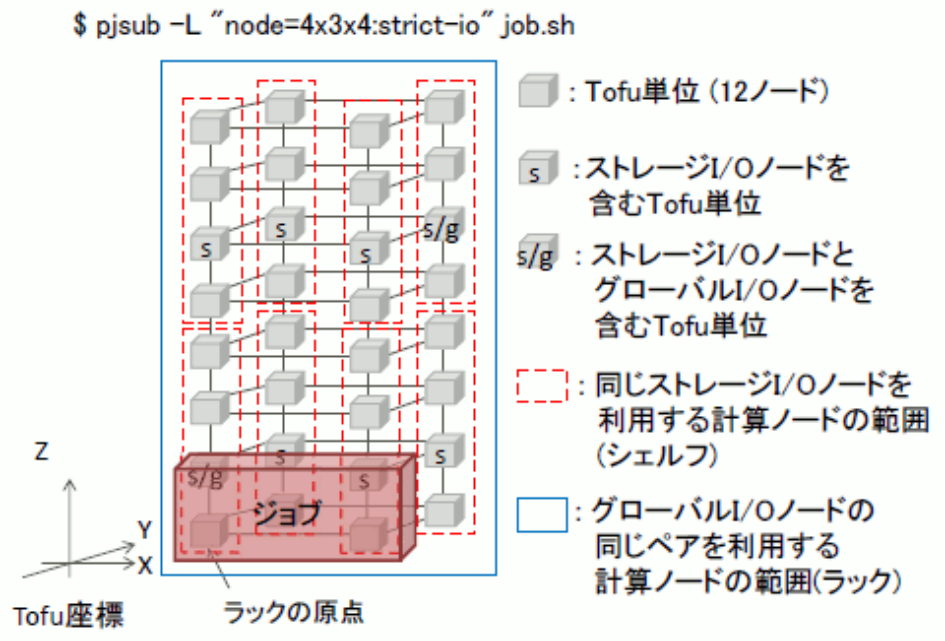
```
$ pjsub -L "node=N1xN2xN3:torus:strict-io" job.sh
```

注意

パラメーターstrict-ioは、torusモードかつ3次元形状のノードを割り当てるときだけ指定できます。
なお、以降の説明では、torusモードはジョブACL機能によってデフォルト値として設定されている前提とし、コマンドラインの例では省略します。

以下の図は、形状4x3x4のノードをパラメーターstrict-ioを指定して割り当てた例です。ノードの形状4x3x4は、6次元のTofu座標X,Y,Z,A,B,C)で表すと(X軸:2、A軸:2)x(Y軸:1、B軸:3)x(Z軸:2、C軸:2)になります。これは、Tofu座標におけるX、Y、Z軸の各辺の長さがそれぞれ2、1、2であるTofu単位の直方体に相当します。これがラックの原点を始点にして割り当てられます。

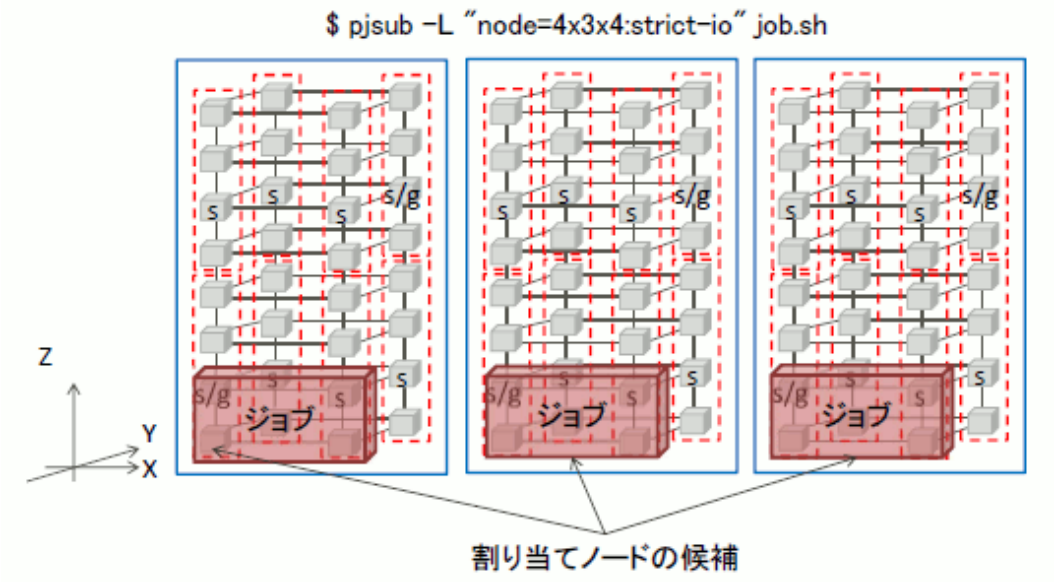
図2.4 パラメーターstrict-ioを指定したときのノードの割り当て



参考

例えばラックが3つあるシステムで、形状4×3×4のノードをパラメーターstrict-io付きでジョブに割り当てようとすると、以下の図のように配置する位置の候補が3箇所あります。どれを選択するかはジョブ運用ソフトウェアが決定します。毎回割り当てられるラックが異なったとしても、ラックの原点が始点となるため、各計算ノードに対するI/Oノードの位置は毎回同じになります。

図2.5 割り当てノードの候補が複数ある場合



[I/Oノードをジョブに専有させる割り当て]

I/O性能を最大限に引き出すには、I/Oノードの位置を考慮するだけではなく、投入するジョブにだけI/Oノードを専有させる必要もあります。このためには以下で説明するように、I/Oノードが入出力を担う範囲のノードを単位として割り当ててください。

- ストレージI/Oノードの専有

  ストレージI/Oノードを1つのジョブに専有させることで、FXサーバにおける第1階層ストレージ(LLIO)のI/O性能を最大限に引き出せます。これをI/O専有モードと呼び、I/O専有モードのジョブをI/O専有ジョブと呼びます。I/O専有モードでは、FXサーバのシェルフ単位(48ノード)でジョブにノードを割り当てることで、ストレージI/Oノードをそのジョブだけが専有でき、ほかのジョブの入出力の影響を受けません。
  I/O専有モードは、ジョブ投入時にパラメーターio-exclusiveを指定してください。

```
$ pjsub -L "node=shape:io-exclusive" job.sh
```

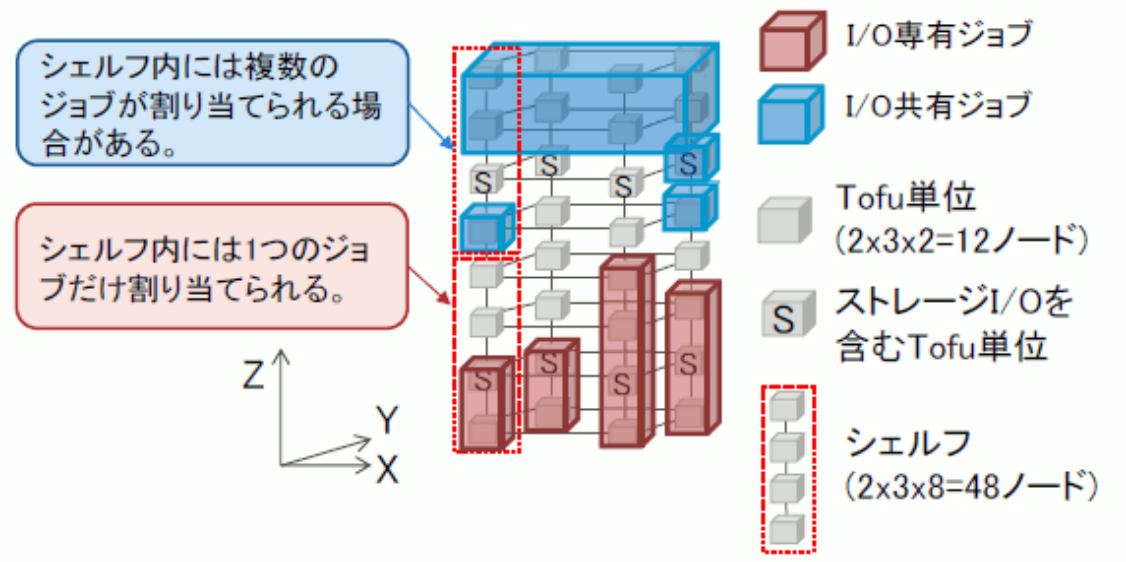
  I/O専有モードは、ノード割り当てモードがtorusモードの場合に指定でき、meshモードとnoncontモードでは指定できません。
  ストレージI/Oノードを専有しないモードをI/O共有モードと呼び、I/O共有モードのジョブをI/O共有ジョブと呼びます。I/O共有モードは、ジョブ投入時にパラメーターno-io-exclusiveを指定してください。

```
$ pjsub -L "node=shape:no-io-exclusive" job.sh
```

参考

I/O専有ジョブが割り当てられたシェルフには、ほかのジョブは割り当てられません。また、ほかのジョブが使用しているシェルフがI/O専有ジョブに割り当てられることはありません。
I/O共有ジョブが使用しているシェルフの空きノードには、別のI/O共有ジョブが割り当てられることがあります。

図2.6 I/O専有モードとI/O共有モード



上記の図では、I/O専有ジョブに割り当てられたノードの位置は複数ありますが、パラメーターio-exclusiveに加えて、パラメーターstrict-ioも指定すると、常にラックの原点を始点としたノードだけが割り当てられます。

- グローバルI/Oノードの専有

  グローバルI/Oノードを専有する場合は、ペアとなる2つのグローバルI/Oノードが入出力を担う範囲のノード(4x6x16、すなわちラック)を単位として割り当ててください。この場合は、ラック内のストレージI/Oノードも専有することになります。また、ラックの原点を始点として割り当てるためにパラメーターstrict-ioも指定してください。

```
$ pjsub -L "node=4x6x16:strict-io" job.sh
```

パラメーターstrict-io、io-exclusive、およびno-io-exclusiveを指定しない場合は、ジョブACL機能で設定されているデフォルト値が適用されます。また、これらのパラメーターは指定できないようにジョブACL機能によって管理者が設定している場合があります。

2.3.2.12 Tofuインターコネクトのリンクダウンに対する動作の指定 [FX]

FXサーバでジョブを実行中に、ハードウェアの故障などが原因でTofuインターコネクトのリンクダウンが発生すると、ジョブのプロセス間通信に影響がでますが、通信経路を動的に変更することで避けられる場合があります。
pjsubコマンドの--net-routeオプションでは、Tofuインターコネクトのリンクダウン時に通信経路を変更するかどうかを指定できます。

表2.16 pjsubコマンドの--net-routeオプション

オプション	説明
--net-route dynamic	Tofuインターコネクトがリンクダウンすると通信経路を変更し、ジョブの実行を継続します。
--net-route static	Tofuインターコネクトがリンクダウンしても通信経路は変更しません。ジョブは異常終了します。

--net-routeオプションは、FXサーバ上のノード専有ジョブかつMPIジョブ(Development StudioによるMPI処理系だけ)に対して指定できます。PRIMERGYサーバ上のジョブに対して指定した場合は無視されます。

オプションを指定しない場合は、ジョブACL機能の項目define net-routeの値に従います。また、--net-routeオプションはジョブACL機能の項目execute pjsub-net-routeがenableに設定されているときだけ指定できます。ジョブACL機能の設定内容の確認方法は"2.2.2 制限情報の確認"を参照してください。

--net-route dynamicオプションを指定した場合、Tofuインターコネクトのリンクダウンが発生するとジョブは以下のように動作します。

- Tofuライブラリによって通信経路が変更され、通信の再実行(再送)が発生します。
  そのあと、再送をしたことを示すメッセージがジョブの標準エラー出力に出力されます。
  このメッセージを出力する必要がない場合は、ジョブスクリプト内で 環境変数UTOFU_LINKDOWN_INFOに0を設定してください。
- 通信経路を変更してもジョブの実行が継続できないとTofuライブラリが判断した場合は、ジョブの標準エラー出力ファイルにエラーメッセージを出力し、ジョブを終了させます。このとき、ジョブの終了コード(PJMコード)は22になります。

出力されるメッセージについては、「ジョブ運用ソフトウェア コマンドリファレンス」の"ジョブ実行中に出力されるメッセージ"にある"Tofuインターコネクト"を参照してください。

注意

--net-route dynamicオプションを指定する場合、以下に注意してください。

- ジョブの実行性能が劣化する可能性があります。実行性能を重視するジョブに対して使用する場合は注意してください。
- Tofu インターコネクトのハードウェア機能として提供されるバリア通信機能は適用できません。詳細は、Development Studioの「MPI使用手引書」を参照してください。
- リンクダウンが発生した箇所によっては回避できる通信経路が存在せず、ジョブの実行を継続できない場合があります。

2.3.3 LLIOに関するパラメーターの指定 [FX]

ジョブ投入時に、LLIOに関するパラメーターをpjsub コマンドの--llioオプションで指定できます。

```
$ pjsub --llio param=arg,...
```

LLIOに関するパラメーターは、第1階層ストレージ内の領域のサイズや階層化ストレージの挙動を指定します。パラメーターを指定しない場合は、ジョブACL機能の設定に従います。

図2.7 階層化ストレージ

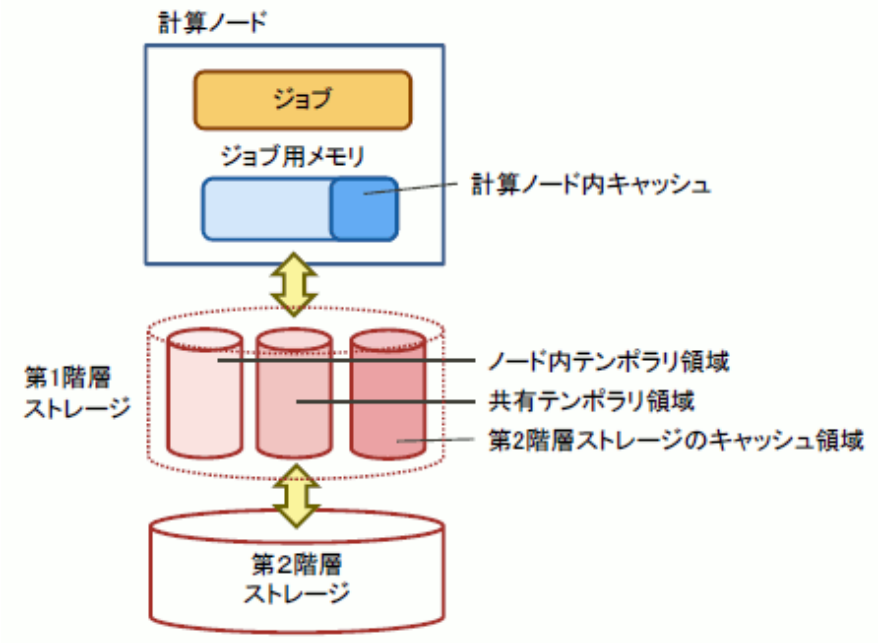


表2.17 第1階層ストレージ内の領域に関するパラメーター

パラメーター	説明
localtmp-size=*size*	第1階層ストレージ上のノード内テンポラリ領域のサイズ 詳細は "2.3.3.1 第1階層ストレージのサイズ" を参照してください。
sharedtmp-size=*size*	第1階層ストレージ上の共有テンポラリ領域のサイズ (注) 1つのジョブが利用できる共有テンポラリ領域のサイズは、*size*×*計算ノード数* です。 詳細は "2.3.3.1 第1階層ストレージのサイズ" を参照してください。
sio-read-cache={on|off}	第2階層ストレージから計算ノードへ読み込んだファイルを第1階層ストレージにキャッシュするか否かの動作

パラメーター	説明
	on: キャッシュします。 off: キャッシュしません。 詳細は "2.3.3.2 第2階層ストレージから読み込んだファイルのキャッシング" を参照してください。

第2階層ストレージのキャッシュの領域サイズは、ノード内テンポラリ領域と共有テンポラリ領域のサイズから決まります。詳細は"2.3.3.1 第1階層ストレージのサイズ"を参照してください。

表2.18 第1階層ストレージに対するストライピングに関するパラメーター

パラメーター	説明
stripe-count=*count*	第1階層ストレージにファイルを分散配置する際のファイル当たりのストライプ数 詳細は "2.3.3.3 ストライプ" を参照してください。
stripe-size=*size*	第1階層ストレージにファイルを分散配置する際のストライプサイズ 詳細は "2.3.3.3 ストライプ" を参照してください。

表2.19 第1階層ストレージおよび第2階層ストレージに対する入出力動作に関するパラメーター

パラメーター	説明
async-close={on|off}	第1階層ストレージおよび第2階層ストレージ上のファイルのクローズを非同期クローズにするか否かの動作 on: 非同期クローズ off: 同期クローズ on(非同期クローズ)を指定した場合、ファイルのクローズ時に書出し完了は保証されません。ただし、ジョブ終了時は書出し完了は保証されます。 off(同期クローズ)を指定した場合、ファイルのクローズ時に書出し完了は保証されます。 詳細は "2.3.3.4 非同期クローズ" を参照してください。
auto-readahead={on|off}	ジョブが複数回続けて第1階層ストレージまたは第2階層ストレージ上の連続した領域を読み込もうとした場合に、自動的に先読みをするか否かの動作 on: 先読みをします。 off: 先読みをしません。 詳細は "2.3.3.5 ファイルの自動先読み" を参照してください。

表2.20 計算ノード内キャッシュに関するパラメーター

パラメーター	説明
cn-cache-size=*size*	第1階層ストレージ上の計算ノード内キャッシュのサイズ 詳細は "2.3.3.6 計算ノード内キャッシュのサイズ" を参照してください。
cn-cached-write-size=*size*	第1階層ストレージへの書込み時にキャッシュするか否かのしきい値 第1階層ストレージへの書込みの際に、書込みサイズが指定した値以下の場合は、すぐにはストレージに書き出さず、一時的に計算ノード内キャッシュにキャッシュします。 小さなサイズについてはまとめて書き出すことで、第1階層ストレージへの転送回数を減らし、性能を向上させるためのパラメーターです。 詳細は "2.3.3.7 第1階層ストレージへの書込み時のキャッシュのしきい値" を参照してください。
cn-read-cache={on|off}	第1階層ストレージまたは第2階層ストレージから読み込んだファイルを計算ノード内キャッシュにキャッシュするか否かの動作 on: キャッシュします。 off: キャッシュしません。 詳細は "2.3.3.8 計算ノードで読み込んだファイルのキャッシング" を参照してください。

表2.21 階層化ストレージに対する入出力の統計情報に関するパラメーター

パラメーター	説明
perf[,perf-path=*path*]	LLIO性能情報をファイルに出力します。 出力先はジョブ投入時のカレントディレクトリ配下で、ファイル名はジョブACL機能で定義されている名前です。なお、パラメーターperf-pathで出力先のファイルを指定できます。 詳細は "2.3.3.9 LLIO性能情報の採取" を参照してください。
uncompleted-fileinfo-path=*path*	ジョブ終了時に、第2階層ストレージへの書出しが完了していないファイルの情報の出力先 詳細は "2.3.3.10 未書出しファイルの情報" を参照してください。

以降では、各パラメーターについて説明します。

2.3.3.1 第1階層ストレージのサイズ

第1階層ストレージには、ジョブが実行中に一時的かつ高速なストレージとして利用できるノード内テンポラリ領域と共有テンポラリ領域があります。
ノード内テンポラリ領域または共有テンポラリ領域を使う場合、そのサイズはジョブ投入時に指定できます。pjsub コマンドの --llio オプションのパラメーターlocaltmp-sizeで1ノード当たりのノード内テンポラリ領域のサイズを、パラメーター sharedtmp-size で共有テンポラリ領域のサイズ÷割当てノード数を指定します。サイズを指定しない場合は、ジョブACL機能で設定されているデフォルト値になります。

注意

パラメーターsharedtmp-sizeで指定する値は、ジョブが利用できる共有テンポラリ領域全体のサイズではありません。共有テンポラリ領域全体のサイズは、パラメーターsharedtmp-sizeで指定する値にジョブに割り当てるノード数を乗じた値になります。

```
$ pjsub --llio localtmp-size=size,sharedtmp-size=size job.sh
```

各領域のサイズに指定できる値の上限、下限、およびデフォルト値は、pjacl コマンドで表示される項目localtmp-size と sharedtmp-size で確認してください。

```
$ pjacl
...
pjsub option parameters
   ...
    (--llio)                          lower           upper           default
        (sharedtmp-size=)             0Mi             2147483647Mi    0Mi
        (localtmp-size=)              0Mi             2147483647Mi    0Mi
   ...
```

第1階層ストレージの領域は、FXサーバにおける入出力を担うストレージI/OノードのSSD(Solid State Drive)から確保されます。1つの計算ノードに割り当てられるSSDのサイズは、1つのストレージI/Oノードが入出力を担う計算ノードの数で等分した値になります。

1つの計算ノードに割り当てられる第1階層ストレージのサイズ
＝ ストレージI/OノードのSSDのサイズ / 1つのストレージI/Oノードが入出力を担う計算ノードの数

例えば、SSDのサイズを800GiB、1つのストレージI/Oノードが入出力を担う計算ノードの数を16とすると、1つの計算ノードに割り当てられる第1階層ストレージのサイズは50GiBになります。ストレージI/OノードのSSDのサイズおよびストレージI/Oが入出力を担う計算ノードの数はシステムによって異なりますので、管理者にお問い合わせください。

1つの計算ノードに割り当てられるSSDの領域のうち、ノード内テンポラリ領域と共有テンポラリ領域を除いた分が、1ノード当たりの第2階層ストレージのキャッシュになります。ただし、第2階層ストレージのキャッシュは最低でも1計算ノードあたり128MiB必要なので、上記の例では、ノード内テンポラリ領域と共有テンポラリ領域のサイズの合計は最大(50GiB-128MiB)まで指定できます。
1つのジョブが使用できる第2階層ストレージのキャッシュはこの値にノード数を乗じたサイズになります。

注意

- ノード内テンポラリ領域のサイズ(localtmp-size)を0にした場合、ノード内テンポラリ領域にファイルを作成できません。
  共有テンポラリ領域のサイズ(sharedtmp-size)を0にした場合、共有テンポラリ領域にファイルを作成できますが、データを格納できません。
- ノード内テンポラリ領域と共有テンポラリ領域は、同じストレージI/Oノードの高速なストレージ(SSD)上に確保されます。このため、それぞれのサイズが上限以下であっても、両者の合計がストレージのサイズを超える場合はジョブ投入時にエラーになります。エラーメッセージには合計値の上限サイズが表示されます。
- 領域に空きがなくなった場合は、write システムコールは ENOSPC でエラーになります。
  共有テンポラリ領域とノード内テンポラリ領域は、ファイルのデータ以外にファイルの管理データによっても消費されます。したがって、ファイルのデータの合計が、領域のサイズ未満でもENOSPCになることがあります。ファイルの管理データによって消費される容量の目安は以下のとおりです。

  ```
  300byte×ファイル数
  ```

- ジョブ投入時に指定した共有テンポラリ領域のサイズまで容量を消費していないにも関わらず、空き容量不足でエラーになることがあります。
  これは、ジョブが複数のストレージI/Oノードを利用し(*)、1つのストレージI/Oノードの利用可能容量を使い切ったことが原因と考えられます。上記現象が発生した場合、共有テンポラリ領域の容量を増やしてジョブを実行してください。
  (*) ジョブが複数のストレージI/Oノードを利用していることは、LLIO性能情報("2.3.3.9 LLIO性能情報の採取"参照)に複数のストレージI/Oノードの情報があるかどうかで確認できます。
- ノード内テンポラリ領域と共有テンポラリ領域のサイズは1MiBを最小単位として、メモリ量と同様の2のべき乗(Ki、Mi、Giなど)で指定してください。
  一般的にディスク装置のサイズは10のべき乗(K、M、Gなど)で表されますが、上述の例では説明を簡単にするために、SSDのサイズを2のべき乗で表しています。サイズを計算する際には、単位の違いに注意してください。

2.3.3.2 第2階層ストレージから読み込んだファイルのキャッシング

ジョブが第2階層ストレージ上のファイルを読む場合に、第1階層ストレージ上にキャッシュすると、ジョブが同じファイルを複数回読み込む場合に性能の向上が期待できます。

第1階層ストレージ上にキャッシュするかどうかは、pjsub コマンドの --llio オプションのパラメーター sio-read-cache で指定できます。

```
$ pjsub --llio sio-read-cache={on|off} job.sh
```

sio-read-cache=on を指定すると、第2階層ストレージから読み込んだファイルを第1階層ストレージ上にキャッシュします。sio-read-cache=off を指定すると読み込んだファイルはキャッシュしません。

パラメーターを指定しない場合は、ジョブACL機能の設定に従います。pjacl コマンドで表示される項目 default llio-sio-read-cache で確認してください。

```
$ pjacl
...
defines
   ...
    default llio-sio-read-cache off
```

2.3.3.3 ストライプ

第1階層ストレージのストライプを利用すると、以下のメリットがあります。

- ストレージI/Oノードに接続された1つのストレージデバイス(SSD)の物理的な容量を超えるサイズのファイルを作成できます。
- 1つのファイルを複数の第1階層ストレージに分散して格納することで、ファイルアクセスの帯域幅が向上します。

ストライプを利用する場合、pjsub コマンドの --llio オプションのパラメーター stripe-count でストライプ数を、パラメーター stripe-size でストライプサイズを指定します。

```
$ pjsub --llio stripe-count=count,stripe-size=size job.sh
```

指定可能な値は、ジョブACL機能で設定されている範囲です。pjacl コマンドで表示される項目 stripe-count および stripe-size で確認してください。

```
$ pjacl
...
pjsub option parameters
   ...
    (--llio)                              lower           upper           default
   ...
        (stripe-count=)                   1               2147483647      2147483647
        (stripe-size=)                    64Ki            4194240Ki       2048Ki
```

注意

ストライプサイズは64KiBを最小単位として、メモリ量と同様の2のべき乗(Ki、Mi、Giなど)で指定してください。

第1階層ストレージのストライプサイズが第2階層ストレージのストライプサイズの倍数でない場合、第2階層ストレージの性能を引き出せない場合があります。第2階層ストレージのストライプサイズとストライプカウントは、lfsコマンドのgetstripeサブコマンドで取得でき、setstripeサブコマンドで変更できます。ただし、どちらの操作もジョブ内で実行してください。指定できるストライプサイズとストライプカウントの上限値、下限値は、第2階層ストレージ(FEFS)の仕様を確認してください。

以下に第2階層ストレージのストライプサイズを変更するジョブスクリプトの例を示します。また、この例では最後にlfsコマンドのgetstripeサブコマンドで第2階層ストレージのストライプに関する情報を取得しています。

```
#!/bin/bash
#
#PJM --llio sharedtmp-size=5Mi,localtmp-size=10Gi
#PJM --llio stripe-size=1Mi
lfs setstripe -S 1m /gfs1/user1/data          ← 第2階層ストレージのストライプサイズの設定
mpiexec /gfs1/user1/a.out                     ← プログラム a.out を実行
lfs getstripe /gfs1/user1/data                ← 第2階層ストレージのストライプ情報の取得
```

lfsコマンドのgetstripeサブコマンドで取得した第2階層ストレージのストライプ情報は、ジョブの標準出力に、以下のように出力されます。

```
/gfs1/user1/data
stripe_count:   1 stripe_size:    1048576 stripe_offset:  0
/gfs1/user1/data/file1
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  0
        obdidx           objid           objid           group
             0           19538         0x4c52               0
```

lfsコマンドのgetstripeサブコマンドおよびsetstripeサブコマンドの詳細は、マニュアル「LLIOユーザーズガイド」の付録"リファレンス"を参照してください。

2.3.3.4 非同期クローズ

ファイルのクローズ処理を非同期クローズにすると、プログラムはファイルをクローズするたびに書出しの完了を待たされることがなく、次の処理に進めます。

pjsub コマンドの --llio オプションのパラメーター async-close で、ファイルのクローズを非同期クローズにするかどうかを指定します。

async-close=on を指定するとクローズ処理は非同期クローズになります。async-close=off を指定するとクローズ処理は同期クローズになります。

```
$ pjsub --llio async-close={on|off} job.sh
```

パラメーターを指定しない場合は、ジョブACL機能の設定に従います。pjacl コマンドで表示される項目 default llio-async-close で確認してください。

```
$ pjacl
...
defines
   ...
    default llio-async-close     on
```

注意

非同期クローズ (async-close=on) を指定しても、ジョブは書出しの完了を待ち合わせてから終了します。

この書出しを待ち合わせる時間は、ジョブの経過時間に含まれます。

2.3.3.5 ファイルの自動先読み

階層化ストレージ上のファイルの読込み時に、計算ノード内キャッシュへ先読みすることで、連続した領域を読み込むプログラムの高速化が見込めます。

pjsub コマンドの--llio オプションのパラメーター auto-readahead で、プログラムが連続した領域を読み込んだ場合、自動的に計算ノード内キャッシュへ先読みするかどうかを指定します。
auto-readahead=on で自動先読みし、auto-readahead=off で自動先読みしません。

```
$ pjsub --llio auto-readahead={on|off} job.sh
```

注意

パラメーター cn-read-cache=off を指定している場合、自動先読みは無効になります。

パラメーターを指定しない場合は、ジョブACL機能の設定に従います。pjacl コマンドで表示される項目 default llio-auto-readahead で確認してください。

```
$ pjacl
...
defines
   ...
    default llio-auto-readahead on
```

2.3.3.6 計算ノード内キャッシュのサイズ

ジョブに割り当てられたメモリに余裕がある場合は、空いたメモリを計算ノード内キャッシュとして使用することで高速な入出力が期待できます。

pjsub コマンドのパラメーター cn-cache-size で、計算ノード内キャッシュの最大サイズを指定します。

```
$ pjsub --llio cn-cache-size=size job.sh
```

指定できる値の上限、下限、およびデフォルト値は、pjacl コマンドで表示される項目 cn-cache-size で確認してください。

```
$ pjacl
...
pjsub option parameters
   ...
    (--llio)                          lower           upper           default
   ...
        (cn-cache-size=)              4Mi             2147483647Mi    128Mi
```

注意

- 計算ノード内キャッシュは、指定したサイズを上限としてジョブに割り当てられたメモリから確保されます。したがって、計算ノード内キャッシュを必要以上に大きく指定すると、ジョブ内のプロセスが必要とするジョブ用メモリの獲得に失敗することがありますので注意してください。

- ファイルの読込みでは、後述のパラメーター cn-read-cache が on に設定されている場合に計算ノード内キャッシュにキャッシュされます。
  ファイルの書込みでは、後述のパラメーターcn-cached-write-sizeに設定されているしきい値以下のサイズの書込みは、計算ノード内キャッシュにキャッシュされます。
- 指定するサイズは、メモリ量と同様の2のべき乗 (Ki、Mi、Giなど) で指定してください。

2.3.3.7 第1階層ストレージへの書込み時のキャッシュのしきい値

ジョブがファイルへ書き込むときに、小さなサイズのデータは一時的に計算ノード内キャッシュにキャッシュし、あとでまとめて第1階層ストレージへ転送することで性能面での効果が見込めます。

pjsubコマンドの --llio オプションのパラメーター cn-cached-write-size で、このサイズのしきい値を指定できます。しきい値より小さいデータの書込みは、一時的に計算ノード内キャッシュにキャッシュします。

```
$ pjsub --llio cn-cached-write-size=size job.sh
```

注意

- 計算ノード内キャッシュのサイズ(--llio cn-cache-size)よりも大きい値を指定した場合、ジョブの投入はエラーになります。
- 指定するサイズは4KiBを最小単位として、メモリ量と同様の2のべき乗 (Ki、Mi、Giなど) で指定してください。

指定できるしきい値の上限、下限、およびデフォルト値は、pjacl コマンドで表示される項目 cn-cached-write-size で確認してください。

```
$ pjacl
...
pjsub option parameters
   ...
    (--llio)                          lower           upper           default
   ...
        (cn-cached-write-size=)       0Mi             2147483647Mi    4Mi
```

2.3.3.8 計算ノードで読み込んだファイルのキャッシング

第1階層ストレージまたは第2階層ストレージから計算ノードへ読み込んだファイルを計算ノード内キャッシュにキャッシュすることで、繰り返しファイルを読み込む場合の性能向上が見込めます。

pjsub コマンドの--llio オプションのパラメーター cn-read-cache で、階層化ストレージから読み込んだファイルを計算ノード内キャッシュにキャッシュするかどうかを指定します。
cn-read-cache=onを指定すると、計算ノードで読み込んだファイルをキャッシュします。cn-read-cache=off を指定するとキャッシュしません。

```
$ pjsub --llio cn-read-cache={on|off}  job.sh
```

パラメーターを指定しない場合は、ジョブACL機能の設定に従います。pjacl コマンドで表示される項目 default llio-cn-read-cache で確認してください。

```
$ pjacl
...
defines
   ...
    default llio-cn-read-cache     on
```

2.3.3.9 LLIO性能情報の採取

第1階層ストレージへのアクセス状況を後から分析し、ジョブのチューニングをするために、LLIO 性能情報を採取できます。

LLIO性能情報は、pjsubコマンドの--llio オプションのパラメーターperfを指定するとジョブ投入時のカレントディレクトリ配下のファイルに出力されます。

```
$ pjsub --llio perf job.sh
```

ファイル名はジョブACL機能で定義されている名前です。パラメーターperfを指定しない場合にLLIO性能情報が出力されるかどうかは、ジョブACL機能の定義に従います。これらのジョブACL機能の定義については、pjaclコマンドで表示される項目default llio-perf-pathとdefault llio-perfを確認してください。

```
$ pjacl
...
defines
  ...
   default llio-perf           off
   default llio-perf-path      %n.%J.llio_perf
```

ファイル名の書式では以下のメタ文字が使用されます。

表2.22 ファイル名のメタ文字

メタ文字	説明
%j	ジョブIDに展開
%J	サブジョブIDに展開
%b	バルク番号に展開
%s	ステップ番号に展開
%n	ジョブ名に展開(最大63文字) ジョブが標準入力から投入された場合は、"STDIN"になります。ジョブ名が半角文字の数字で始まる場合は、"*ジョブ名*"の先頭に文字"J"が付加されます。
%o	標準出力ファイルパスに展開。会話型ジョブの場合には、"./%n.%J.out" に展開。
%e	標準エラー出力ファイルパスに展開。会話型ジョブの場合には、"./%n.%J.err" に展開。

出力ファイルのパスは、パラメーターperf-pathで指定できます。

```
$ pjsub --llio perf,perf-path=path job.sh
```

パラメーターperf-pathで指定するファイル名には"表2.22 ファイル名のメタ文字"で示したメタ文字が使用できます。

ファイルに出力されるLLIO性能情報は以下のようになります。

```
I/O     2ndLayerCache   NodeTotal                       Count           Amount(Byte)        Time(us)
                Write(Cached I/O)       Sum             1               15                  236
                                        [1,4Ki)         1               15                  236
                                        [4Ki,1Mi)       0               0                   0
                                        [1Mi,4Mi)       0               0                   0
                                        [4Mi+           0               0                   0
                Read(Cached I/O)        Sum             12              1827                12987
                                        [1,4Ki)         12              1827                12987
                                        ...
I/O     2ndLayerCache   ComputeNode<->CommBuf           Count           Amount(Byte)        Time(us)
                Write(Cached I/O)       Sum             -               -                   68
...
Resource        LocalTmp        CacheUsage              Amount(Byte)
                                        MaximalUsedSpace                0

SIO Information
Node ID : 0x01010002
I/O     2ndLayerCache   NodeTotal                       Count           Amount(Byte)        Time(us)
                Write(Cached I/O)       Sum             1               15                  236
                                        [1,4Ki)         1               15                  236
                                        [4Ki,1Mi)       0               0                   0
                                        [1Mi,4Mi)       0               0                   0
                                        [4Mi+           0               0                   0
                Read(Cached I/O)        Sum             12              1827                12987
                                        [1,4Ki)         12              1827                12987
                                        ...
```

```
I/O     2ndLayerCache   ComputeNode<->CommBuf          Count               Amount(Byte)        Time(us)
                Write(Cached I/O)       Sum            -                   -                   68
                ...
...
```

参照

出力されるLLIO統計情報の詳細についてはマニュアル「LLIOユーザーズガイド」の付録"統計情報の出力項目"にある"LLIO性能情報の出力項目"を参照してください。

参考

- pjsubコマンドの-s、-Sオプション、およびpjstatコマンドの-s、-Sオプションで出力されるジョブ統計情報の中にも、第1階層ストレージに関する統計情報が含まれます。詳細はmanマニュアルpjstatsinfo(7)を参照してください。
- LLIOが提供するライブラリ関数getlliostat()を使うアプリケーションを作成することで、ジョブの中で第1階層ストレージの利用に関する統計情報を取得できます。関数getlliostat()の使用方法と取得できる統計情報の詳細については、マニュアル「LLIOユーザーズガイド」の付録"統計情報の出力項目"にある"計算ノード統計情報の採取方法と出力項目"を参照してください。

注意

LLIO 性能情報の採取はジョブが RUNOUT 状態の時に実施しています。

LLIO性能情報は、ジョブを実行したノード数が多くなるほどその採取に時間がかかり、これにともなってジョブがRUNOUT状態となっている時間も増えていきます。

1 万ノード程度であれば数分内で完了しますが、2 万ノードで 30 分、5 万ノードで 2 時間程度かかる場合があります。

2.3.3.10 未書出しファイルの情報

ジョブ終了時に第1階層ストレージ上から第2階層ストレージへの書出しが完了できない場合があります。例えば、書出し処理中にジョブの経過時間制限に達した場合は、中断されてしまいます。この場合、未書出しファイルの情報を分析することで、ジョブの経過時間制限値を調整するための目安にできます。

pjsub コマンドの --llio オプションのパラメーター uncompleted-fileinfo-path で、未書出しファイルの情報を出力できます。

```
$ pjsub --llio uncompleted-fileinfo-path=path job.sh
```

path には未書出しファイルの情報の出力先を指定します。出力先には以下のメタ文字が使用できます。

表2.23 指定可能なメタ文字

メタ文字	説明
%j	ジョブIDに展開
%J	サブジョブIDに展開
%b	バルク番号に展開
%s	ステップ番号に展開
%n	ジョブ名に展開(最大63文字) ジョブが標準入力から投入された場合は、"STDIN"になります。ジョブ名が半角文字の数字で始まる場合は、"ジョブ名"の先頭に文字"J"が付加されます。
%o	標準出力ファイルパスに展開。会話型ジョブの場合には、"./%n.%J.out" に展開。
%e	標準エラー出力ファイルパスに展開。会話型ジョブの場合には、"./%n.%J.err" に展開。

パラメーターを指定しない場合は、ジョブACL機能の設定に従います。pjacl コマンドで表示される項目 default llio-uncompleted-fileinfo-path で確認してください。

```
$ pjacl
...
defines
   ...
    default llio-uncompleted-fileinfo-path  %e
```

未書出しファイルの情報は以下のように出力されます。

```
<file transfer error information>  # タイトル行
<summary>                          # サマリ情報のタイトル行
total num: 3                       # 転送失敗ファイル数
total size: 128000000              # 転送失敗サイズ[B]
error: E1,E3,E8                    # エラー情報
<detail>                           # 詳細情報のタイトル行
E1 /gfs1/userA/outfile1            # エラー番号 ファイルパス
E1 /gfs1/userA/dir/outfile2
E3,E8 /gfs1/userA/dir/outfile3
```

表2.24 出力される情報

種別	出力項目	出力内容
サマリ <summary>	total num	未書出しファイルの数 情報が取得できないファイルがあった場合は、数値の後ろに "+" が付きます。 未書出しファイルの数が1000を超えた場合は、"1000*" または "1000*+" と出力されます。
	total size	未書出しファイルの合計サイズ 情報が取得できないファイルがあった場合は、数値の後ろに "+" が付きます。
	error	エラー情報 (書出しが完了しなかった理由) E1:第2階層ストレージのQUOTA超過 E2:書出し中断 E3:書出しタイムアウト E4:書出しエラー E5: デバイスの空き容量不足 E8:そのほかのエラー
詳細情報 <detail>	第1フィールド	エラー情報 (書出しが完了しなかった理由) E1:第2階層ストレージのQUOTA超過 E2:書出し中断 E3:書出しタイムアウト E4:書出しエラー E5: デバイスの空き容量不足 E8:そのほかのエラー
	第2フィールド	ファイルのパス

未書出しファイルの数が1000を超えた場合は、詳細情報には以下のメッセージのみが出力されます。

```
The detailed information is not output because the number of uncompleted files exceeds the upper limit (1000).
```

2.3.4 ジョブの種類ごとの投入方法

通常ジョブ以外のジョブは、その種類によって固有の指定が必要です。ここではジョブの種類による指定方法の違いについて説明します。

2.3.4.1 バルクジョブの投入方法

ジョブがバルクジョブであることを宣言するために、pjsub コマンドの --bulk オプションを指定します。
また、バルクジョブとして投入したいサブジョブの数を、--sparam オプションを使用して、バルク番号の開始、終了で示します。

```
$ pjsub --bulk --sparam "開始バルク番号-終了バルク番号" [ジョブスクリプト]
```

バルク番号は0から999999まで指定でき、終了バルク番号は開始バルク番号より大きくなければいけません。バルク番号は1ずつ増加します。

以下にバルクジョブの投入例を示します。

```
データファイルが in-0.dat から in-9.dat まで10種類あり、それぞれを入力としてプログラムを実行する場合
$ cat bulkjob.sh
#!/bin/sh
...
INFILE=in-${PJM_BULKNUM}.dat          ← バルク番号から入力データファイル名を決定
OUTFILE=out-${PJM_BULKNUM}.dat        ← バルク番号から出力データファイル名を決定
./program ${INFILE} ${OUTFILE}        ← 入出力データファイルをプログラムの引数に指定
...

$ pjsub --bulk --sparam "0-9" bulkjob.sh    ← バルク番号を0から9まで指定し、サブジョブ bulkjob.sh を投入
```

上記は、バルクジョブに関する部分だけ示しています。

2.3.4.2 ステップジョブの投入方法

ジョブがステップジョブであることを宣言するために、pjsubコマンドの --step オプションを指定します。
ステップジョブでは、どのステップジョブのサブジョブかを示すために、ジョブIDまたはジョブ名を指定します。

[ジョブIDを指定する方法]

ステップジョブのジョブIDを指定してサブジョブを投入します。この場合、1番目のサブジョブと2番目以降のサブジョブでは、投入時のオプションに違いがあります。

1. 1番目のサブジョブ投入

```
$ pjsub --step stepjob1.sh
[INFO] PJM 0000 pjsub Job 100_0 submitted.
```

2. 2番目以降のサブジョブ投入

   1番目のサブジョブと同じステップジョブであることを示すために、--sparam "jid=" オプションで、ジョブID を指定して投入します。

```
$ pjsub --step --sparam "jid=100" stepjob2.sh
[INFO] PJM 0000 pjsub Job 100_1 submitted.
```

   1番目のサブジョブがすでに終了している場合は、該当するジョブが存在しないため、サブジョブの投入がエラーになります。

[ジョブ名を指定する方法]

各サブジョブのジョブ名を同じにし、ジョブ名を指定してサブジョブを投入します。この方法では、1番目と2番目以降のサブジョブの投入方法は基本的に同じにできます。

1. 1番目のサブジョブ投入

   サブジョブの投入時に、--sparam "jnam=" オプションも指定することでジョブ名を設定します。

```
$ pjsub --step --sparam "jnam=mystepjob" stepjob1.sh
[INFO] PJM 0000 pjsub Job 200_0 submitted.
```

2. 2番目以降のサブジョブ投入

   --sparam "jnam=" オプションで、ジョブ名を設定すると同時に、すでに存在する同名のステップジョブのサブジョブを示します。

```
$ pjsub --step --sparam "jnam=mystepjob" stepjob2.sh
[INFO] PJM 0000 pjsub Job 200_1 submitted.
```

   該当するジョブ名のステップジョブが存在しない場合は、新規ステップジョブが生成されます。

注意

--sparam "jnam=" オプションを使用する場合は、以下に注意してください。

- --sparam "jnam=" オプションを指定した場合、-N または --name オプションの有無に関係なく、--sparam "jnam=" オプションで指定したジョブ名がサブジョブに設定されます。
- pjsub コマンドに --sparam "jnam=" オプションを指定した場合は、ジョブスクリプト内での --sparam "jnam=" オプションの指定は無視されます。
- --sparam "jnam=" オプションを指定せずに投入したステップジョブに対して、ジョブ名を指定してサブジョブを投入するときは、最初のサブジョブ(ステップ番号0)のジョブ名を指定してください。
- --sparam "jnam=" オプションを使用せず、-N または --name オプションでジョブ名を指定した場合は、新規のステップジョブが生成されます。
  このため、同じジョブ名の異なるステップジョブが複数存在する場合があります。このとき、--sparam "jnam=" オプションでジョブ名を指定してサブジョブを投入すると、最後に生成されたステップジョブを指定したものとみなされます。

投入したサブジョブのステップ番号は、指定がなければ投入順に0から1ずつ増えます。ステップ番号は--sparam "sn=" オプションでユーザーが指定できます。

```
$ pjsub --step --sparam "sn=10" stepjob1.sh        # ステップ番号を 10 に設定
```

ユーザーがステップ番号を指定する場合、その時点での最大ステップ番号より大きい値でなければいけません。ステップ番号は pjstat コマンドで確認できます("2.4 ジョブの状況確認" 参照)。

1つのステップジョブに対する複数のサブジョブは、コンマまたは空白で区切って、pjsub コマンドでまとめて投入できます。

```
$ pjsub --step stepjob1.sh,stepjob2.sh       (または pjsub --step stepjob1.sh stepjob2.sh)
[INFO] PJM 0000 pjsub Job 300_0 submitted.
[INFO] PJM 0000 pjsub Job 300_1 submitted.
```

上記の場合、ステップ番号はジョブスクリプトを指定した順に 0、1、... と設定されます。--sparam "sn=" オプションを指定した場合は、指定値を起点としてステップ番号が設定されます。
また、すでに存在するステップジョブに対して、複数のサブジョブをまとめて投入できます。

```
$ pjsub --step stepjob1.sh
[INFO] PJM 0000 pjsub Job 400_0 submitted.
$ pjsub --step --sparam "jid=400" stepjob2.sh,stepjob3.sh  (または pjsub --step --sparam "jid=400" stepjob2.sh stepjob3.sh)
[INFO] PJM 0000 pjsub Job 400_1 submitted.
[INFO] PJM 0000 pjsub Job 400_2 submitted.
```

注意

複数のサブジョブをまとめて投入する場合は以下に注意してください。

- pjsub コマンドの引数で指定したオプションは、同時に投入するすべてのサブジョブに対する共通の指定になります。サブジョブごとに異なる指定をしたい場合は、ジョブスクリプト内で指定してください。
  ただし、ジョブ名、ジョブIDおよびステップ番号の指定については以下に注意してください。
  - サブジョブごとにジョブ名を変えたい場合は、ジョブスクリプト内で -N または --name オプションを指定してください。ジョブスクリプト内で --sparam "jnam=" オプションを指定する場合は、同時に投入するすべてのジョブスクリプト内で --sparam "jnam=" オプションが指定され、ジョブ名は同じでなければいけません。
  - 同時に指定した複数のジョブスクリプト内で、ステップジョブのジョブIDを指定する--sparam "jid="オプションを指定した場合は、最初に指定したジョブスクリプトの指定が有効になります。
  - ステップ番号を指定する--sparam "sn="オプションを pjsub コマンドの引数で指定した場合は、最初のサブジョブに対するステップ番号の指定になり、それ以降のサブジョブに対してはステップ番号は1ずつ増加します。

- サブジョブの標準出力ファイル、標準エラー出力ファイル、およびジョブ統計情報出力ファイルを指定する場合は、ファイル名の書式にステップ番号を含めるなどして、サブジョブごとに異なるファイルにしてください。同じファイル名の場合は、最後のサブジョブの出力で上書きされます。

参考

ジョブスクリプトのファイル名にコンマ(,)を使用する場合、コンマが区切り文字として解釈されないために、pjsub コマンドの --script-delimiter オプションに none を指定してください。

```
$ pjsub --script-delimiter none --step step,job1.sh
```

2番目以降のサブジョブ投入時は、先行するサブジョブの結果に応じてどういう動作をするかを --sparam "sd=" オプションで指定します。これをステップジョブの「依存関係式」と呼びます。
依存関係式の指定方法は以下の書式となります。

```
--sparam "sd=form[:[deletetype][:stepno[:stepno[...]]]]"
```

form は投入するサブジョブを実行するかどうかを判断する条件を示します。*deletetype* は、サブジョブを実行しない場合の詳細動作の指定です。*stepno* は式 *form* をどのサブジョブの実行結果に対して適用するかを示すステップ番号です。*stepno* を省略すると、直前のサブジョブの結果が対象となります。

式 *form*、削除タイプ *deletetype* に指定できる値は以下のとおりです。

式 form	説明
NONE	依存するサブジョブはないことを示します。つまり、"sd=" 指定をしない場合と同じです。 投入するサブジョブは先行するサブジョブが終了すると実行されます。 注) ただし、先行するサブジョブで、後続のサブジョブを削除した場合は実行されません。
param==*値* *param*!=*値* *param*>*値* *param*<*値* *param*>=*値* *param*<=*値*	*deletetype* の指定に従ってサブジョブを削除する(実行しない)ための条件を示します。 *param* は ec または pc で、それぞれ以下を意味します。 ec: 依存するサブジョブのジョブスクリプトの終了ステータス pc: 依存するサブジョブのジョブ終了コード (PJM コード) 条件 == や != の場合、値はコンマ(,)で区切って複数指定できます。

参考

ジョブスクリプトの終了ステータス(ec) とジョブ終了コード(pc) の違い

ジョブ終了コード (pc) は、ジョブ運用管理機能が正常にジョブを処理できたかどうかを表すコードです。ジョブスクリプトの終了ステータス (ec) が 0 以外であっても、ジョブとして正常に処理された場合は、ジョブ終了コードは 0 になります。
ジョブとして正常に処理できなかった場合、ジョブスクリプトの終了ステータスが0でも、ジョブ終了コードが 0 以外になる場合があります (例: 経過時間制限超過)。
ユーザは、ジョブ終了コード(pc)とジョブスクリプトの終了ステータス(ec)の違いを意識して依存関係式を考える必要があります。ジョブ終了コード pc の値の意味については、"A.1 pjstat や pjstat -v の出力" の "表A.1 pjstat コマンドの出力項目" にある項目 "PC" の説明を参照してください。

削除タイプ deletetype	説明
one	このサブジョブだけ削除します。 このサブジョブの結果に依存する後続のサブジョブは削除されません。 *deletetype* を省略した場合は one が指定されたものとみなします。
after	このサブジョブ、およびそれに依存する後続のサブジョブだけ削除します。
all	このサブジョブ、および後続のサブジョブをすべて削除します。

以下は、ジョブID 500 のステップジョブに対するサブジョブとして投入する例です。
この例では、ステップ番号 0 のサブジョブのジョブスクリプトの終了コードが 0 以外の場合はこのサブジョブ以降は実行しないものとします。

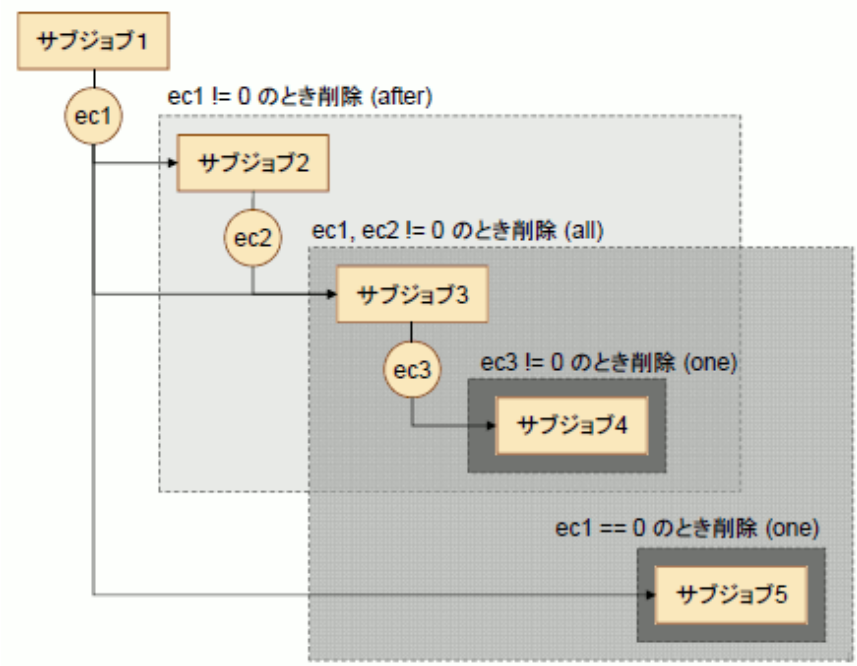
```
$ pjsub --step --sparam "jid=500,sd=ec!=0:all:0" stepjob2.sh
```

注意

--sparam オプションの引数は、1つの文字列となるようにシングルクォートまたはダブルクォートで囲んでください。

具体的なステップジョブの例を以下に示します。

図2.8 ステップジョブの例



- サブジョブ1

  最初に実行されます。

- サブジョブ2

  サブジョブ1の終了コードが 0 でない場合、サブジョブ2およびサブジョブ2に依存するジョブはすべて削除します。サブジョブ2の削除の影響を受けるサブジョブ3、4も削除対象となります。

- サブジョブ3

  サブジョブ1またはサブジョブ2の終了コードが 0 でない場合、サブジョブ3以降のすべてのジョブを削除します。サブジョブ4およびサブジョブ5も削除対象となります。

- サブジョブ4

  サブジョブ3の終了コードが0でない場合、サブジョブ4だけ削除します。サブジョブ5は削除されません。

- サブジョブ5

  サブジョブ1の終了コードが0の場合、サブジョブ5だけ削除します。

このステップジョブは以下のように投入します。

```
$ pjsub --step --sparam "sn=1" job1.sh                          ← ステップ番号 1 として投入
[INFO] PJM 0000 pjsub Job 500_1 submitted.                      ← この例ではジョブIDは 500
$ pjsub --step --sparam "jid=500,sn=2,sd=ec!=0:after:1" job2.sh ← 以下、ステップ番号 2 以降を投入
$ pjsub --step --sparam "jid=500,sn=3,sd=ec!=0:all:1:2" job3.sh
```

```
$ pjsub --step --sparam "jid=500,sn=4,sd=ec!=0:one:3" job4.sh
$ pjsub --step --sparam "jid=500,sn=5,sd=ec==0:one:1" job5.sh
```

1つのステップジョブで、サブジョブを異なるリソースユニットやリソースグループに投入できます。この場合、サブジョブの投入先リソースユニット名やリソースグループ名を指定する必要があります。指定しない場合は、ジョブACL機能で設定されているデフォルトのリソースユニット、リソースグループになることに注意してください。
以下は、ステップ番号1と3のサブジョブをリソースユニットrscunitPGに投入し、ステップ番号2のサブジョブをリソースユニットrscunitFXに投入する例です。

```
$ pjsub --step -L "rscunit=rscunitPG" --sparam "sn=1" job-PG1.sh
[INFO] PJM 0000 pjsub Job 600_1 submitted.
$ pjsub --step -L "rscunit=rscunitFX" --sparam "jid=600,sn=2,sd=ec!=0:after:1" job-FX.sh
[INFO] PJM 0000 pjsub Job 600_2 submitted.
$ pjsub --step -L "rscunit=rscunitPG" --sparam "jid=600,sn=3,sd=ec!=0:all:1:2" job-PG2.sh
[INFO] PJM 0000 pjsub Job 600_3 submitted.
```

2.3.4.3 ワークフロージョブの投入方法

ワークフロージョブは、複数のジョブの投入をユーザーが制御する方法です。"1.2.1.4 ワークフロージョブ" に示したように、ユーザーは複数のジョブを投入するシェルスクリプトを作成し、それを実行します。

2.3.4.4 マスタ・ワーカ型ジョブの投入方法

マスタ・ワーカ型ジョブを投入する場合は、pjsub コマンドに対し、--mswk オプションを指定してください。

```
$ pjsub --mswk ジョブスクリプト
```

資源の上限値の指定(-L、--rsc-list)やノード形状などは、必要に応じて pjsub コマンドの引数またはジョブスクリプト内に記述してください。

注意

- pjsub コマンドの --mswk オプションは、--step オプション、--bulk オプション、--interact オプションと同時には指定できません。
- マスタ・ワーカ型ジョブでは、ワーカプロセスを生成するノードはジョブ運用ソフトウェアの並列実行環境が決定、またはユーザーがプロセス生成時に指定します。
  このため、pjsub コマンドの --mpi rank-map-hostfile オプションの指定は意味がありません。このオプションを指定しても無視されます。

2.3.4.5 会話型ジョブの投入方法

ジョブが会話型ジョブであることを宣言するために、pjsub コマンドの --interact オプションを指定します。

参考

計算クラスタ管理ノードにログインできる管理者は、そこからpjsubコマンドでジョブを投入できますが、会話型ジョブだけはログインノードから投入しなければいけません

会話型ジョブには、ユーザーが端末からジョブの内容を対話的に入力する方法と、バッチジョブと同じようにジョブスクリプトでジョブの内容を指定する方法があります。

pjsubコマンドの引数にジョブスクリプトを指定しない場合は、ジョブの内容を対話的に入力する方法となり、擬似端末上でシェルが起動され、入力待ちの状態となります。

```
$ pjsub --interact
[INFO] PJM 0000 pjsub Job 405916 submitted.              ← 会話型ジョブ投入を示すメッセージ
[INFO] PJM 0081 .connected.                              ← 会話型ジョブの準備中を示すメッセージ
[INFO] PJM 0082 pjsub Interactive job 405916 started.    ← 会話型ジョブ開始を示すメッセージ
$                                                        ← 会話型ジョブ内のシェルプロンプト
...
```

```
$ exit                                                    ← シェルの終了
[INFO] PJM 0083 pjsub Interactive job 405916 completed.   ← 会話型ジョブ終了を示すメッセージ
```

このときのプロンプトは、会話型ジョブを実行する計算ノード上のシェルが表示しているものです。計算ノードでのカレントディレクトリは、ログインノードで pjsub コマンドを実行した際のカレントディレクトリと同じパスになります。

シェルを終了させるか、pjdel コマンドでジョブを削除すると("2.5.1 ジョブの削除" 参照)、会話型ジョブは終了します。

擬似端末からの入力は会話型ジョブで実行しているシェルに渡されますが、会話型ジョブに対して、特別な動作を指示するにはエスケープ文字としてチルダ記号 '~' を使用します。チルダ記号とそれに続く入力によって以下の動作を指示できます。なお、エスケープ文字としてのチルダ記号は、改行後の最初の文字でなければいけません。

表2.25 会話型ジョブにおけるエスケープ

入力	動作
~~	チルダ記号をシェルに送ります。
~.	会話型ジョブを終了します。会話型ジョブで実行していたシェルは終了し、擬似端末は切断されます。
~<Ctrl+z>	会話型ジョブを実行している pjsub コマンドをサスペンドし、pjsub コマンドを実行したコマンドラインに端末制御を戻します。pjsub コマンドをフォアグラウンドに戻すことで会話型ジョブを継続します。
~?	エスケープの一覧を表示します。
キーシーケンス <Ctrl+¥>, <Ctrl+_>, <Ctrl+¥>, <Ctrl+_>, <Q(shift+q)>	McKernelモードの会話型ジョブでは、このキーシーケンスを入力するとジョブは終了します。

pjsub コマンドの引数にジョブスクリプトを指定する場合は、バッチジョブと同様にジョブスクリプトの内容に従って実行されます。

```
$ pjsub --interact interactjob.sh
[INFO] PJM 0000 pjsub Job 405918 submitted.
[INFO] PJM 0081 .connected.
[INFO] PJM 0082 pjsub Interactive job 405918 started.
...                                                       ← ジョブスクリプトの出力内容
[INFO] PJM 0083 pjsub Interactive job 405918 completed.
```

資源不足によって会話型ジョブがすぐに実行できない場合、資源割り当て待ちになり、後続のジョブ実行に影響が出る可能性があります。このため、会話型ジョブでは資源割り当て最長待ち時間(秒)を --sparam "wait-time=" オプションで指定できます。資源割り当て待ちが指定した時間を超えた場合は、ジョブはキャンセルされます。

```
$ pjsub --interact --sparam "wait-time=600"    ← 資源割り当て待ちを最長600秒(10分)待つ
[INFO] PJM 0000 pjsub Job 291 submitted.
[INFO] PJM 0081 .
[INFO] PJM 0080 pjsub Interactive job 291 is canceled due to the resource allocation timeout. ← 待ち時間が600秒を超えたため
       The timeout period "t" can be specified by "--sparam wait-time=t".                      キャンセルされた
```

会話型ジョブでは、pjsub コマンドの以下のオプションは無視されます。

表2.26 会話型ジョブで無視されるpjsubコマンドのオプション

オプション	説明
--at	ジョブ実行開始時刻
-e、--err	標準エラー出力ファイル
-j	標準エラー出力を標準出力ファイルに向ける
--bulk, --step	ジョブモデル
-m e	実行終了のメール通知
-m r	実行再開のメール通知
--restart	ジョブの自動再実行の有効化

オプション	説明
-o、--out	標準出力ファイル
-p	ジョブの優先度
-w	pjsub コマンドの復帰待ち合わせ

注意

会話型ジョブを実行した際、<Ctrl+c>を入力後にexitで会話型ジョブを終了すると、通常の終了メッセージの前に以下のメッセージ"[INFO] PLE 0094"が表示されますが、動作には問題ありませんので無視してください。

```
[INFO] PLE 0094 plexec The interactive job has received the signal.(sig=2)
[INFO] PJM 0083 pjsub Interactive job 405920 completed.
```

2.3.5 ノード選択ポリシーの指定 [PG]

PRIMERGYサーバでは、ノード資源を割り当てるときに、ノード選択ポリシー("1.6.4 仮想ノード単位での割り当て"参照)を指定できます。

2.3.5.1 仮想ノード配置ポリシー

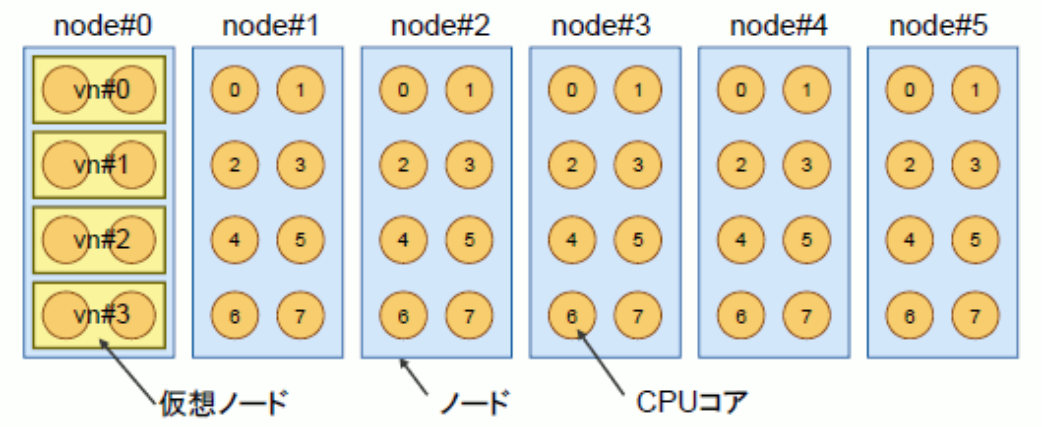
仮想ノード配置ポリシーは、仮想ノード割り当てのジョブに対してだけ指定でき、pjsub コマンドの -P "vn-policy=" オプションで指定します。

Absolutely PACK

すべての仮想ノードを必ず1つのノードに配置する場合は -P "vn-policy=abs-pack" オプションを指定します。
以下は、2つのCPUコアを持つ4つの仮想ノードを Absolutely PACK で配置する例です。

```
$ pjsub -L "vnode=4,vnode-core=2" -P "vn-policy=abs-pack" job.sh
または
$ pjsub -L "vnode=4(core=2)" -P "vn-policy=abs-pack" job.sh
```

図2.9 Absolutely PACKでの仮想ノード配置

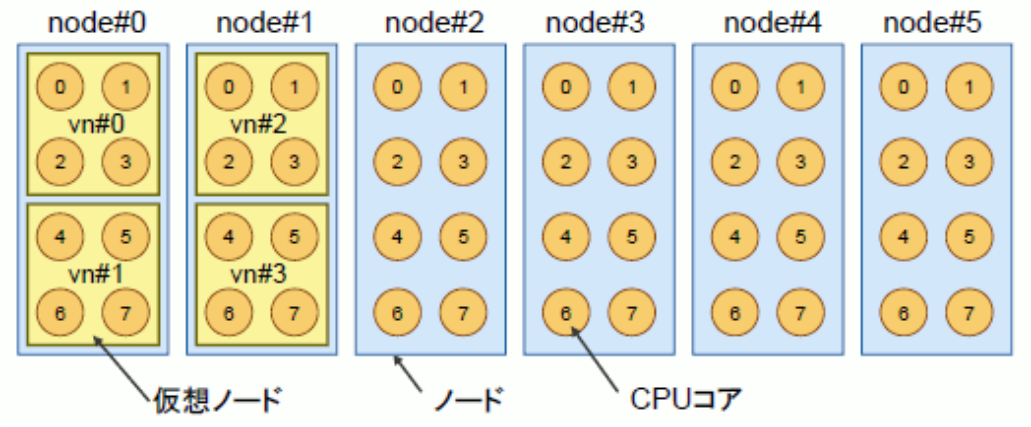


PACK

仮想ノードをできるだけ1つのノードに配置する場合は -P "vn-policy=pack" オプションを指定します。
以下は、4つのCPUコアを持つ4つの仮想ノードを PACK で配置する例です。

```
$ pjsub -L "vnode=4,vnode-core=4" -P "vn-policy=pack" job.sh
または
$ pjsub -L "vnode=4(core=4)" -P "vn-policy=pack" job.sh
```

図2.10 PACKでの仮想ノード配置



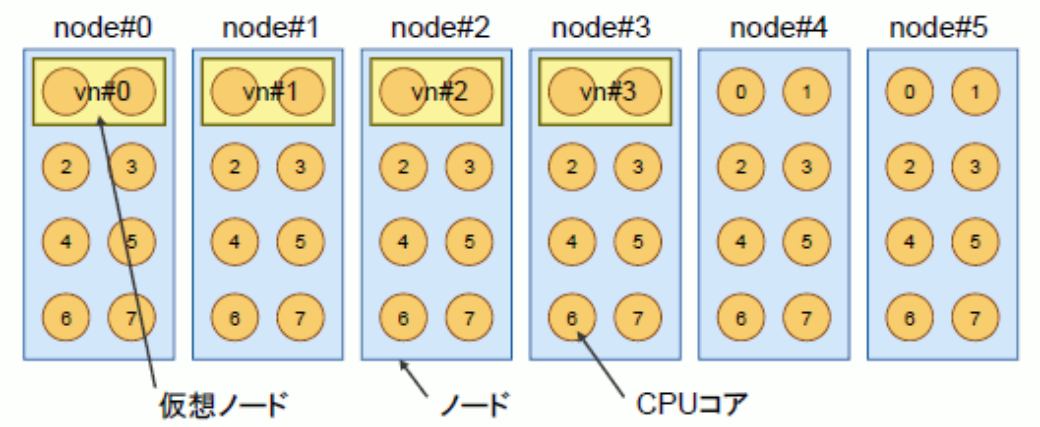
上記では、1つのノードに仮想ノードを2つだけ配置できるため、2ノードを使用しています。

Absolutely UNPACK

各仮想ノードを必ず異なるノードに配置する場合は -P "vn-policy=abs-unpack" オプションを指定します。
以下は、2つのCPUコアを持つ4つの仮想ノードを Absolutely UNPACK で配置する例です。

```
$ pjsub -L "vnode=4,vnode-core=2" -P "vn-policy=abs-unpack" job.sh
または
$ pjsub -L "vnode=4(core=2)" -P "vn-policy=abs-unpack" job.sh
```

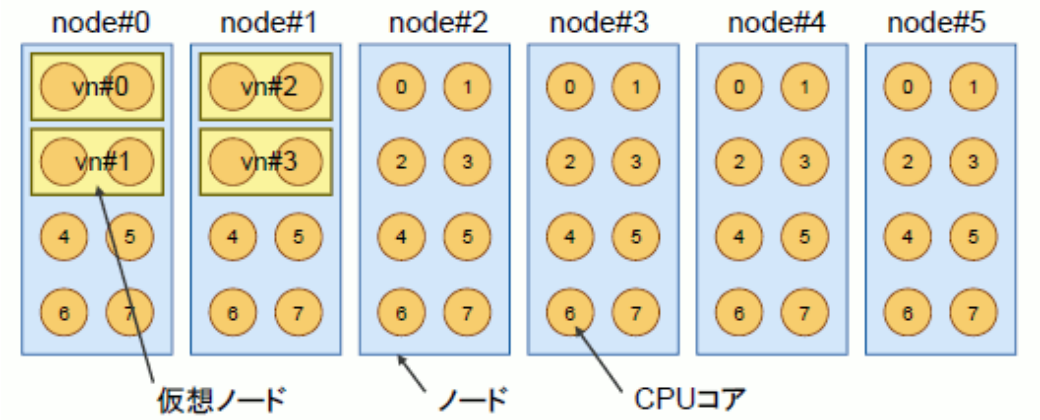
図2.11 Absolutely UNPACKでの仮想ノード配置



Absolutely UNPACK では、仮想ノードを必ず N 個ずつ、異なるノードに配置するという指定もできます。この場合、-P "vn-policy=abs-unpack=*N*" オプションを指定します。
以下は、2つのCPUコアを持つ4つの仮想ノードを、各ノードに2つずつ Absolutely UNPACK で配置する例です。

```
$ pjsub -L "vnode=4,vnode-core=2" -P "vn-policy=abs-unpack=2" job.sh
または
$ pjsub -L "vnode=4(core=2)" -P "vn-policy=abs-unpack=2" job.sh
```

図2.12 Absolutely UNPACKで2つずつ仮想ノードを配置



注意

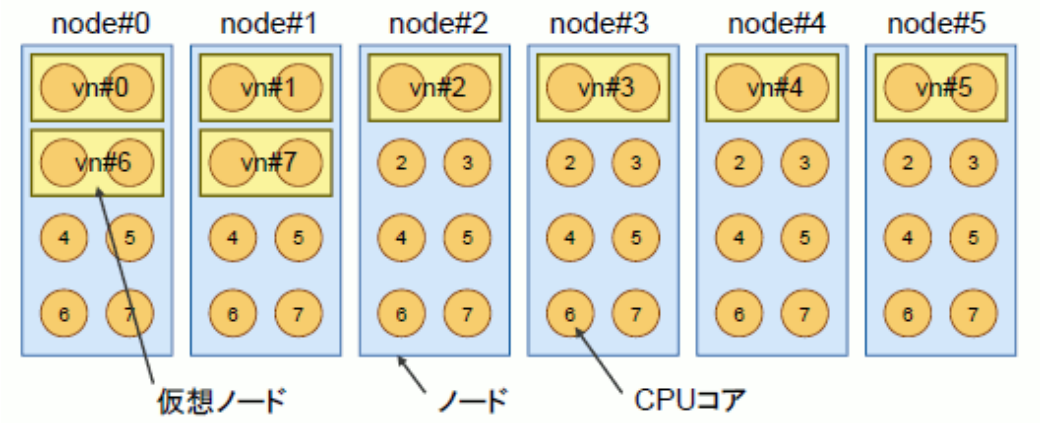
"vnode=*M*"(仮想ノード数)と"vn-policy=abs-unpack=*N*" を指定した場合、*M*が *N*の倍数でなければ pjsub コマンドはエラーとなります。

UNPACK

各仮想ノードをできるだけ異なるノードに配置する場合は、-P "vn-policy=unpack" オプションを指定します。
以下は、2つのCPUコアを持つ8つの仮想ノードを UNPACK で配置する例です。

```
$ pjsub -L "vnode=8,vnode-core=2" -P "vn-policy=unpack" job.sh
または
$ pjsub -L "vnode=8(core=2)" -P "vn-policy=unpack" job.sh
```

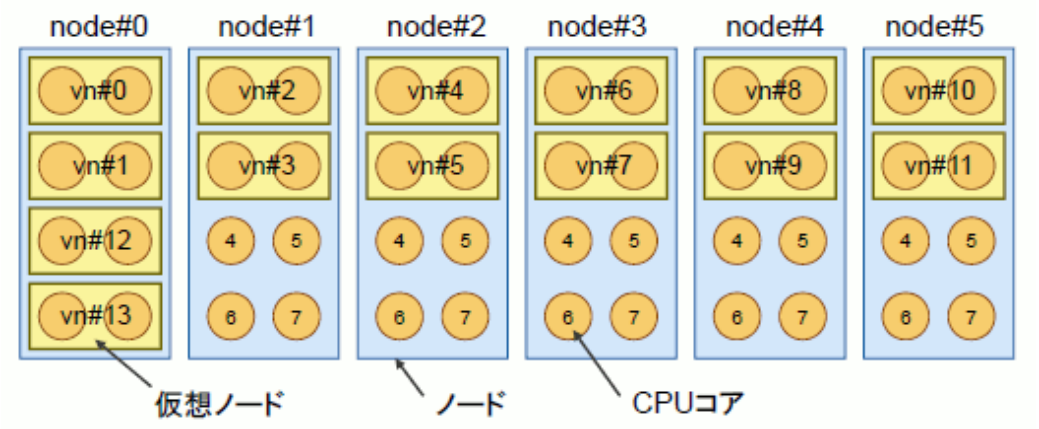
図2.13 UNPACKでの仮想ノード配置



UNPACK では、仮想ノード N 個ずつ、できるだけ異なるノードに配置するという指定もできます。この場合、-P "vn-policy=unpack=*N*" オプションを指定します。
以下は、2つのCPUコアを持つ14個の仮想ノードを、UNPACK でできるだけ2つずつノードに配置する例です。

```
$ pjsub -L "vnode=14,vnode-core=2" -P "vn-policy=unpack=2" job.sh
または
$ pjsub -L "vnode=14(core=2)" -P "vn-policy=unpack=2" job.sh
```

図2.14 UNPACKで2つずつ仮想ノードを配置



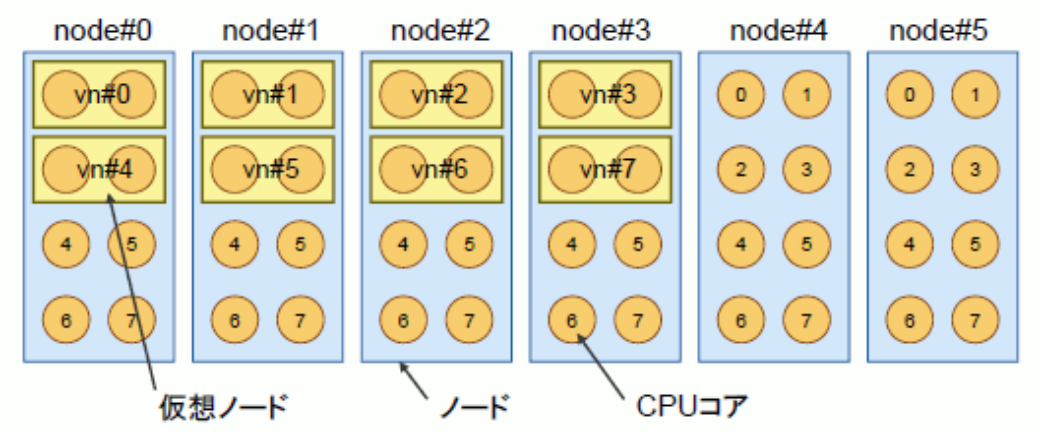
注意

"vnode=*M*"(仮想ノード数) と "vn-policy=unpack=*N*" を指定した場合、*M* が *N* の倍数でなければ pjsub コマンドはエラーとなります。

-P "vn-policy=unpack" 指定時に、仮想ノード数だけでなく、ノード数も指定した場合 (-L vnode=*M*,node=*N*、ただし *M* ≧ *N*)、仮想ノードは1つのノードに、*M*/*N* 個ずつ配置されます。

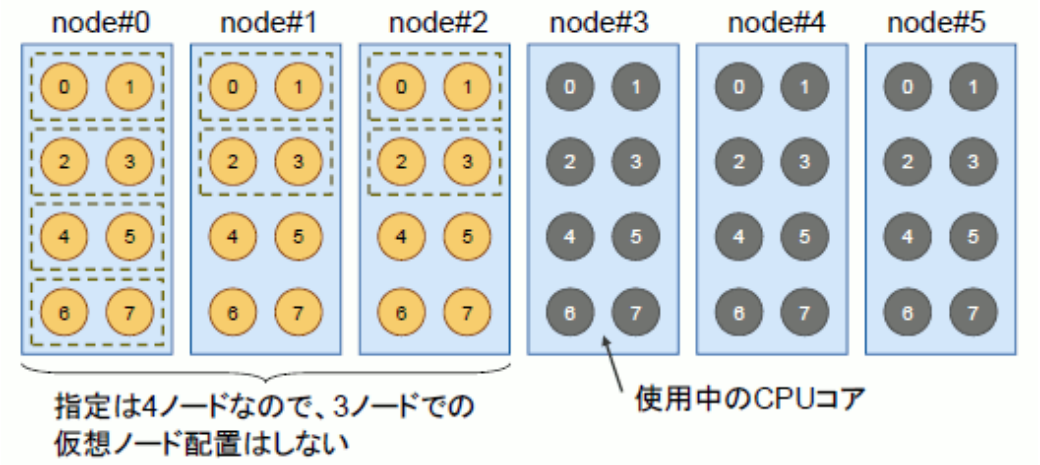
```
$ pjsub -L "vnode=8,vnode-core=2,node=4" -P "vn-policy=unpack" job.sh
または
$ pjsub -L "vnode=8(core=2),node=4" -P "vn-policy=unpack" job.sh
```

図2.15 UNPACKとノード数指定 (1)



また、ノード数を指定した場合は、必ず指定した数のノードを必要とします。このため、以下の図のようにノードが不足する場合は、-P "vn-policy=unpack" 指定であっても仮想ノードは配置できません。

図2.16 UNPACKとノード数指定 (2)



上記では、ノード数が 4 と指定されているため、ノード0,1,2 の 3 ノードだけで UNPACK 配置をすることはありません。

2.3.5.2 ランクマップ

ランクマップについては、"2.3.6 MPI ジョブの投入" を参照してください。

2.3.5.3 ノード選択方式

仮想ノードを配置するノードの選び方には、分散方式と集中方式の2つがあります。どちらの方式を使うかは、管理者が設定し、ユーザーは指定できません。
以下は、2つのCPUコアを持つ5つの仮想ノードを、できるだけ異なるノードに配置するように投入した場合の(UNPACK)、分散方式と集中方式における割り当ての違いを示します。

```
$ pjsub -L "vnode=5,vnode-core=2" -P "vn-policy=unpack" job.sh
または
$ pjsub -L "vnode=5(core=2)" -P "vn-policy=unpack" job.sh
```

図2.17 分散方式によるノードの選択

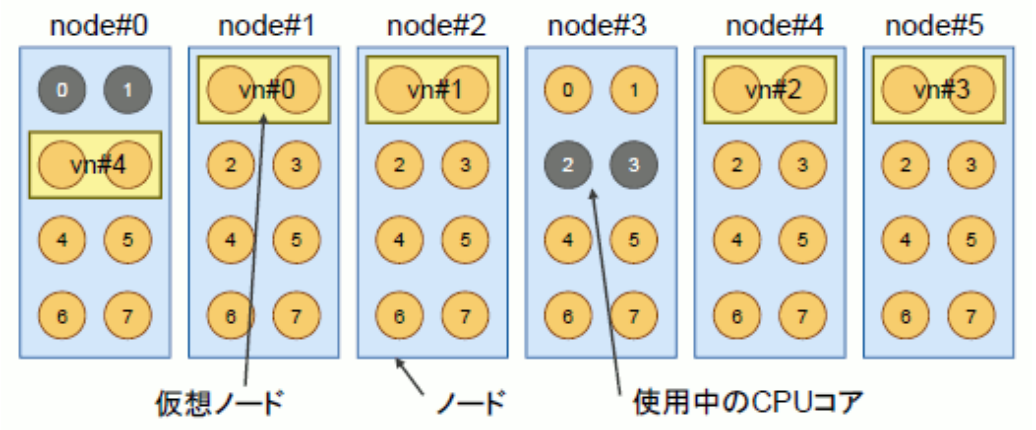
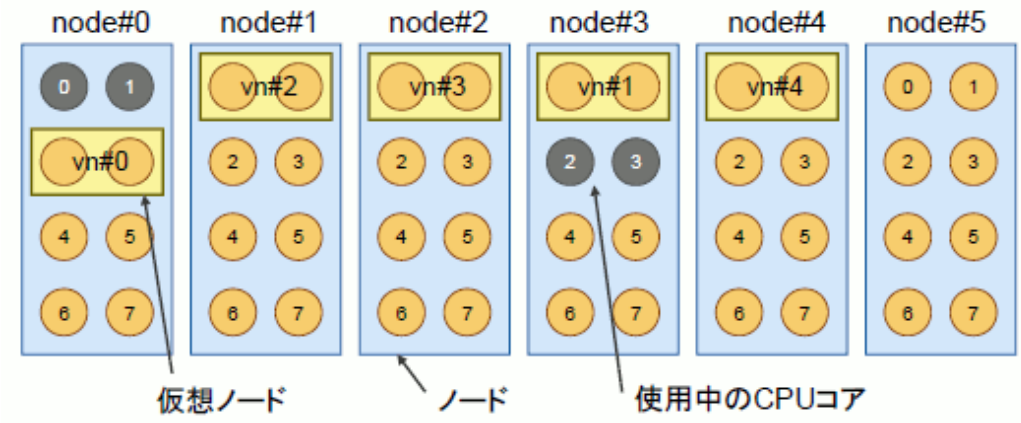


図2.18 集中方式によるノードの選択



2.3.5.4 割り当てノード優先度制御

割り当てノード優先度制御は、ノードに設定された優先度順に従ってノードを選択する方式です。
ノードの優先度およびこの方式を使うかどうかは、管理者が設定するため、ユーザーはジョブ投入時に意識する必要はありません。

2.3.5.5 実行モードポリシー

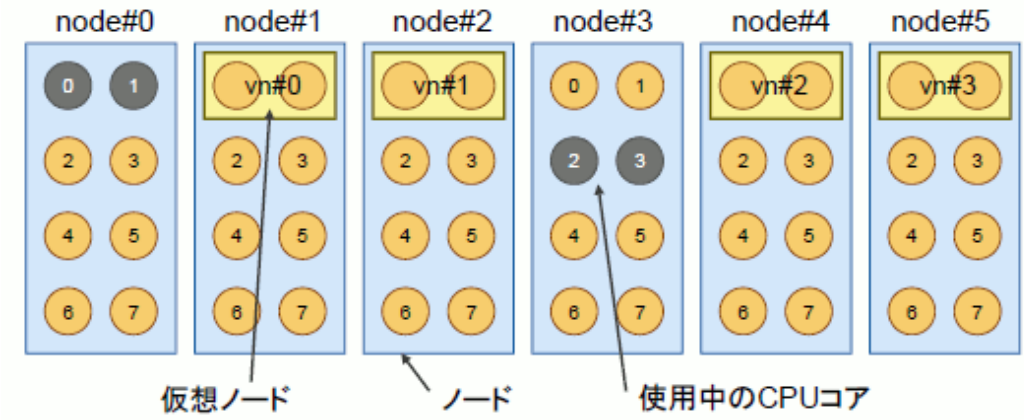
実行モードポリシーは、投入する仮想ノード割り当てジョブが、ほかのジョブとノードを共有することを許すかどうかを指定します。
これには、SIMPLEX (専有) と SHARE (共有) があり、pjsub コマンドの -P "exec-policy=" オプションで指定します。
以下は、SIMPLEX と SHARE をそれぞれ指定した場合の例です。

SIMPLEX

以下は、2つのCPUコアを持つ4つの仮想ノードを、必ず異なるノードに配置し、ジョブがそれらノードを専有する例です。

```
$ pjsub -L "vnode=4,vnode-core=2" -P "vn-policy=abs-unpack,exec-policy=simplex" job.sh
または
$ pjsub -L "vnode=4(core=2)" -P "vn-policy=abs-unpack,exec-policy=simplex" job.sh
```

図2.19 SIMPLEX 指定によるノードの専有



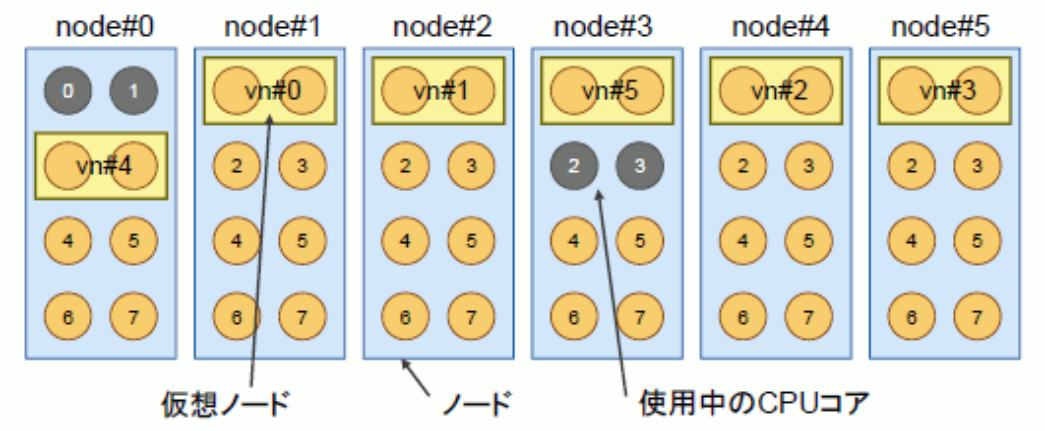
なお、上記では、専有できるノードは4つ(node#1、node#2、node#4、node#5)だけなので、5つ以上の仮想ノードを SIMPLEX で割り当てることはできません。

SHARE

以下は、2つのCPUコアを持つ6つの仮想ノードを、必ず異なるノードに配置し、それらノードはほかのジョブと共有を許す例です。

```
$ pjsub -L "vnode=6,vnode-core=2" -P "vn-policy=abs-unpack,exec-policy=share" job.sh
または
$ pjsub -L "vnode=6(core=2)" -P "vn-policy=abs-unpack,exec-policy=share" job.sh
```

図2.20 SHARE 指定によるノードの共有



上記では、5つ目および6つ目の仮想ノード(vn4, vn5)は、ほかのジョブとノードを共有しています。

2.3.6 MPI ジョブの投入

MPI ジョブ投入時には pjsub コマンドの --mpi オプションで以下のパラメーターを指定できます。

- プロセスの形状 [FX]
- 生成するプロセス数
- 生成するプロセスのランクの割り当てルール

--mpi オプションの指定例を以降で説明します。

注意

- ノードを1つ(node=1)または仮想ノードを1つ(vnode=1)割り当てて MPI ジョブを実行する場合は、--mpi オプションを必ず指定してください。
- 本節では、Development Studioで作成したMPIプログラムをジョブとして投入するための方法を説明します。Development Studio以外の MPI 処理系を実行する方法は、"付録C Development Studio以外の MPI 処理系の実行について" を参照してください。

2.3.6.1 プロセスの形状の指定 [FX]

FXサーバで、ノード割り当てジョブとして MPI ジョブを実行する場合、pjsub コマンドの --mpi オプションの shape パラメーターを使用することで、プログラム起動時に生成するプロセスの形状 (使用するノードの形状) を指定できます。

プロセスの形状は、1次元、2次元、または3次元の形状で、-L オプション (または --rsc-list) の node パラメーターで指定するノード形状と同じ次元数でなければいけません。--mpi オプションの shape パラメーターが省略された場合は -L オプションの node パラメーターの指定が使用されます。

```
[1次元形状] --mpi "shape=X"
[2次元形状] --mpi "shape=XxY"
[3次元形状] --mpi "shape=XxYxZ"
```

並列実行環境は、MPI プログラム起動時に生成するプロセスと動的に生成するプロセスを同じノードに割り当てないようにノードを選択します。このため、MPI ジョブに割り当てるノード形状 (-L node=) は、動的に生成するプロセスも含め、ジョブ内で同時に実行される MPI プロセスがすべて収まる大きさでなければいけません。

参考

割り当てるノードとプロセス(ランク)の対応を以下の方法で指定する場合は、起動時に生成するMPIプロセスと動的に生成するMPIプロセスを同じノードに割り当てることができます。

- mpiexec コマンドの --vcoordfile オプション

- MPI_Comm_spawn 関数や MPI_Comm_spawn_multiple 関数の info キー vcoordfile

mpiexecコマンド、MPI_Comm_spawn 関数、およびMPI_Comm_spawn_multiple 関数の詳細については、Development Studioのマニュアル「MPI使用手引書」を参照してください。

例えば、MPI ジョブ投入時に -L node=1 を指定すると、1ノードだけ割り当てられるため、動的なプロセス生成は失敗します。
なお、動的に生成したプロセスが終了すると、それが使用していたノードはプロセスを新たに動的生成するために使用できます。

以下は、MPI プログラム prog_A に対し、3次元のプロセス形状 (X 軸 4、Y 軸 3、Z 軸 2 ノード) を指定する例です。

```
$ cat job.sh                                          ← MPIプログラム prog_A を実行するジョブスクリプト job.sh
#!/bin/sh
...
mpiexec ./prog_A
$ pjsub -L "node=4x3x2" --mpi "shape=4x3x2" job.sh    ← ノード形状 4x3x2、プロセス形状 4x3x2を指定してジョブ投入
```

2.3.6.2 生成するプロセス数の指定

MPI プログラムが起動時に生成するプロセス数は、pjsub コマンドの --mpi オプションの proc パラメーターで指定します。

```
--mpi "proc=procnum"
```

proc パラメーターの値は以下に従います。

- FXサーバで実行するジョブの場合
  指定できる最大値は、"*shape パラメーターで指定したノード数* × *N*"です。ここで *N* は計算ノード当たりのCPUコア数です。
  最大値を指定した場合、1ノード当たりのすべてのコアに、重なりがないよう、各プロセスを固定させて実行するフラット並列ジョブとなります。
  この最大値を言い換えると、1つのノード内で生成できるプロセス数は "*proc パラメーター*/*shape パラメーターで指定したノード数*" (小数点以下切り上げ) です。
  この1ノード当たりのプロセス数は、MPI プログラムがプロセスを動的に生成する際にも適用されます。
  proc パラメーターを省略した場合は、shape パラメーターで指定したノード数を採用します。shape パラメーターも省略した場合は、-L オプションの node パラメーターで指定したノード数を採用します。

- PRIMERGYサーバで実行するジョブの場合
  1つの仮想ノード内で生成できるプロセス数は 1 です。また、node パラメーターでノードを指定する場合、1つのノードは1つの仮想ノードと同様に扱うため、1つのノード内で生成できるプロセス数も 1 です。
  このため、proc パラメーターで指定できる最大値は、指定するノード数または仮想ノード数です。proc パラメーターを省略した場合は、指定したノード数または仮想ノード数を採用します。

注意

proc パラメーターの最大値を超えて指定した場合は以下のようになります。

- FXサーバで実行するジョブの場合
  pjsub コマンドは、ジョブの受付けを拒否します。

- PRIMERGYサーバで実行するジョブの場合
  ジョブ実行時に並列実行環境の内部コマンド plexec のエラーメッセージがジョブの標準エラー出力ファイルに出力されます。メッセージの詳細はマニュアル「ジョブ運用ソフトウェア コマンドリファレンス」の"第3章 エンドユーザ向けコマンドリファレンス" にある "plexec コマンド" を参照してください。

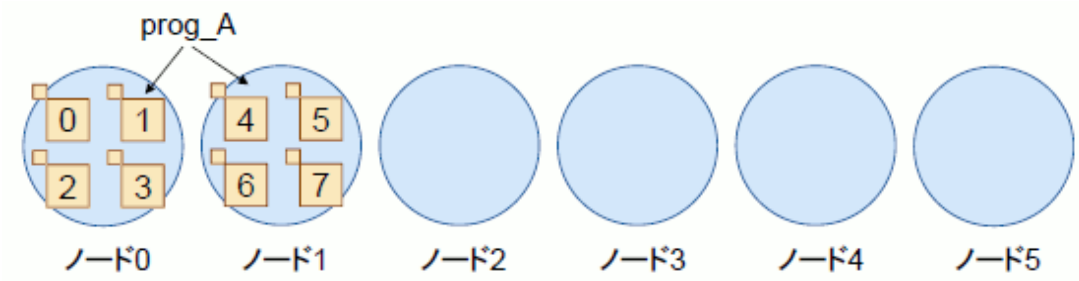
  ```
  [ERR.] PLE 0070 plexec The number of processes "n" must be smaller than or equal to the number of virtual nodes "m":
  "pjsub -L vnode=m", "pjsub --mpi proc=n".
  ```

以下では、動的プロセス生成をする MPI プログラムを例にして、node、shape および proc パラメーターの関係を説明します。

```
$ cat job.sh
#!/bin/sh
...
```

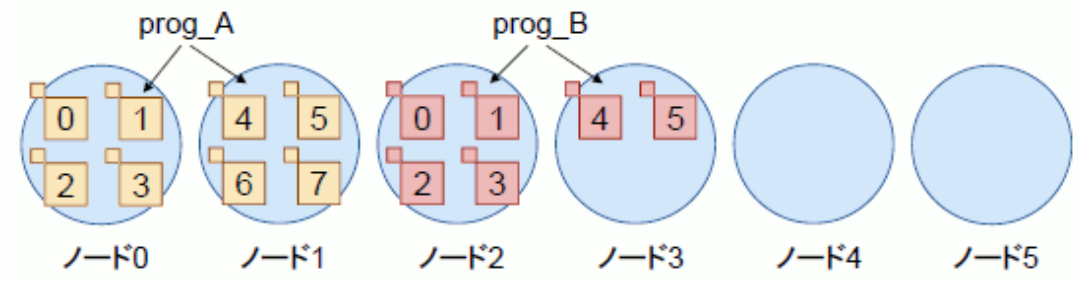
```
mpiexec ./prog_A
$ pjsub -L "node=6" --mpi "shape=2,proc=8" job.sh
```

1. この例では、ジョブはノード割り当てジョブとし、ジョブに 6 ノードを割り当てます (-L "node=6")。
2. MPI プログラム prog_A が実行されると、ノード0、1 の2ノード上で8つのプロセスが生成されます (--mpi "shape=2,proc=8")。このとき、1つのノードには 4 プロセスずつ生成されます (proc/shape)。



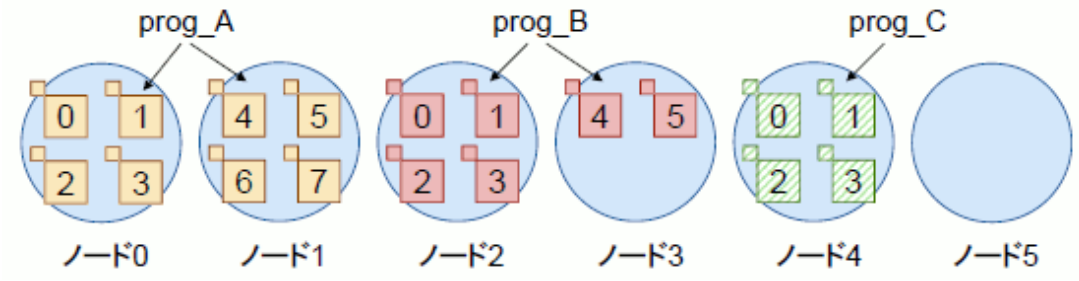
3. MPI プログラム prog_A が MPI_Comm_spawn 関数で MPI プログラム prog_B のプロセスを 6 つ生成すると、prog_A のプロセス形状を継承し、1ノード当たり最大4プロセスまで生成され、ノード 2、3 が使用されます。

```
MPI_Comm_spawn("prog_B", NULL, 6,...);
```



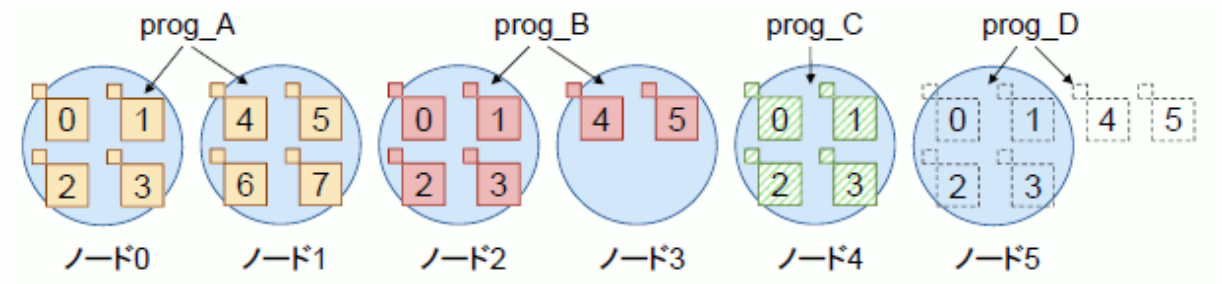
4. MPI プログラム prog_A が MPI プログラム prog_C のプロセスを4つ生成すると、prog_A のプロセス形状を継承し、1ノード当たり最大4プロセスまで生成され、ノード4 でプロセスが生成されます。このとき、ノード3 に空きがありますが、prog_C は prog_B とノードを共有しないように、プロセスはノード4 で生成されます。

```
MPI_Comm_spawn("prog_C", NULL, 4,...);
```



5. MPI プログラム prog_A がさらに MPI プログラム prog_D のプロセスを6つ生成しようとすると、ノードが2つ必要ですが、空きノードはノード5だけなので、プロセス生成は失敗します。

```
MPI_Comm_spawn("prog_D", NULL, 6,...);
```



注意

本例では pjsub の--mpi shapeパラメーターを指定した場合ですが、指定を省略した場合(MPI_Comm_spawn関数やMPI_Comm_spawn_multiple関数のinfoキーvcoordfileを指定せずにMPIジョブの動的プロセスを生成する場合)でも、動的プロセスの生成先ノードは、静的プロセスの存在するノードとは別のノードとなります。

ノード割り当てジョブに対しては、ノード当たりのMPIプロセス生成数の上限を指定できます。

表2.27 ノード当たりのMPIプロセス生成数の上限

オプション	意味
--mpi max-proc-per-node=*n*	1ノード当たりに生成するMPIプロセス数の上限値。 ノード割り当てジョブに対してだけ有効です。

これを --mpi proc=*n* オプションと同時に指定した場合の動作は以下のとおりです。

表2.28 MPIプロセスの生成数を指定するオプションの組み合わせと動作

--mpi proc=*n1*	--mpi max-proc-per-node=*n2*	動作
指定あり	指定なし	[プログラム開始時] 次節のランク割り当てルールに従って、1つのノードに *n1*/*shape* 個または *n1*/*node*個ずつ(小数点以下切り上げ)、計*n1*個に達するまで、MPIプロセスが生成されます。 [動的プロセス生成時] 1つのノード内に最大*n1*/*shape*個または*n1*/*node*個(小数点以下切り上げ)のMPIプロセスが生成できます。 *shape*: -L shape パラメーターで指定したノード数 (FXサーバだけ) *node*: -L node パラメーターで指定したノード数 (-L shape パラメーターを指定しない場合)
指定なし	指定あり	[プログラム開始時] 各ノードに*n2*個のMPIプロセスが生成されます。 [動的プロセス生成時] 各ノードに*n2*個まで、MPIプロセスを生成できます。
指定あり	指定あり	[プログラム開始時] 次節のランク割り当てルールに従って、1つのノードに*n2*個まで、計*n1*個に達するまで、MPIプロセスが生成されます。 [動的プロセス生成時] 各ノードに*n2*個まで、MPIプロセスを生成できます。

注意

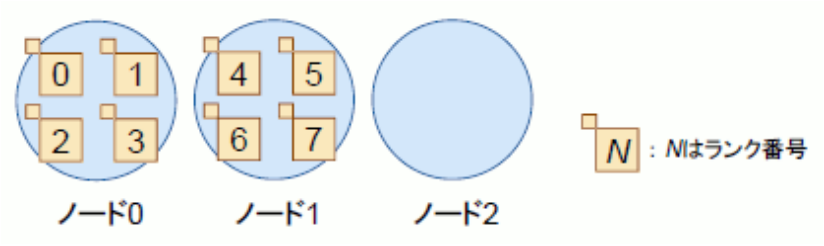
上記の動作は、ジョブスクリプト内で指定したmpiexecコマンドにオプションを指定しない場合のデフォルトの動作です。

以下に、例を示します。

[例1]

```
[ジョブスクリプト]
#PJM -L node=3
#PJM --mpi "shape=2,proc=8"
mpiexec a.out
```

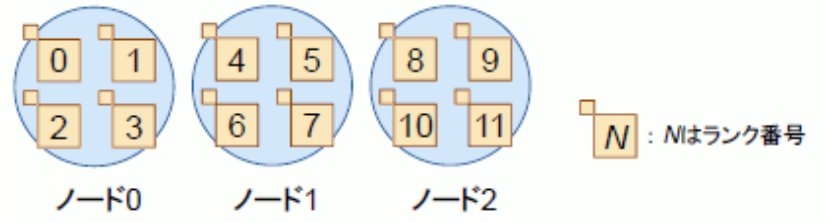
この場合、MPIプロセスの配置は以下のようになります。



[例2]

```
[ジョブスクリプト]
#PJM -L node=3
#PJM --mpi max-proc-per-node=4
mpiexec a.out
```

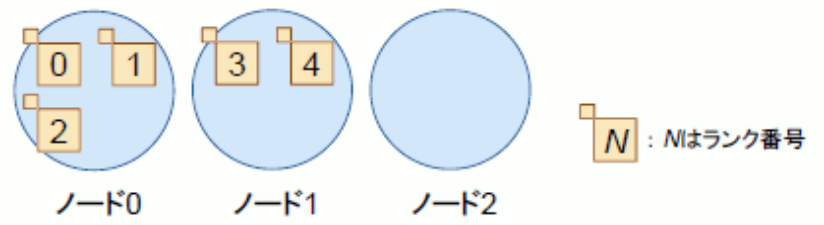
この場合、MPIプロセスの配置は以下のようになります。



[例3]

```
[ジョブスクリプト]
#PJM -L node=3
#PJM --mpi proc=5
#PJM --mpi max-proc-per-node=3
mpiexec a.out
```

この場合、MPIプロセスの配置は以下のようになります。



2.3.6.3 ランクに対するノードの割り当てルール

MPI では、プロセスの識別のために、プロセス番号に相当するランクと呼ぶ番号を割り当てます。ジョブ運用管理機能は、プロセスをランクの順でノードに割り当てます。このとき、ランクに対して割り当てるノードの選択ルールを pjsub コマンドの --mpi オプションで指定できます。

- 連続したランクに対して、同一のノードから順に割り当てる場合は、rank-map-bychip パラメーターを指定します。詳細は "2.3.6.4 rank-map-bychip パラメーター" を参照してください。
  連続したランクに対して、別のノードを順に割り当てる場合は、rank-map-bynode パラメーターを指定します。詳細は "2.3.6.5 rank-

map-bynode パラメーター" を参照してください。
両パラメーターは互いに排他です。
どちらも指定しない場合、FXサーバとPRIMERGYサーバではそれぞれ以下のようにみなします。

- FXサーバの場合 (ノード割り当てジョブに対してだけ)
  rank-map-bychip パラメーターが指定されたものとみなします。
- PRIMERGYサーバの場合 (仮想ノード割り当てジョブに対してだけ)
  仮想ノード配置ポリシーによる仮想ノード配置順に仮想ノードIDが設定されます。なお、PRIMERGYサーバでは、仮想ノードIDとランクの順番(ランク番号)は一致します。

参考

1ノードに1つのランク(プロセス)を生成する場合は、どちらのパラメーターを指定してもノードの割り当て方は同じになります。

- FXサーバでは、ランクに対して別のノードを割り当てる際に、どのノードをどのような順番で選択するかを rank-map-hostfile パラメーターで指定できます。詳細は "2.3.6.7 rank-map-hostfile パラメーター [FX]" を参照してください。
  rank-map-hostfile パラメーターは、rank-map-bychip パラメーターまたは rank-map-bynode パラメーターと共に指定できます。
  rank-map-hostfile パラメーターの指定を省略した場合、rank-map-bynode または rank-map-bychip パラメーターの *rankmap* に従います。詳細は、"2.3.6.6 rankmap に指定するノード割り当て順序 [FX]" を参照してください。

FXサーバでは、自分の指定したプロセス間の通信距離が短くなるように配置することで、通信の効率化を図ることができます。

注意

- FXサーバでは、ノード割り当てジョブに対してだけrank-map-bychip、rank-map-bynode、および rank-map-hostfileパラメーターが指定できます。
- PRIMERGYサーバをノード単位で割り当てる場合、ノードと同じサイズの仮想ノードを割り当てるイメージになります。このため、rank-map-bychip と rank-map-bynode パラメーターを指定する意味はありません。

以降では、--mpi オプションで指定可能なランク割り当てルールを説明します。

2.3.6.4 rank-map-bychip パラメーター

ここでは、rank-map-bychip パラメーターによるノードの割り当て方法について説明します。

FXサーバの場合 (ノード割り当てジョブに対してだけ)

rank-map-bychip パラメーターの書式は以下のとおりです。

```
--mpi "rank-map-bychip[:rankmap]"
--mpi "rank-map-bychip,rank-map-hostfile=filename"
```

ノードは以下のように割り当てられます。

1. 1つのノードに、指定された数のランクを割り当てます。
   このランクの数は、(procパラメーター)/(shapeパラメーターのノード数)で指定します(小数点以下切り上げ)。shape パラメーターが指定されない場合は、node パラメーターで指定された数を使用します。
2. 次のノードを選択します。
   選択するノードの順番は、*rankmap* の指定、または、rank-map-hostfile パラメーター ("2.3.6.7 rank-map-hostfile パラメーター [FX]" 参照) で指定できます。*rankmap* とrank-map-hostfile パラメーターを同時に指定した場合、rank-map-hostfile パラメーターが優先されます。*rankmap* に指定する値は、"2.3.6.6 rankmap に指定するノード割り当て順序 [FX]" を参照してください。
3. 1、2を繰り返して、すべてのランクにノードを割り当てます。

PRIMERGYサーバの場合 (仮想ノード割り当てジョブに対してだけ)

rank-map-bychip パラメーターの書式は以下のとおりです。

```
--mpi "rank-map-bychip=n"
```

仮想ノードは以下のように割り当てられます。

1. 1つのノードに、--mpi "rank-map-bychip=*n*" で指定された数 *n* の仮想ノードを設定します。
   rank-map-bychip=*n* パラメーターは、unpack=*m* または abs-unpack=*m* と共に指定し、*m* は *n* の倍数としてください。
   設定される仮想ノードの順番は、ノードID、仮想ノードIDが小さい順です。また、割り当てるプロセスのランク番号は、仮想ノードIDに一致します。
2. 次のノードを選択します。
3. 1、2を繰り返してすべての仮想ノードを配置します。

2.3.6.5 rank-map-bynode パラメーター

ここでは、rank-map-bynode パラメーターによるノードの割り当て方法について説明します。

FXサーバの場合 (ノード割り当てジョブに対してだけ)

rank-map-bynode パラメーターの書式は以下のとおりです。

```
--mpi "rank-map-bynode[=rankmap]"
--mpi "rank-map-bynode,rank-map-hostfile=filename"
```

ノードは以下のように割り当てられます。

1. 1つのランクにノードを割り当てると、次のランクには別のノードを割り当てます。
   割り当てるノードの順番は、*rankmap* の指定、または、rank-map-hostfile パラメーター ("2.3.6.7 rank-map-hostfile パラメーター [FX]" 参照) で指定できます。*rankmap* とrank-map-hostfile パラメーターを同時に指定した場合、rank-map-hostfile パラメーターを優先します。*rankmap* に指定する値は、"2.3.6.6 rankmap に指定するノード割り当て順序 [FX]" を参照してください。
2. すべてのランクに一通りノードを割り当てると、最初に割り当てたノードに戻ります。
3. 1、2 を繰り返して、すべてのランクにノードを割り当てます。

PRIMERGYサーバの場合 (仮想ノード割り当てジョブに対してだけ)

rank-map-bynode パラメーターの書式は以下のとおりです。

```
--mpi "rank-map-bynode"
```

仮想ノードは以下のように割り当てられます。

1. 1つのランクにノード内の仮想ノードを割り当てると、次のランクには別のノードの仮想ノードを割り当てます。
   割り当てる仮想ノードの順番は、ノードID、仮想ノードIDが小さい順です。また、割り当てるプロセスのランク番号は、仮想ノードIDに一致します。
2. 割り当てたすべてのノードにおける仮想ノードに一通り割り当てると、最初に割り当てたノードに戻り、未使用の仮想ノードをランクに割り当てます。
3. 1、2 を繰り返して、すべてのランクに仮想ノードを割り当てます。

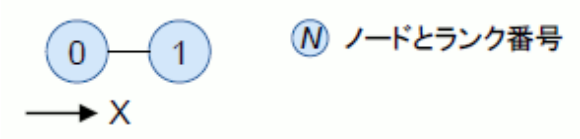
2.3.6.6 rankmap に指定するノード割り当て順序 [FX]

FXサーバで実行するノード割り当てジョブに対して、rank-map-bynode または、rank-map-bychip パラメーターで指定する rankmap は、shape パラメーターで指定したノード形状で、どの軸方向からノードをプロセスに割り当てていくかを指定するものです。指定は軸を示す文字 X、Y、Zの組み合わせで表現し、例えば2次元形状の場合は XY、3次元形状の場合は XYZ のように指定します。座標の原点をランク0とし、rankmap で指定した軸方向にランクを並べ、形状の端まで達したら次の軸に移動します。
なお、1次元形状の場合は rankmap は指定できません。ノード割り当ては、ランク0 のノードから隣接するノードに移動します。

以下に、1次元形状が指定された場合に割り当てられるノードのイメージを示します。

```
[1次元形状] --mpi "rank-map-bynode"
[1次元形状] --mpi "rank-map-bychip"
```

図2.21 1次元でのノード割り当て順序



ノード形状を1次元で指定したジョブの場合は、1次元座標に従ってノードを割り当てます。
--mpi オプションの shape パラメーターに 2、proc パラメーターに 4を指定した場合、rank-map-bynode パラメーターとrank-map-bychip パラメーターのそれぞれのプロセス割り当ては以下になります。

表2.29 1次元ノード割り当てで --mpi shape=2 proc=4 を指定した場合

割り当てノードの順番	割り当てるランク	
	rank-map-bynode の場合	rank-map-bychipの場合
0番	ランク0と2	ランク0と1
1番	ランク1と3	ランク2と3

図2.22 1次元割り当てで rank-map-bynode パラメーターを指定した場合のCPU ランク割り当て図

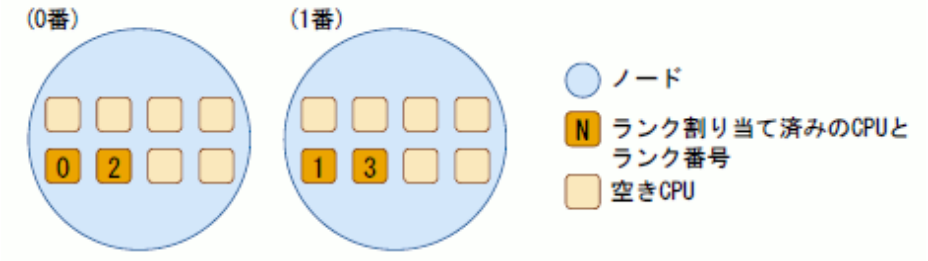
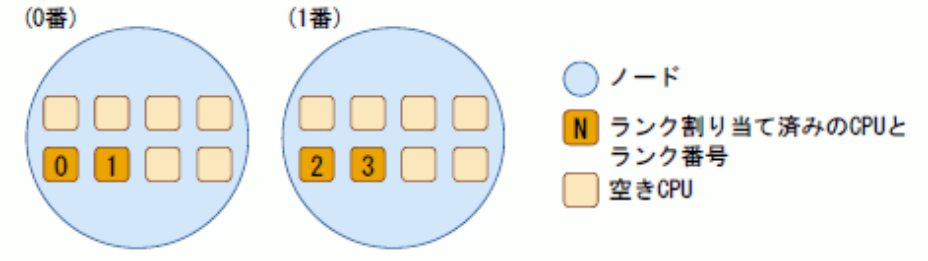


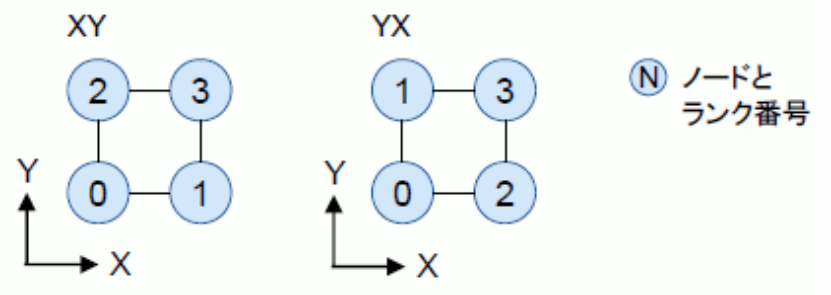
図2.23 1次元割り当てで rank-map-bychip パラメーターを指定した場合のCPU ランク割り当て図



次に、2次元形状が指定された場合に割り当てられるノードのイメージを示します。rankmap が指定されなかった場合、XY が指定されたとみなします。

```
[2次元形状] --mpi "rank-map-bynode[={XY|YX}]"
[2次元形状] --mpi "rank-map-bychip[:{XY|YX}]"
```

図2.24 2次元でのノード割り当て順序



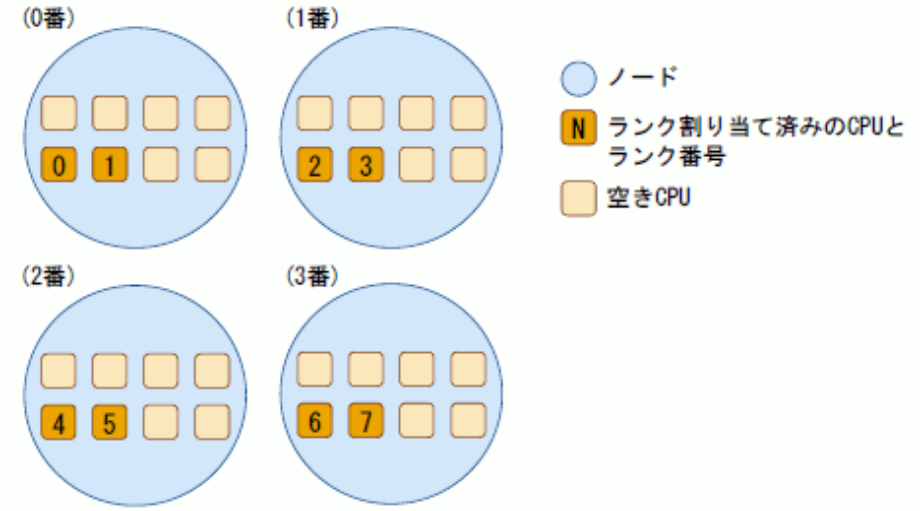
ノード形状を2次元で指定したジョブの場合は、上の図に示す順番に従ってノードを割り当てます。
--mpi オプションの shape パラメーターに 2x2、procパラメーターに8を指定した場合、rank-map-bynode パラメーターと rank-map-bychip パラメーターのそれぞれのプロセス割り当ては以下になります。

表2.30 2次元ノード割り当てで --mpi shape=2x2 proc=8 を指定した場合

割り当てノードの順番	割り当てるランク	
	rank-map-bynode の場合	rank-map-bychip の場合
0番	ランク0と4	ランク0と1
1番	ランク1と5	ランク2と3
2番	ランク2と6	ランク4と5
3番	ランク3と7	ランク6と7

図2.25 2次元割り当てで rank-map-bynode パラメーターを指定した場合のCPU ランク割り当て図

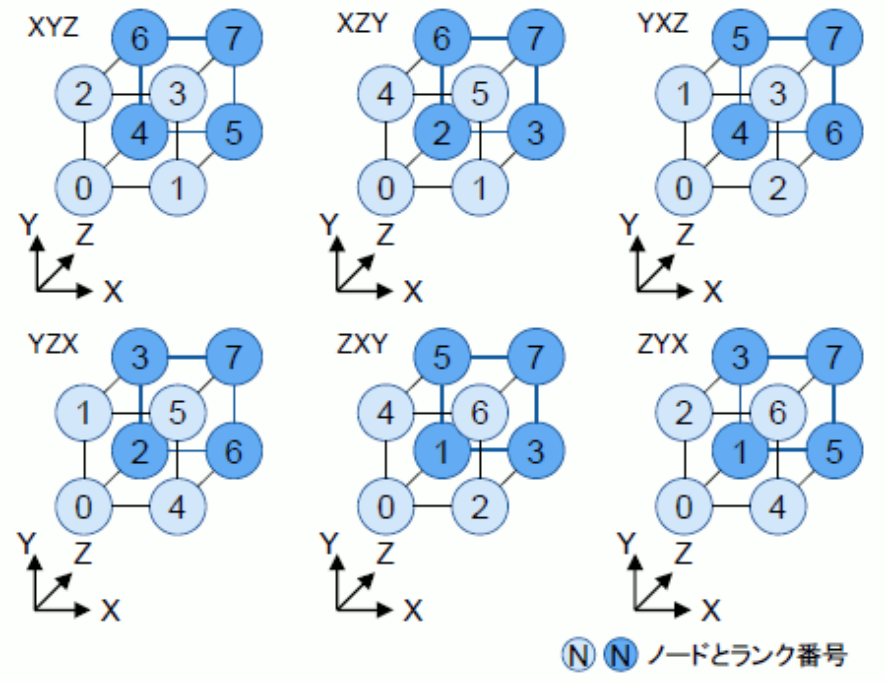
図2.26 2次元割り当てで rank-map-bychip パラメーターを指定した場合のCPU ランク割り当て図



次に、3次元形状が指定された場合に割り当てられるノードのイメージを示します。rankmap が指定されなかった場合、XYZ が指定されたとみなします。

```
[3次元形状] --mpi "rank-map-bynode[={XYZ|XZY|YXZ|YZX|ZXY|ZYX}]"
[3次元形状] --mpi "rank-map-bychip[:{XYZ|XZY|YXZ|YZX|ZXY|ZYX}]"
```

図2.27 3次元でのノード割り当て順序



ノード形状を3次元で指定したジョブの場合は、上の図に示す順番に従ってノードを割り当てます。
--mpi オプションのshape パラメーターに2x2x2、proc パラメーターに16を指定した場合、rank-map-bynode パラメーターとrank-map-bychip パラメーターのそれぞれのプロセス割り当ては以下になります。

表2.31 3次元ノード割り当てで --mpi shape=2x2x2 proc=16 を指定した場合

割り当てノードの順番	割り当てるランク	
	rank-map-bynode の場合	rank-map-bychip の場合
0番	ランク0と8	ランク0と1
1番	ランク1と9	ランク2と3
2番	ランク2と10	ランク4と5
3番	ランク3と11	ランク6と7
4番	ランク4と12	ランク8と9
5番	ランク5と13	ランク10と11
6番	ランク6と14	ランク12と13
7番	ランク7と15	ランク14と15

図2.28 3次元割り当てで rank-map-bynode パラメーターを指定した場合のCPU ランク割り当て図

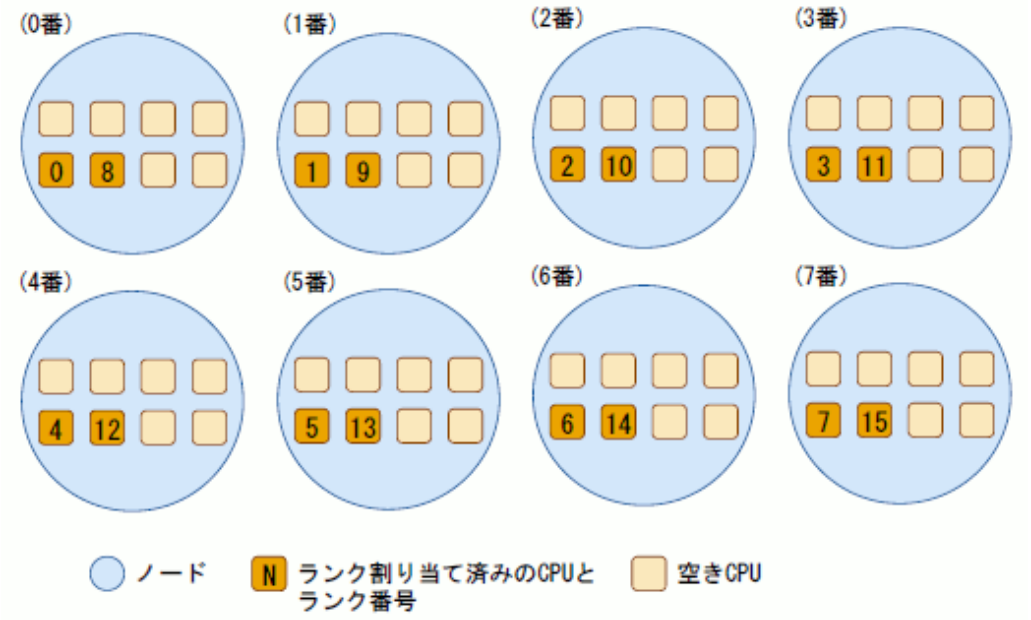
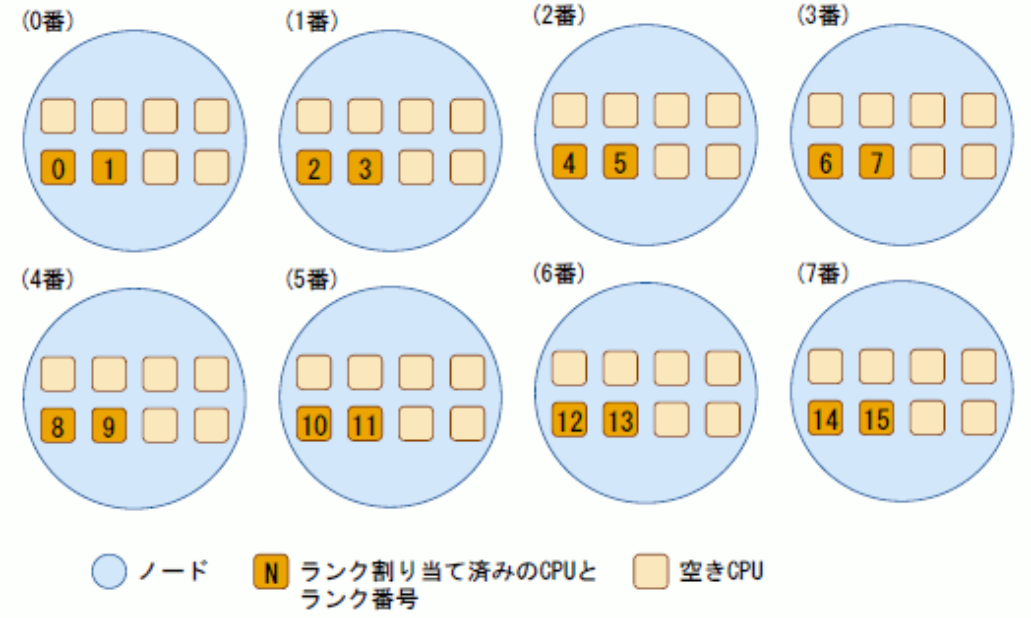


図2.29 3次元割り当てで rank-map-bychip パラメーターを指定した場合のCPU ランク割り当て図



2.3.6.7 rank-map-hostfile パラメーター [FX]

ランクに割り当てるノードの選択順序を指定したい場合は、rank-map-hostfile パラメーターを指定します。
rank-map-hostfile パラメーターの書式は以下のとおりです。

```
--mpi "rank-map-hostfile=filename"
--mpi "rank-map-bychip,rank-map-hostfile=filename"
--mpi "rank-map-bynode,rank-map-hostfile=filename"
```

MPI はランクに対して、ファイル *filename* に記述された座標のノードを割り当てます。
ノードの指定は、プロセスの形状に合わせて、1次元、2次元、または3次元の座標で指定します。
ファイル *filename* には、1 行につき 1 座標を記述し、括弧で囲みます。

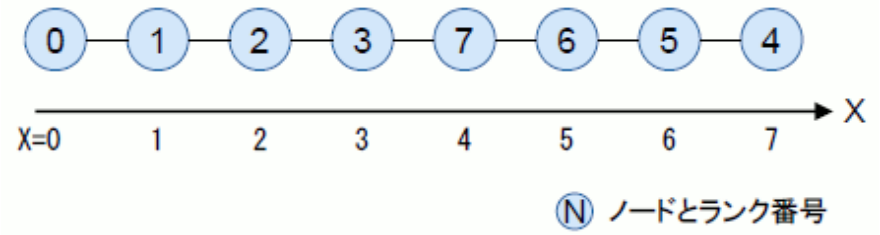
表2.32 rank-map-hostfile パラメーターの記述方式

指定する座標	記述内容
1次元座標	(X)
2次元座標	(X,Y)
3次元座標	(X,Y,Z)

ファイル *filename* にはジョブを投入するユーザーに対する読込み権が必要です。
以下に rank-map-hostfile パラメーターを使用し、1次元ノード割り当てを指定する例を示します。

```
$ cat rankmapfile-1
(0)
(1)
(2)
(3)
(7)
(6)
(5)
(4)
$ pjsub -L "node=8" --mpi "rank-map-hostfile=rankmapfile-1" job.sh
```

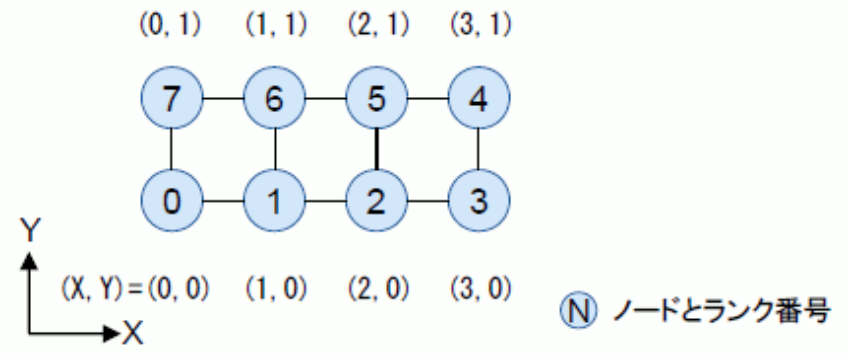
図2.30 rank-map-hostfile パラメーターを使用した1次元ノード割り当て図



次に rank-map-hostfile パラメーターを使用し、2次元ノード割り当てを指定する例を示します。

```
$ cat rankmapfile-2
(0,0)
(1,0)
(2,0)
(3,0)
(3,1)
(2,1)
(1,1)
(0,1)
$ pjsub -L "node=4x2" --mpi "rank-map-hostfile=rankmapfile-2" job.sh
```

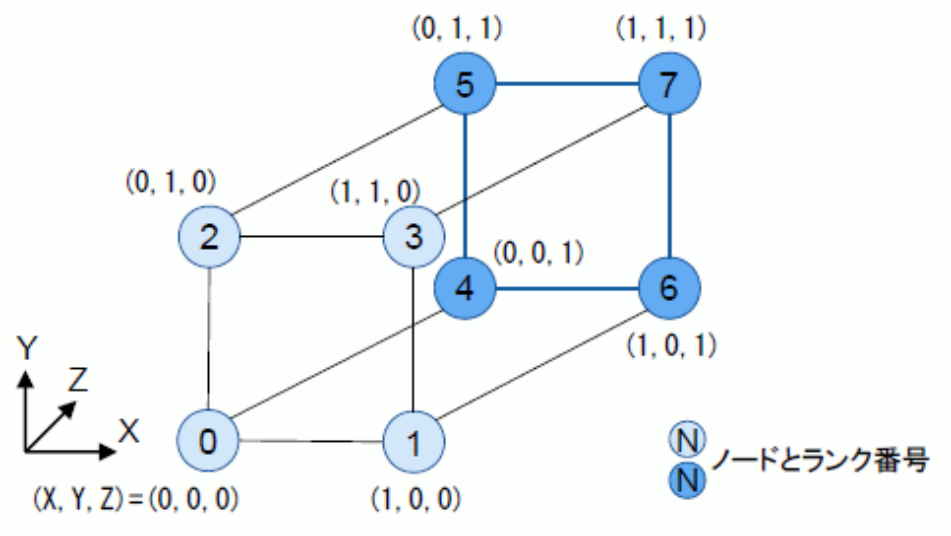
図2.31 rank-map-hostfile パラメーターを使用した2次元のノード割り当て図



次に rank-map-hostfile パラメーターを使用し、3次元ノード割り当てを指定する例を示します。

```
$ cat rankmapfile-3
(0,0,0)
(1,0,0)
(0,1,0)
(1,1,0)
(0,0,1)
(0,1,1)
(1,0,1)
(1,1,1)
$ pjsub -L "node=2x2x2" --mpi "rank-map-hostfile=rankmapfile-3" job.sh
```

図2.32 rank-map-hostfile パラメーターを使用した3次元ノード割り当て図



注意

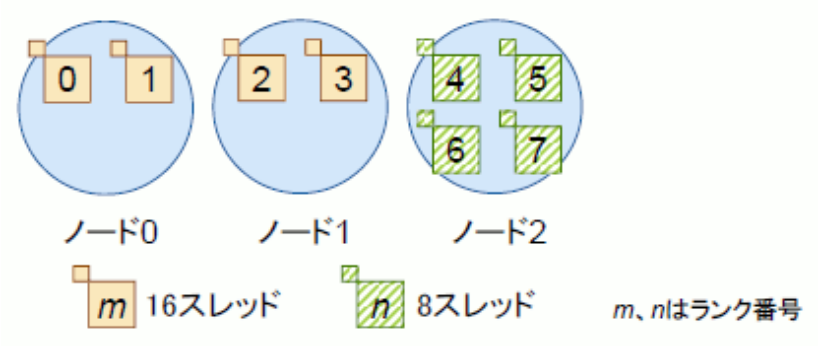
- ファイル *filename* 内の空行は、無視します。
- ファイル *filename* 内に記述する座標は、ノードの形状を表す shape パラメーターで指定した範囲内の値でなければいけません。例えば、shape=2x3 が指定された場合、記述できる座標は (0,0)、(0,1)、(0,2)、(1,0)、(1,1)、(1,2) です。
- rank-map-hostfile パラメーターを rank-map-bychip パラメーターと共に指定する場合、ファイル *filename* 内の記述は以下に従う必要があります。
  - ファイル *filename* 内に記述する座標の個数は shape パラメーターで指定された形状が示すノード数と同じにしてください。
    例えば、shape=3x2 の場合、ノード数は6台なので、ファイル *filename* 内には6つの座標を記述してください。
    記述した座標の数が shape パラメーターで指定した形状が示すノード数よりも少ない場合、pjsub コマンドはジョブの受付けを拒否します。記述した座標の数が shape パラメーターで指定した形状が示すノード数よりも多い場合、残りの座標は無視されます。
  - ファイル *filename* 内には、複数の同じ座標は記述できません。同じ座標を記述した場合は、pjsubコマンドがエラーになります。
- rank-map-hostfile パラメーターを rank-map-bynode パラメーターと共に指定する場合、ファイル *filename* 内の記述は以下に従う必要があります。
  - ファイル *filename* 内に記述した座標の個数が proc パラメーターで指定したプロセス数よりも少ない場合、最後の座標のノードまで割り当てたら最初の座標のノードに戻って割り当てます。
    座標の数が proc パラメーターで指定したプロセス数よりも多い場合、残りの座標は無視されます。
  - ファイル *filename* 内には、同じ座標は計算ノード当たりのCPUコア数以下であれば記述できます。

2.3.6.8 mpiexecコマンドの --vcoordfile オプション [FX]

ランクに対するノードの割り当ての指定方法には、前述の rank-map-hostfile パラメーターのほか、mpiexecコマンドの --vcoordfile オプションがあります。

以下は、mpiexec コマンドの --vcoordfile オプションで、各ランクに割り当てるノードの座標とコア数を指定する例です。

```
[ジョブスクリプト]
#PJM -L node=3
#PJM --mpi max-proc-per-node=4
mpiexec --vcoordfile vfile a.out

[vfile の内容]
(0) core=16    ← ランク0に座標(0)のノードの16コアを割り当て
(0) core=16    ← ランク1に座標(0)のノードの16コアを割り当て
(1) core=16    ← ランク2に座標(1)のノードの16コアを割り当て
(1) core=16    ← ランク3に座標(1)のノードの16コアを割り当て
(2) core=8     ← ランク4に座標(2)のノードの8コアを割り当て
(2) core=8     ← ランク5に座標(2)のノードの8コアを割り当て
(2) core=8     ← ランク6に座標(2)のノードの8コアを割り当て
(2) core=8     ← ランク7に座標(2)のノードの8コアを割り当て
```



参照

mpiexec コマンドの --vcoordfile オプションの詳細は、Development Studioのマニュアル「MPI使用手引書」を参照してください。

参考

pjsub コマンドの rank-map-hostfile パラメーターでは、各ランクに割り当てるノードはすべて異なる必要があります。それに対して、mpiexec コマンドの --vcoordfile オプションでは、CPUコアに空きがあれば複数のランクに対して同じノードを割り当てることができます。

また、rank-map-hostfile パラメーターはジョブ内で実行する MPI プログラムに共通の指定になりますが、--vcoordfile オプションは実行する MPI プログラムごとに (mpiexec コマンドの実行ごとに) 指定を変えることができます。
ランクに対するノードの割り当ては、rank-map-hostfile パラメーターによる指定よりも --vcoordfile オプションによる指定が優先されます。

2.3.6.9 mpiexecコマンドの標準出力/標準エラー出力 [FX]

mpiexecコマンドで実行する並列プロセスの標準出力と標準エラー出力は、出力先のファイルをオプションで指定できます。指定を省略した場合にはジョブACL機能で設定されている出力先が適用されます("[mpiexecコマンドの標準出力/標準エラー出力に関するデフォルトの動作]"を参照)。なお、mpiexecコマンドの標準出力と標準エラー出力は、それぞれジョブの標準出力と標準エラー出力です。

出力先は、mpiexecコマンド単位と並列プロセス単位が指定できます。

- mpiexecコマンド単位の出力先
  並列プロセスの出力は、指定された同じファイルに出力されます。

  — mpiexec { -of | --of | -std | --std } *stdfile*

  並列プロセスの標準出力と標準エラー出力をファイル *stdfile* に出力します。

  — mpiexec { -ofout | --ofout | -stdout | --stdout } *out_file*

  並列プロセスの標準出力をファイル *out_file* に出力します。

－ mpiexec { -oferr | --oferr | -stderr | --stderr } *err_file*

並列プロセスの標準エラー出力をファイル *err_file* に出力します。

・ 並列プロセス単位の出力先
並列プロセスの出力は、ランク(プロセス)ごとに異なるファイルに出力されます。
ファイル名は"*指定した名前.mpiexec.rank*"になります。ここで、*mpiexec*はジョブ内でのmpiexecコマンドの実行回数、*rank*は並列プロセスのランク番号を表します。

－ mpiexec { -of-proc | --of-proc | -std-proc | --std-proc } *proc_file*

並列プロセスの標準出力と標準エラー出力をランクごとにファイル *proc_file.mpiexec.rank* に出力します。

－ mpiexec { -ofout-proc | --ofout-proc | -stdout-proc | --stdout-proc } *out_proc_file*

並列プロセスの標準出力をランクごとにファイル *out_proc_file.mpiexec.rank* に出力します。

－ mpiexec { -oferr-proc | --oferr-proc |-stderr-proc | --stderr-proc } *err_proc_file*

並列プロセスの標準エラー出力をランクごとにファイル *err_proc_file.mpiexec.rank* に出力します。

ファイル名には以下のメタ文字を使用できます。

表2.33 mpiexecコマンドの出力先ファイル名で使用できるメタ文字(1)

メタ文字	意味
%j	ジョブID
%J	サブジョブID
%b	バルク番号 バルクジョブでない場合は、空文字になります。
%s	ステップ番号 ステップジョブでない場合は、空文字になります。
%n	ジョブ名
%o	ジョブの標準出力ファイル名 会話型ジョブの場合は、"./%n.%J.out"になります。
%e	ジョブの標準エラー出力ファイル名 会話型ジョブの場合は、"./%n.%J.err"になります。
%m	ジョブ内でのmpiexecコマンドの実行回数
%r	ランク番号とspawn番号 静的プロセスの場合:ランク番号 動的プロセスの場合:ランク番号@spawn番号 出力先がmpiexecコマンド単位の場合は、空文字になります。
%R	ランク番号 出力先がmpiexecコマンド単位の場合は、空文字になります。
%S	spawn番号 静的プロセスの場合:0 動的プロセスの場合:spawn番号 出力先がmpiexecコマンド単位の場合は、空文字になります。

また、メタ文字%r、%Rおよび%Sは以下の指定もできます。

表2.34 mpiexecコマンドの出力先ファイル名で使用できるメタ文字(2)

指定方法	意味	例
%0*N*r %0*N*R %0*N*S	ランク番号やspawn番号の表示文字列が最小フィールド幅*N*に満たない場合は、0でパディングします。	[ランクが3の場合] %02r : 03 (静的プロセスの場合) %02r : 03@01 (動的プロセスの場合) %02R : 03 %02S : 01

指定方法	意味	例
%/*N*r %/*N*R %/*N*S	ランク番号やspawn番号を数値*N*単位で切り捨てます。	[ランクが3の場合] %/100r : 0 (静的プロセスの場合) %/100r : 0@0 (動的プロセスの場合) %/100R : 0 %/100S : 0 [ランクが203の場合] %/100r : 200 (静的プロセスの場合) %/100r : 200@0 (動的プロセスの場合) %/100R : 200 %/100S : 0
%0*M*/Nr %0*M*/*N*R %0*M*/*N*S	ランク番号やspawn番号を数値*N*単位で切り捨てます。 切り捨てた値の表示文字列が最小フィールド幅*M*に満たない場合は、0でパディングします。	[ランクが3の場合] %04/100r : 0000 (静的プロセスの場合) %04/100r : 0000@0000 (動的プロセスの場合) %04/100R : 0000 %04/100S : 0000 [ランクが203の場合] %04/100r : 0200 (静的プロセスの場合) %04/100r : 0200@0000 (動的プロセスの場合) %04/100R : 0200 %04/100S : 0000

指定と出力ファイルの例を以下に示します。ここでは、ジョブIDは123とします。

例1) 標準出力/標準エラー出力をそれぞれ1つのファイルへ出力する。

```
mpiexec --stdout ./%j.stdout --stderr ./%j.stderr ./a.out
```

```
$ ls
123.stderr 123.stdout a.out
```

例2) ランクごとにファイルへ出力する。

```
mpiexec -stdout-proc ./%j.stdout -stderr-proc ./%j.stderr ./a.out
```

```
$ ls
123.stderr.1.0 123.stderr.1.1 123.stdout.1.0 123.stdout.1.1 a.out
```

例3) 10ランクごとにディレクトリを変えて出力する。

```
mpiexec -stdout-proc ./%/10R/%j.stdout -stderr-proc ./%/10R/%j.stderr ./a.out
```

```
$ ls
0/ 10/ a.out
$ ls 0/
123.stderr.1.0 123.stderr.1.1 ... 123.stderr.1.9
123.stdout.1.0 123.stdout.1.1 ... 123.stdout.1.9
```

例4) spawn番号ごとに分けて出力する

```
mpiexec -stdout-proc ./%S/%j.stdout -stderr-proc ./%S/%j.stderr ./a.out
```

```
$ ls
0/ 1/ a.out
$ ls 0/
123.stderr.1.0 123.stderr.1.1 ... 123.stderr.1.9
123.stdout.1.0 123.stdout.1.1 ... 123.stdout.1.9
$ ls 1/
```

```
123.stderr.1.0@1 123.stderr.1.1@1 ... 123.stderr.1.9@1
123.stdout.1.0@1 123.stdout.1.1@1 ... 123.stdout.1.9@1
```

[mpiexecコマンドの標準出力/標準エラー出力に関するデフォルトの動作]

mpiexecコマンドの標準出力/標準エラー出力に関するデフォルトの動作は、ジョブACL機能で設定されています。設定内容はpjaclコマンドで確認できます。

表2.35 mpiexecコマンドの標準出力/標準エラー出力に関するデフォルトの動作

動作	ジョブACL機能の項目	値
出力単位	mpiexec-stdouterr-unit	mpiexec : mpiexecコマンド単位 proc : 並列プロセス(ランク)単位
標準出力 (バッチジョブ)	mpiexec-stdout	*ファイル名* : 出力先ファイル。"*ファイル名*"は、メタ文字を含む場合もあります。 noset : mpiexecコマンドの標準出力/標準エラー出力 出力先を変更できないようにジョブACL機能で設定されている場合があります。ジョブACL機能の設定項目executeのmpiexec(*xxxx*)を確認してください。
標準エラー出力 (バッチジョブ)	mpiexec-stderr	
標準出力 (会話型ジョブ)	mpiexec-stdout(interact)	
標準エラー出力 (会話型ジョブ)	mpiexec-stderr(interact)	
出力がない場合の動作	mpiexec-std-emptyfile	on : 空ファイルを作成します。 off : 空ファイルは作成しません。 この動作を変更したい場合は、環境変数PLE_MPI_STD_EMPTYFILEで指定します。 例: PLE_MPI_STD_EMPTYFILE="off" 値がforce-onまたはforce-offの場合は以下の動作になります。 force-on : PLE_MPI_STD_TMPTYFILEの指定を無視し、空ファイルを作成します。 force-off : PLE_MPI_STD_TMPTYFILEの指定を無視し、空ファイルを作成しません。

参照

pjaclコマンドの出力については、"2.2.2 制限情報の確認"も参照してください。

大規模MPIジョブを実行する場合の注意

並列度が高く(おおむね10000個以上のMPIプロセスを生成)、ランクごとに標準出力や標準エラー出力をファイルへ出力するMPIジョブは、ファイルに書き出す処理のシステム負荷を考慮して、以下のように実行することを推奨します。

- mpiexecコマンドの標準出力/標準エラー出力は、ランク(プロセス)ごとに異なるファイルに出力する。
- 各ランクの標準出力/標準エラー出力ファイルは同じディレクトリに出力せず、複数のファイルごとに異なるディレクトリに出力する。
- ランクの標準出力/標準エラー出力がない場合に空ファイルを作成しない。

例:ランク番号が1000ごとに標準出力および標準エラー出力の出力先ディレクトリを変える。また、標準出力や標準エラー出力がない場合に空ファイルは作成しない。

```
export PLE_MPI_STD_EMPTYFILE="off"
mpiexec -stdout-proc ./%/1000R/%j.stdout -stderr-proc ./%/1000R/%j.stderr ./a.out
```

2.3.6.10 MPIプロセスに設定される環境変数 [FX]

MPIプロセス内では、ジョブ運用ソフトウェアによって、以下の環境変数が設定されています。

表2.36 MPIプロセスに設定される環境変数

環境変数	説明
PMIX_RANK	MPIプロセスのランク番号が10進数で指定されます。
PLE_RANK_ON_NODE	計算ノード内でのMPIプロセスの識別番号が10進数で設定されます。 識別番号とは、同じ計算ノード上の同じMPI_COMM_WORLDに属するMPIプロセスの中で一意に割り当てられる番号で、0から始まります。

2.3.7 MPIジョブの実行指定例

ここでは、ノード割り当てジョブとしてMPIジョブを実行する場合の例を説明します。

参照

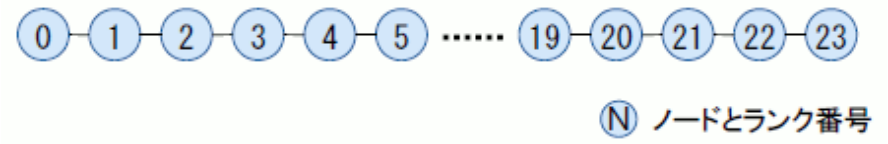
- MPI プログラムを実行する mpiexec コマンドの使用方法については、Development Studioのマニュアル「MPI 使用手引書」を参照してください。
- Development Studio以外の MPI 処理系の実行については、"付録C Development Studio以外の MPI 処理系の実行について" を参照してください。

2.3.7.1 1次元のノード形状で1つのジョブを実行する場合

以下は mpiexec コマンドで MPI プログラム prog_A を実行する例です。1次元形状のノードに、24 プロセスを連続してマッピングします。

```
$ cat job.sh
#!/bin/sh
#PJM -L "node=24"                          ← ノード形状 1次元 24 ノード
#PJM --mpi "shape=24"                      ← プロセス形状 1次元 24 ノード
...
mpiexec -n 24 ./prog_A                     ← 並列プロセス数 24 で prog_A を実行
$ pjsub job.sh
```

図2.33 1次元で1つのジョブを実行する場合



以下の指定をしても、1次元形状のノード上に、24 プロセスを連続してマッピングできます。

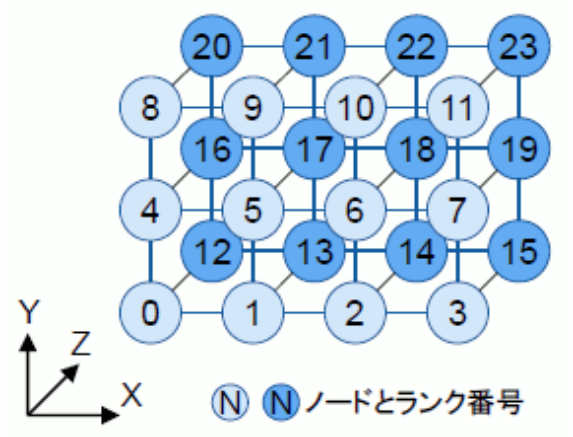
```
$ cat job.sh
#!/bin/sh
#PJM -L "node=24"                          ← ノード形状 1次元 24 ノード
...
mpiexec -n 24 ./prog_A        ← 並列プロセス数 24 で prog_A を実行。プロセス形状はノード形状 -L node=24と同じ。
$ pjsub job.sh
```

2.3.7.2 3次元のノード形状で1つのジョブを実行する場合

以下は mpiexec コマンドで MPI プログラム prog_A を実行する例です。3次元形状のノード上に、24 プロセスを連続してマッピングします。

```
$ cat job.sh
#!/bin/sh
#PJM -L "node=4x3x2"                       ← ノード形状 3次元 24 ノード
...
mpiexec ./prog_A
$ pjsub job.sh
```

図2.34 3次元で1つのジョブを実行する場合



2.3.7.3 1つのジョブでプログラムを複数回実行する場合

1 つのジョブで、MPI プログラムを複数回実行する場合の例を以下に示します。この場合、必要なノードの数に注意してください。
pjsub コマンドで指定するノード形状 (--mpi shape パラメーターの指定値、または shape パラメーターを省略した場合は、-L node パラメーターの指定値)は、最大プロセス数を指定します。
mpiexec コマンドのプロセス数指定オプション(-n、--n、-np または --np) で、最大プロセス数以下の任意の値を指定します。

```
$ cat job.sh
#!/bin/sh
#PJM -L "node=4x3x2"                          ← 3次元の 24 ノード
...
mpiexec -n 12 ./prog_A                        ← 並列プロセス数 12
mpiexec ./prog_B                              ← 並列プロセス数は生成可能な最大値(24)
mpiexec -n 16 ./prog_C                        ← 並列プロセス数 16
$ pjsub job.sh
```

最初にランク配置を決定し、指定されたプロセスまで使用します。

図2.35 ランク配置の決定

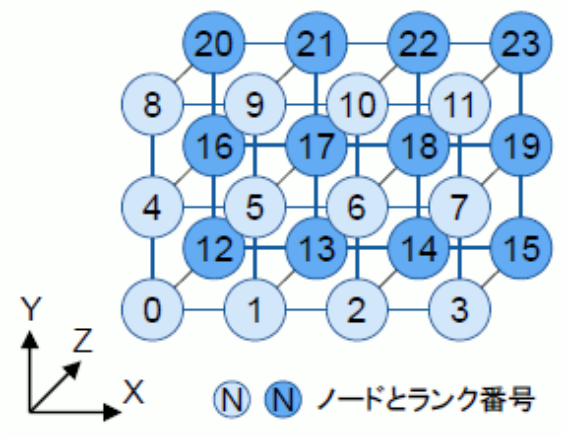
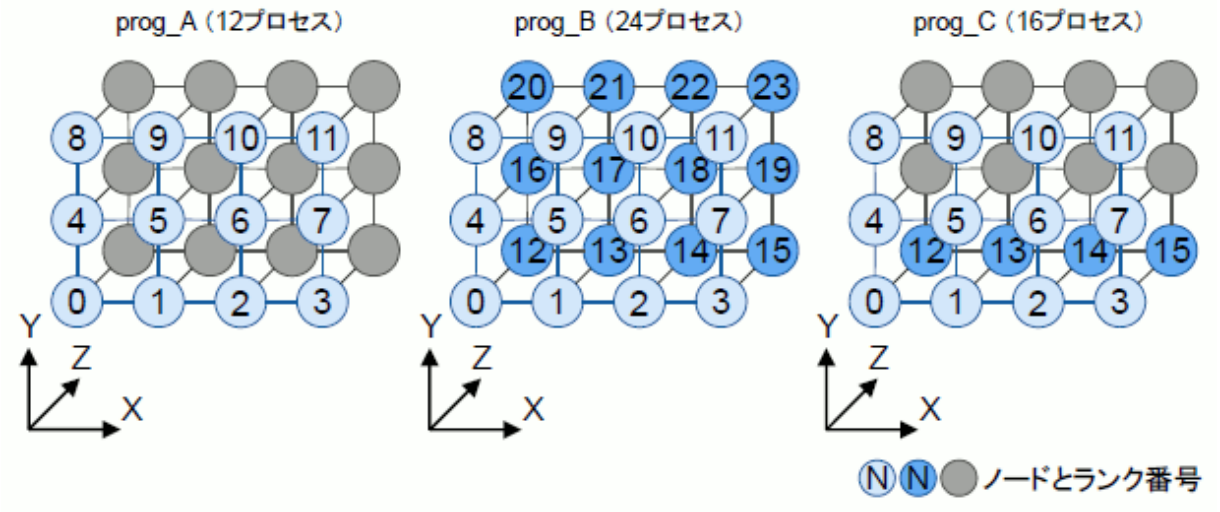


図2.36 プロセス実行例



2.3.7.4 rank-map-bynode パラメーターとrank-map-hostfile パラメーターを指定した1ノード複数プロセスジョブの実行例

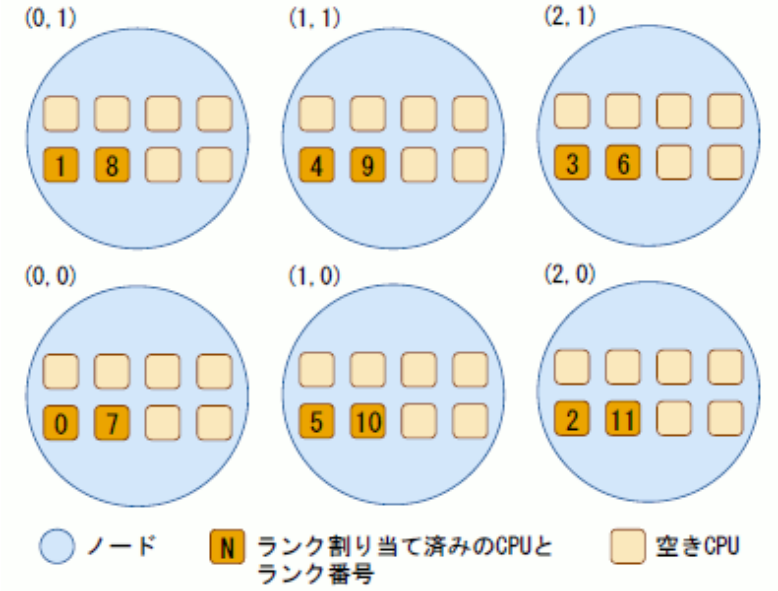
rank-map-bynode パラメーターと rank-map-hostfile パラメーターを指定し、1ノード当たり、2 つのプロセスを生成するジョブを実行する場合の例を以下に示します。

以下の例では、hostfile の先頭行の座標で示すノードをランク 0 として、1 行ごとにノードを割り当てます。

```
$ cat hostfile
(0, 0)
(0, 1)
(2, 0)
(2, 1)
(1, 1)
(1, 0)
(2, 1)
(0, 0)
(0, 1)
(1, 1)
(1, 0)
(2, 0)
$ cat job.sh
#!/bin/sh
#PJM -L "node=3x2"                                    ← 2次元の 6 ノード
#PJM --mpi "rank-map-hostfile=hostfile"
#PJM --mpi "proc=12"
#PJM --mpi "rank-map-bynode"
...
mpiexec ./prog_A
```

1 ノードに生成するプロセス数は、12 ÷ (3×2) = 2 となります。

図2.37 rank-map-bynode パラメーターと rank-map-hostfile パラメーターを指定した場合のノード割り当て順序



2.3.7.5 rank-map-bychip パラメーターとrank-map-hostfile パラメーターを指定した1ノード複数プロセスジョブの実行例

rank-map-bychip パラメーターと rank-map-hostfile パラメーターを指定し、1ノード当たり、2 つのプロセスを生成するジョブを実行する場合の例を以下に示します。

以下の例では、hostfile ファイルの先頭行の座標で示すノードをランク0として、rank-map-bychip パラメーターで指定した数ずつ割り当てます。hostfile ファイルの余った行は無視します。

```
$ cat hostfile
(0, 0)
(0, 1)
(2, 0)
(2, 1)
(1, 1)
(1, 0)
(2, 1)
(0, 0)
(0, 1)
(1, 1)
(1, 0)
(2, 0)
$ cat job.sh
#!/bin/sh
#PJM -L "node=3x2"                                    ← 2次元の 6 ノード
#PJM --mpi "rank-map-hostfile=hostfile"
#PJM --mpi "proc=12"
#PJM --mpi "rank-map-bychip"
...
mpiexec ./prog_A
$ pjsub job.sh
```

1 ノードに生成するプロセス数は、12 ÷ (3×2) = 2 となります。

図2.38 rank-map-bychip パラメーターと rank-map-hostfile パラメーターを指定した場合のノード割り当て順序

2.3.7.6 MPMD モデルの MPI プログラムの実行方法

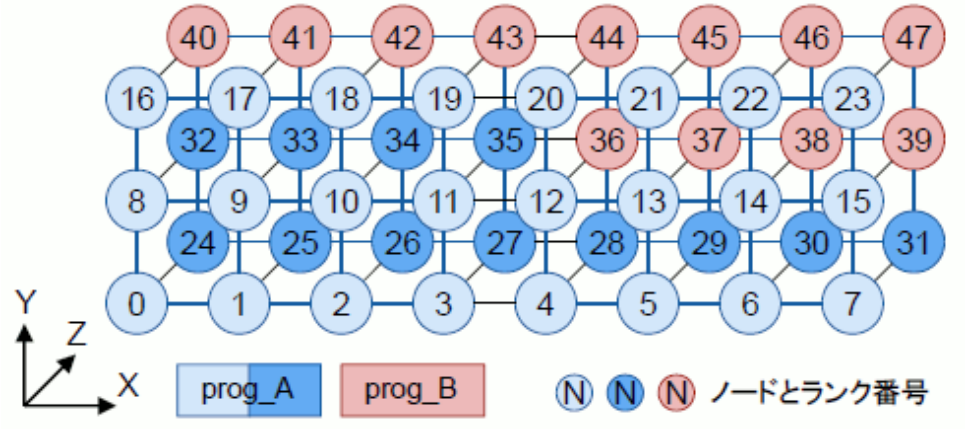
異なる複数のプログラムからなる MPI プログラム、すなわち、MPMD (Multiple Program Multiple Data) モデルの MPIプログラムを実行する場合の指定例を以下に示します。

pjsub コマンドで指定するプロセス数 (--mpi shape の指定値、shape を省略した場合は、-L node の指定値)は、各プログラムの並列度(プロセス数)の合計値を指定します。生成するプロセス数と MPI プログラム名の組み合わせはコロンで区切ります。

mpiexec のプロセス数指定オプション(-n) で、各プログラムの並列度(プロセス数)を指定します。-n オプションを指定しない場合は、-L node オプションで指定した値が渡され、結果的に合計ノード数を超える値を指定していることになり、ジョブはエラーとなります。

```
$ cat job.sh
#!/bin/sh
#PJM -L "node=8x3x2"                          ← 3次元 48 ノード。prog_A とprog_B が必要とする合計ノード数
...
mpiexec -n 36 ./prog_A : -n 12 ./prog_B       ← MPMDモデルの場合、並列プロセス数の指定オプションは省略不可
$ pjsub job.sh
```

図2.39 1 ジョブの連続実行を指定する場合



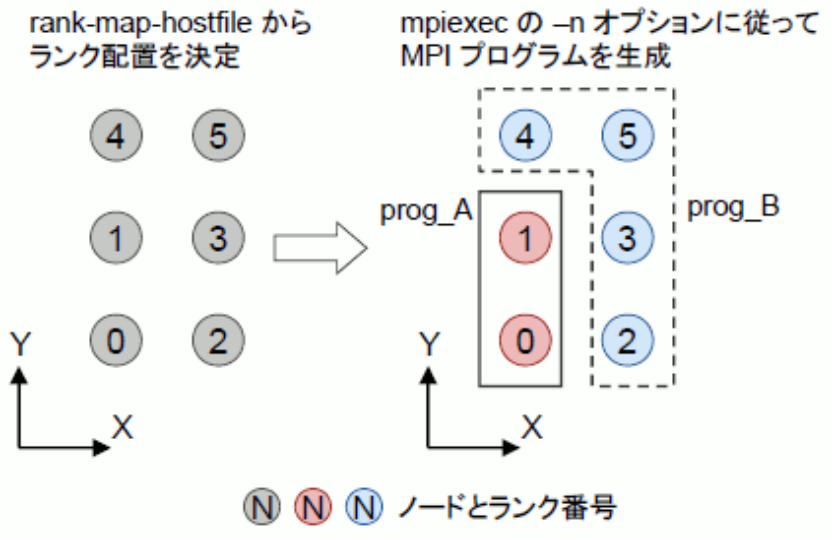
2.3.7.7 MPMD モデルの MPI プログラムに対するランク指定

pjsub コマンドの --mpi rank-map-hostfile パラメーターで MPMD モデルの MPI プログラムに対し、ランク割り当てを指定できます。mpiexec コマンドには、生成するプロセス数と MPI プログラム名の組み合わせをコロンで区切って指定します。ジョブは *filename* ファイルの先頭からプロセスを割り当てます。

以下に指定例を示します。

```
$ cat filename                                          ← ランクマップを指定するファイル
(0,0)
(0,1)
(1,0)
(1,1)
(0,2)
(1,2)
$ cat job.sh
#!/bin/sh
#PJM -L "node=2x3"
#PJM --mpi "rank-map-hostfile=filename"                 ← ランクマップの指定
...
mpiexec -n 2 ./prog_A : -n 4 ./prog_B                   ← MPMDモデルのMPIプログラム prog_A、prog_B の実行
$ pjsub job.sh
```

図2.40 MPMD プログラムの形状割り当て



2.3.7.8 同一ノード上で複数のMPIプログラムを実行する場合(静的プロセス) [FX]

ここでは、1つのジョブ内で複数のMPIプログラムを実行する例を示します。複数の MPI プログラムを実行するには mpiexec コマンドをバックグラウンド実行します。

```
#!/bin/sh
#PJM -L node=4
#PJM --mpi max-proc-per-node=8
mpiexec --vcoordfile file_a a.out &  ←MPIプログラム a.out
mpiexec --vcoordfile file_b b.out &  ←MPIプログラム b.out
mpiexec --vcoordfile file_c c.out    ←MPIプログラム c.out
```

この例では、MPIプログラム a.out、b.out、および c.out をそれぞれ mpiexec コマンドの vcoordfile オプションで指定したノードで実行します。

プロセス数が4のMPIプログラム a.out にはランクごとに4コアを割り当てます。

```
[ファイル file_a]
(0) core=4    ← a.outのランク0に座標(0)のノードの4コアを割り当て
(1) core=4    ← a.outのランク1に座標(1)のノードの4コアを割り当て
(2) core=4    ← a.outのランク2に座標(2)のノードの4コアを割り当て
(3) core=4    ← a.outのランク3に座標(3)のノードの4コアを割り当て
```

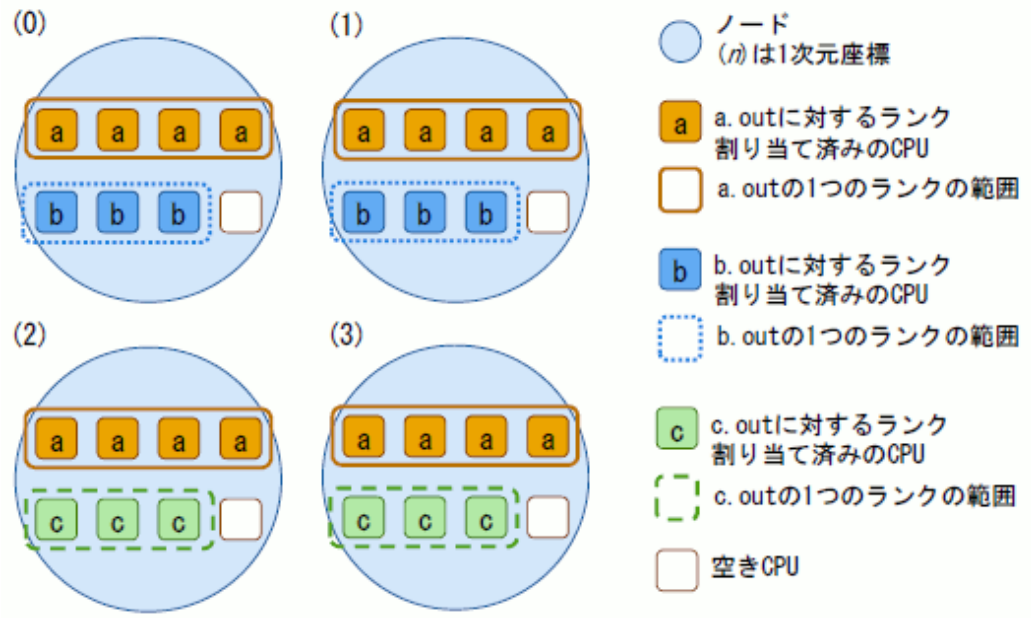
プロセス数が2のMPIプログラム b.out にはランクごとに3コアを割り当てます。

```
[ファイル file_b]
(0) core=3      ← b.outのランク0に座標(0)のノードの3コアを割り当て
(1) core=3      ← b.outのランク1に座標(1)のノードの3コアを割り当て
```

プロセス数が2のMPIプログラム c.out にはランクごとに3コアを割り当てます。

```
[ファイル file_c]
(2) core=3      ← c.outのランク0に座標(2)のノードの3コアを割り当て
(3) core=3      ← c.outのランク1に座標(3)のノードの3コアを割り当て
```

図2.41 同一ノードで複数のMPIプログラムを実行する場合のランク割り当て



2.3.7.9 同一ノード上で複数のMPIプログラムを実行する場合(動的プロセス) [FX]

ここでは、ノード上にすでにMPIプロセスが存在する状況で、さらに動的プロセスを生成する例を2つ紹介します。

まず、1つのジョブ内で静的プロセスが存在するノード上にさらに動的プロセスを生成する実行例を以下に示します。

```
[ジョブスクリプト]
#PJM -L node=2
#PJM --mpi max-proc-per-node=8
mpiexec --vcoordfile vfile_a a.out  ← 静的プロセスの生成
```

この例では、動的プロセスを生成する際に MPI_Comm_spawn 関数の infoキー:vcoordfile を利用して、動的プロセスの生成先ノードのランク配置を指定します。
プロセス数が2のMPIプログラム a.out にはランクごとに4コアを割り当てます。ファイル vfile_a には、ランクごとの割り当てノード、コア数を指定します。

```
[ファイル vfile_a]
(0) core=4     ← a.outのランク0に座標(0)のノードの4コアを割り当て
(1) core=4     ← a.outのランク1に座標(1)のノードの4コアを割り当て
```

MPIプログラム a.out のソースコード (C言語プログラム a.c) の例を以下に示します。

この例では MPI_Comm_spawn 関数を使用して動的プロセス b.out を生成します。動的プロセス b.out は プロセス数2で、ランクごとに2コアを割り当てます。次に、MPI プログラム a.out のプロセスが存在するノード上にプロセス b.out が生成されます。プロセス自身がノードの座標を取得するには、FJMPI_Topology_get_coords 関数を使用します。

参照

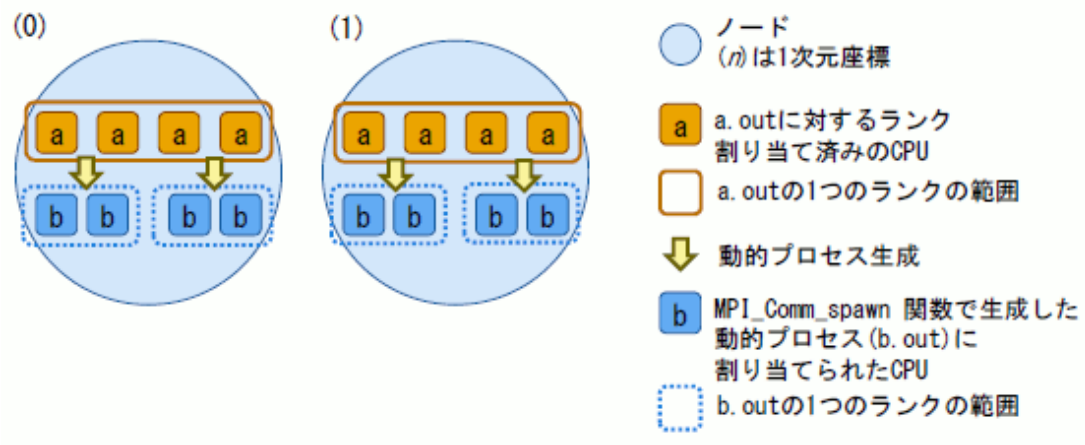
FJMPI_Topology_get_coords関数については、Development Studioのマニュアル「MPI使用手引書」を参照してください。

```
[a.c]
#include <mpi.h>
#include <mpi-ext.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    MPI_Info info;
    MPI_Comm comm;
    int rank;
    FILE *fp = NULL;
    char vfile[256];
    int coord[1];

    MPI_Init(&argc, &argv);
     ...
    // ランク番号の取得
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // 自プロセスが存在するノードの1次元座標の取得
    FJMPI_Topology_get_coords(MPI_COMM_WORLD, rank, FJMPI_LOGICAL, 1, coord);

    // vcoordfile ファイルを生成
    sprintf(vfile, "vcoordfile_%d", rank);
    fp = fopen(vfile, "w");
    ...
    // 動的プロセスの生成先ノードの1次元座標を指定
    fprintf(fp, "(%d) core=2¥n", coord[0]);
    fprintf(fp, "(%d) core=2¥n", coord[0]);
    fclose(fp);

    // infoキーの生成
    MPI_Info_create(&info);
    // 動的プロセスの生成先ノードを vcoordfileで指定
    MPI_Info_set(info, "vcoordfile", vfile);
    // 動的プロセスの生成 (b.outはプロセス数2のMPIプログラム)
    MPI_Comm_spawn("./b.out", MPI_ARGV_NULL, 2, info, 0,
    MPI_COMM_SELF, &comm, MPI_ERRCODES_IGNORE);
    ...
    MPI_Finalize();
    return 0;
}
```

MPI_Comm_spawn 関数を呼び出して MPIプロセス b.out を生成する前に作成されるファイル vcoordfile_*N*(*N*=0または1) には、ランクごとの割り当てノード、コア数が設定されます。

```
[ファイルvcoordfile_N (N=0または1)]
(N) core=2     ← b.out のランク0に座標(N)のノードの2コアを割り当て
(N) core=2     ← b.out のランク1に座標(N)のノードの2コアを割り当て
```

図2.42 静的プロセスと動的プロセスがノード内で共存する場合のランク割り当て



次に、1つのジョブ内で、動的プロセスが存在するノードでさらに別の動的プロセスを生成する例を以下に示します。

```
[ジョブスクリプト]
#PJM -L node=4
#PJM --mpi max-proc-per-node=8
mpiexec --vcoordfile vfile_a a.out  ← 静的プロセスの生成
```

この例では動的プロセスから、さらに動的プロセスを生成する際に、MPI_Comm_spawn 関数の info キー: vcoordfile を利用して、動的プロセスの生成先ノードのランク配置を指定しています。プロセス数が4のMPIプログラム a.out にはランクごとに8コアを割り当てます。ファイル file_a にランクごとの割り当てノード、コア数を指定します。

```
[ファイル vfile_a]
(0) core=8    ← a.outのランク0に座標(0)のノードの8コアを割り当て
(1) core=8    ← a.outのランク1に座標(1)のノードの8コアを割り当て
```

MPIプログラム a.out のソースコード (以下例ではCプログラム a.c) の例を以下に示します。この例ではMPI_Comm_spawn 関数を使用して動的プロセス b.out を生成します。プロセス数が2の動的プロセス b.out には、1プロセスに2コアを割り当てます。

```
[a.c]
#include <mpi.h>
#include <mpi-ext.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    MPI_Info info;
    MPI_Comm comm;
    int rank;
    FILE *fp = NULL;
    char vfile[256];
    int coord[1];

    MPI_Init(&argc, &argv);
    ...
    // ランク番号の取得
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // 自プロセスが存在するノードの1次元座標の取得
    FJMPI_Topology_get_coords(MPI_COMM_WORLD, rank, FJMPI_LOGICAL, 1, coord);

    // vcoordfile ファイルを生成
    sprintf(vfile, "vcoordfile_b_%d", rank);
    fp = fopen(vfile, "w");
    ...
    // 動的プロセスの生成先ノードの1次元座標を指定
    // a.outは1次元座標(0), (1)を専有するため、座標(2)または(3)を指定
    fprintf(fp, "(%d) core=2¥n", coord[0]+2);
```

```
    fprintf(fp, "(%d) core=2¥n", coord[0]+2);
    fclose(fp);

    // infoキーの生成
    MPI_Info_create(&info);
    // 動的プロセスの生成先ノードを vcoordfileで指定
    MPI_Info_set(info, "vcoordfile", vfile);
    // 動的プロセスの生成 (b.outはプロセス数2のMPIプログラム)
    MPI_Comm_spawn("./b.out", MPI_ARGV_NULL, 2, info, 0,
    MPI_COMM_SELF, &comm, MPI_ERRCODES_IGNORE);
    ...
    MPI_Finalize();
    return 0;
}
```

MPI_Comm_spawn 関数を呼び出して動的プロセス b.out を生成する前に作成するファイル vcoordfile_b_*N*(*N*はノードの1次元座標) には、ランクごとの割り当てノード、コア数が設定されます。

```
[ファイル vcoordfile_b_N (N はノードの1次元座標)]
(N) core=2    ← b.outのランク0に座標(N)のノードの2コアを割り当て
(N) core=2    ← b.outのランク1に座標(N)のノードの2コアを割り当て
```

動的プロセス b.out のソースコード (以下例ではCプログラム b.c) の例を以下に示します。

この例では、MPI_Comm_spawn 関数を使用して動的プロセス c.out を生成します。プロセス数が2つの動的プロセス c.out には1プロセスに1コアを割り当てます。

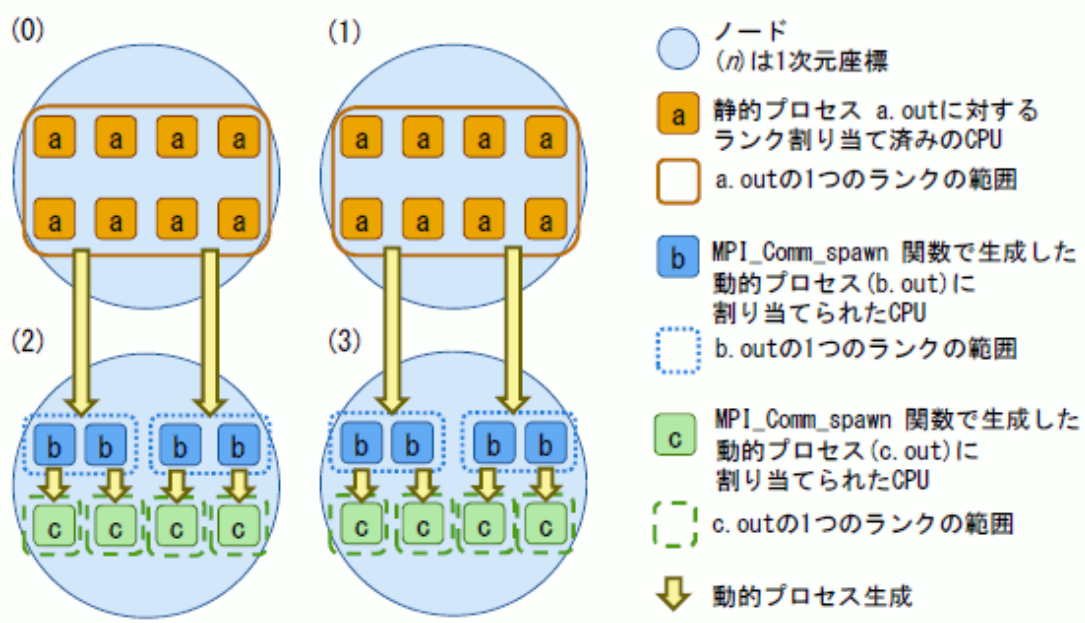
```
[b.c]
#include <mpi.h>
#include <mpi-ext.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    MPI_Info info;
    MPI_Comm comm;
    int rank;
    FILE *fp = NULL;
    char vfile[256];
    int coord[1];

    MPI_Init(&argc, &argv);
     ...
    // ランク番号の取得
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // 自プロセスが存在するノードの1次元座標の取得
    FJMPI_Topology_get_coords(MPI_COMM_WORLD, rank, FJMPI_LOGICAL, 1, coord);

    // vcoordfile ファイルを生成
    sprintf(vfile, "vcoordfile_c_%d", rank);
    fp = fopen(vfile, "w");
    ...
    // 動的プロセスの生成先ノードの1次元座標を指定
    fprintf(fp, "(%d) core=1¥n", coord[0]);
    fprintf(fp, "(%d) core=1¥n", coord[0]);
    fclose(fp);

    // infoキーの生成
    MPI_Info_create(&info);
    // 動的プロセスの生成先ノードを vcoordfileで指定
    MPI_Info_set(info, "vcoordfile", vfile);
    // 動的プロセスの生成 (c.outはプロセス数2のMPIプログラム)
    MPI_Comm_spawn("./c.out", MPI_ARGV_NULL, 2, info, 0,
    MPI_COMM_SELF, &comm, MPI_ERRCODES_IGNORE);
```

```
    ...
    MPI_Finalize();
    return 0;
}
```

MPI_Comm_spawn 関数を呼び出して動的プロセス c.out を生成する前に作成するファイル vcoordfile_c_*N*(*N*はノードの1次元座標) には、ランクごとの割り当てノード、コア数が設定されます。

```
[ファイルvcoordfile_c_N]
(N) core=1     ← c.outのランク0に座標(N)のノードの1コアを割り当て
(N) core=1     ← c.outのランク1に座標(N)のノードの1コアを割り当て
```

図2.43 複数の動的プロセスがノード内で共存する場合のランク割り当て



2.3.7.10 プログラムのリモート実行 [FX]

pjrsh コマンドで、MPIプロセスから同一ジョブ内の任意の計算ノードに対してプログラムを実行できます。プログラムの生成先は割り当てた計算ノードの形状における座標またはIPアドレスで指定します。

- 座標を指定する場合

```
pjrsh "座標" プログラム 引数
```

  "*座標*" は、以下の書式です。

  - 1次元座標: "(*x*)"
  - 2次元座標: "(*x*,*y*)"
  - 3次元座標: "(*x*,*y*,*z*)"

- IP アドレスを指定する場合

```
pjrsh IPaddress プログラム 引数
```

  IPaddress には、プログラムを実行するノードのIPアドレスを *XXX.XXX.XXX.XXX* の形式で指定します。ジョブに割り当てられた計算ノードのIPアドレスは、pjshowip コマンドで取得できます。なお、ジョブに割り当てられた計算ノード以外のノードのIPアドレスを指定すると pjrsh コマンドはエラーになります。

以降では、リモート実行時に計算ノードを座標で指定する例とIPアドレスで指定する例の2つを紹介します。

まず、MPIプログラム a.out のプロセスから、計算ノードの座標を指定してプログラムを実行する例を示します。

```
[ジョブスクリプト]
#PJM -L node=3
#PJM --mpi max-proc-per-node=4
mpiexec --vcoordfile vfile_a a.out  ← 静的プロセスの生成
```

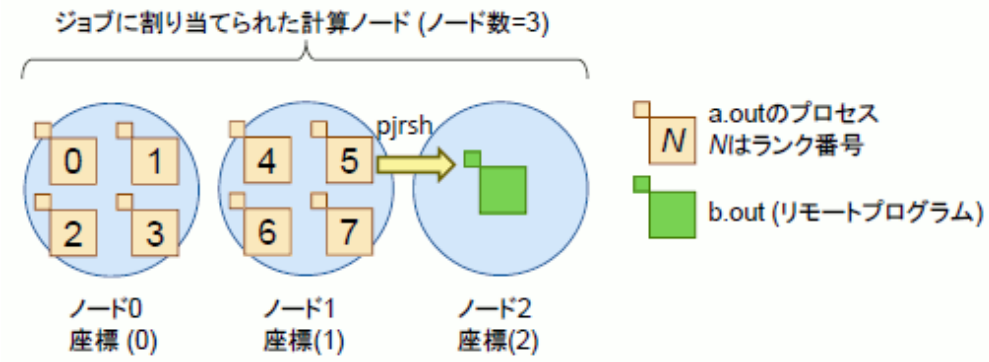
この例では、MPIプログラム a.out の起動時に生成されるプロセス(プロセス数は8)には、ファイル vfile_a でランクごとに割り当てるノードの座標を指定しています。

```
[ファイル vfile_a]
(0)       ← a.outのランク0に1次元座標(0)のノードを割り当て
(0)       ← a.outのランク1に1次元座標(0)のノードを割り当て
(0)       ← a.outのランク2に1次元座標(0)のノードを割り当て
(0)       ← a.outのランク3に1次元座標(0)のノードを割り当て
(1)       ← a.outのランク4に1次元座標(1)のノードを割り当て
(1)       ← a.outのランク5に1次元座標(1)のノードを割り当て
(1)       ← a.outのランク6に1次元座標(1)のノードを割り当て
(1)       ← a.outのランク7に1次元座標(1)のノードを割り当て
```

MPI プログラム a.out のランク5のプロセスから、座標が2のノードでプロセス b.out (MPIプログラムは除く) を生成するには、MPI プログラム a.out 内で以下のコマンドをfork(2)およびexecve(2)を使って実行します。

```
pjrsh "(2)" ./b.out
```

図2.44 プログラムのリモート実行のイメージ (1)



注意

- execve(2)の引数envpには、呼び出し元の環境変数をすべて指定してください。
  すべての環境変数を指定しない場合は、プログラムが正常に実行できない可能性があります。例えば、execve(2)の引数envpにNULLを指定すると、以下のエラーメッセージが出力され、プロセスの生成に失敗します。

  ```
  [ERR.] PLE 0002 plexec PLE service error occurred. (nid=own node) (CODE=code1, code2, code3)
  ```

- b.outがMPIプログラムの場合は、MPIプログラムa.outからMPI_Comm_spawn関数やMPI_Comm_spawn_multiple関数を使用してb.outを実行してください。

次に、MPIプログラム a.out のプロセスから、計算ノードのIPアドレスを指定してプログラムを実行する例を示します。

```
[ジョブスクリプト]
#PJM -L node=2
#PJM --mpi max-proc-per-node=4
pjshowip > /foo/bar/ipaddr.lst      ← MPIプロセスからも参照可能なファイルとする
mpiexec --vcoordfile vfile_a a.out  ← 静的プロセスの生成
```

この例では、ジョブスクリプト内で pjshowip コマンドを実行し、割り当てられている計算ノードのIPアドレスの一覧を取得してファイルに保存しています。また、MPIプログラム a.out の起動時に生成されるプロセス(プロセス数6)には、ファイル vfile_a でランクごとに割り当てるノードの座標を指定しています。

```
[ファイル vfile_a]
(0)       ← a.outのランク0に1次元座標(0)のノードを割り当て
(0)       ← a.outのランク1に1次元座標(0)のノードを割り当て
(0)       ← a.outのランク2に1次元座標(0)のノードを割り当て
(0)       ← a.outのランク3に1次元座標(0)のノードを割り当て
(1)       ← a.outのランク4に1次元座標(1)のノードを割り当て
(1)       ← a.outのランク5に1次元座標(1)のノードを割り当て
```

pjshowip コマンドの実行結果には、計算ノードのIPアドレスの一覧が出力されています。出力される計算ノードの順序は、pjsub コマンドのオプション -L node=2 および --mpi max-proc-per-node=4 の指定に従って付与されるランク番号の順です。

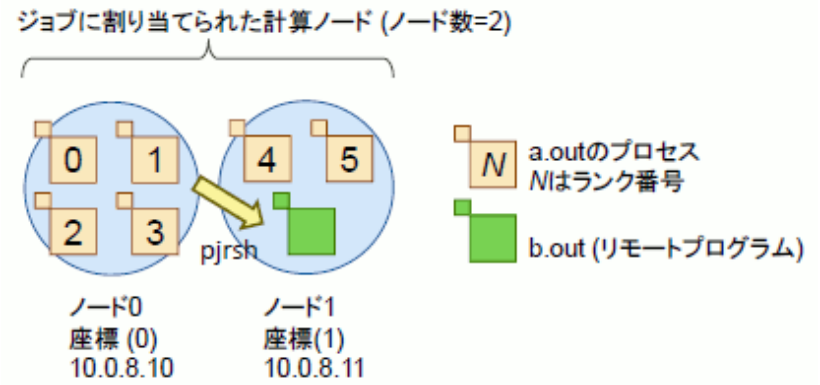
```
[ファイル ipaddr.lst]
10.0.8.10  ← ランク0の計算ノードのIPアドレス
10.0.8.10  ← ランク1の計算ノードのIPアドレス
10.0.8.10  ← ランク2の計算ノードのIPアドレス
10.0.8.10  ← ランク3の計算ノードのIPアドレス
10.0.8.11  ← ランク4の計算ノードのIPアドレス
10.0.8.11  ← ランク5の計算ノードのIPアドレス
10.0.8.11  ← ランク6の計算ノードのIPアドレス(*)
10.0.8.11  ← ランク7の計算ノードのIPアドレス(*)
```

(*) この例では、MPIプログラムa.outのプロセス数が6つなので、ランク6とランク7のプロセスは実際には生成されません。

MPIプログラム a.out のランク1のプロセスから、ランク4のプロセスが存在するノードにプロセス b.out を生成するには、pjshowip コマンドの出力結果を保存したファイル /foo/bar/ipaddr.lst からランク4が配置されるノードのIPアドレスを取得し、以下のコマンドを fork(2) および exec(2) を使って実行します。

```
pjrsh 10.0.8.11 ./b.out
```

図2.45 プログラムのリモート実行イメージ (2)



2.3.7.11 ハイブリッド並列プログラムの実行方法

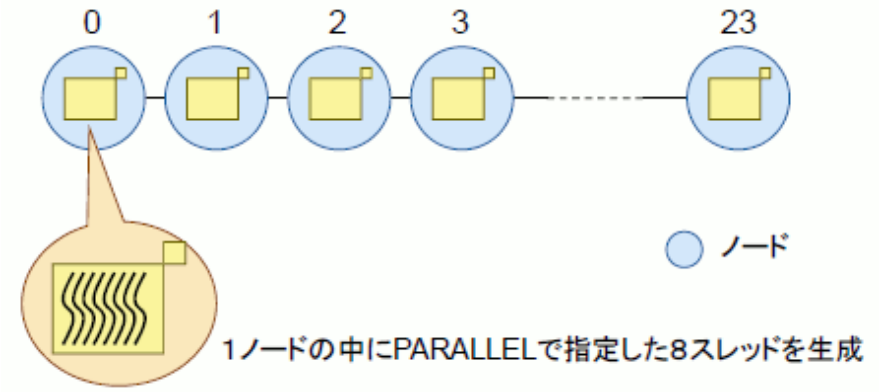
プロセス並列とスレッド並列の両方を利用するハイブリッド並列プログラムを実行する例を示します。
mpiexec コマンドの -n オプションを省略した場合は、pjsub コマンドの -L オプションで指定した値が自動的に渡されます。

```
$ cat job.sh
#!/bin/sh
#PJM -L "node=24"

export PARALLEL=8                                   ← スレッド数の指定
...
mpiexec ./prog_A                                    ← 並列プロセス数 24
$ pjsub job.sh
```

注意

上記例で、環境変数PARALLELは自動並列化におけるスレッド数の指定です。詳しくは、Development Studioのマニュアルを参照してください。

図2.46 ハイブリッド並列プログラムを生成した場合



2.3.8 ジョブの実行環境の指定

アプリケーションを実行するために、例えば特定のソフトウェアがインストールされていることが必要な場合があります。また、アプリケーションの実行をほかのユーザーが再現したい場合や、システムが更新されても従来のアプリケーションを古い環境で実行したい場合などもあります。

ジョブ運用ソフトウェアでは、デフォルトのジョブ実行環境(通常モード)のほかに、Linuxコンテナなどの仮想化技術を使用したアプリケーションが実行できるよう、ジョブの実行環境を明示的に指定してジョブ投入する環境を実現します。エンドユーザは、管理者があらかじめ用意した実行環境を指定し、その中でアプリケーションをジョブとして実行できます(SDI指定)。また、エンドユーザによる実行環境のカスタマイズもできます(UDI指定)。UDI指定については、各ジョブ実行環境の説明を参照ください。

ジョブ運用ソフトウェアでは、ジョブの実行環境はカスタム資源の一種として扱います。このため、ジョブ実行環境を利用したい場合は、ジョブの投入時に対応するカスタム資源を指定します。ジョブの実行環境のカスタム資源名は"jobenv"です。

ジョブの実行環境が複数用意されている場合、目的に応じて選択できます。この場合は、カスタム資源 jobenv の値としてジョブ実行環境の名前を指定します。

```
$ pjsub -L jobenv=container1 job1.sh       # ジョブ実行環境 container1 で job1.sh を実行
$ pjsub -L jobenv=container2 job2.sh       # ジョブ実行環境 container2 で job2.sh を実行
```

エンドユーザが利用できるジョブ実行環境名は、pjaclコマンドでジョブACL機能の設定を確認してください。また、ジョブ実行環境の名前を省略した場合 (-L jobenv)、適用されるデフォルトのジョブ実行環境についてもジョブACL機能の設定を確認してください。

```
$ pjacl
#
# JOBACL information
#
...
pjsub option parameters
    ...
    (-L/--rsc-list)                    select     default
        ...
        (jobenv=)                      container1,container2    container1
        ...
```

上記の場合、利用できるジョブ実行環境は container1とcontainer2です。ジョブ実行環境名を指定しなかった場合(pjsub -L jobenv)は、container1 を利用します。ご利用のシステムで用意されているジョブ実行環境がどのモードになるかは、管理者にお問い合わせください。

注意

ジョブ実行環境を指定すると、ジョブを実行する際に、まずジョブ実行環境の起動処理が行われます。ジョブ実行環境の起動処理に時間がかかると、会話型ジョブでは以下のように資源の割り当てがタイムアウトする場合があります。

```
$ pjsub --interact -L "jobenv=container"
[INFO] PJM 0000 pjsub Job 497 submitted.
[INFO] PJM 0081 .connected.
[INFO] PJM 0082 pjsub Interactive job 497 started.
[ERR.] PLE 0022 plexec has timed out.
[INFO] PJM 0083 pjsub Interactive job 497 completed.
```

この現象が発生した場合は、pjsubコマンドの--wait-timeオプションで資源の割り当て待ち時間を延長してください。必要な待ち時間はジョブ実行環境のイメージによって異なり、一概には決められません。いくつかの値を試すか、資源が割り当てられるまで待ち続けるunlimitedを指定してください。

以下では、各モードの利用における注意事項を説明します。

[Dockerモード]

UDI指定のジョブ実行環境にジョブを投入する場合は、管理者から通知されたUDI指定用のカスタム資源jobenvの値を使用し、環境変数でジョブ実行環境のイメージを指定します。
DockerモードにおけるUDI指定用のjobenvの値を"custom-docker"とした場合、次のように環境変数にdockerイメージを指定してジョブを投入します。

- Dockerイメージファイル(export形式) my-docker.tarから起動したコンテナインスタンス上でjob3.shを実行

```
$ pjsub -L jobenv=custom-docker -x PJM_JOBENV_DOCKER_IMAGE=/directory/my-docker.tar job3.sh
```

- Dockerリポジトリmy-docker:latestから起動したコンテナインスタンス上でjob4.shを実行

```
$ pjsub -L jobenv=custom-docker -x PJM_JOBENV_DOCKER_IMAGE=my-docker:latest job4.sh
```

なお、リポジトリは、dockerコマンドなどで使用される以下の書式で指定します。

```
[REGISTRY_HOST[:REGISTRY_PORT]/]NAME[:TAG]
```

UDI指定が可能かどうかはシステムの設定に依存します。このため、UDI指定を使用する場合は、指定方法を管理者に確認してください。

注意

- UDI指定で使用するコンテナイメージは、計算ノードから参照できる必要があります。
  - ー exportしたtar形式のファイル指定の場合は、計算ノードから参照できるユーザー領域に配置してください。
  - ー リポジトリ指定の場合は、リポジトリを登録するレジストリサーバを計算ノードからネットワーク経由でアクセスできる場所に配置してください。

  また、FXサーバで使用するコンテナイメージについては、"E.1.1 Docker モードの場合"の「参考」も参照してください。
- セキュリティ対策として、ジョブ内でのsetuidまたはsetgid付きバイナリの実行は禁止しています。suコマンドなども使用できません。
- ユーザーIDはホストOS側のユーザーIDと同一ですが、Technical Computing Suiteではユーザー IDとユーザー名のマッピングはしません。ジョブ内のプログラムでユーザー名を使用する場合は、ホストOSの/etc/passwdをバインドマウントするか、またはコンテナ内で使用するユーザー名を用意してください。
- 以下のディレクトリは、Technical Computing Suiteでマウントポイントとして使用します。このため、コンテナ内でこれらのディレクトリをマウントポイントとして使用できません。
  - ー /etc/opt/FJSVtcs
  - ー /var/opt/FJSVtcs
  - ー /usr/libexec/FJSVtcs/krm

- /var/opt/FJSVtcs/krm/sys/fs/cgroup
- /run/systemd/journal
- /dev/log

[McKernelモード]

UDI指定のジョブ実行環境にジョブを投入する場合には、管理者から通知されたUDI指定用のカスタム資源jobenvの値を使用し、環境変数でジョブ実行環境のイメージを指定します。
McKernelモードにおけるUDI指定用のjobenvの値を"custom-mckernel"とした場合、次のように環境変数にMcKernelのイメージファイルを指定してジョブを投入します。

- McKernelイメージファイルmy-mck.imgで起動したMcKernel上でjob.shを実行

```
$ pjsub -L jobenv=custom-mckernel -x PJM_JOBENV_MCKERNEL_IMAGE=/directory/my-mck.img job.sh
```

UDI指定が可能かどうかはシステムの設定に依存します。このため、UDI指定を使用する場合は、指定方法を管理者に確認してください。

McKernelモードでジョブを投入する場合は、以下の3つの指定が可能です。

- ジョブスクリプトの実行環境

  McKernelモードでは、ジョブスクリプトをホストOS上で実行するか、McKernel上で実行するかを環境変数PJM_JOBENV_MCKERNEL_JOBEXECで選択できます。
  "hostlinux"を指定した場合、ホストOS上でジョブスクリプトを実行します。"mckernel"を指定した場合、McKernel上でジョブスクリプトを実行します。環境変数を指定しない場合は、ホストOS上でジョブスクリプトを実行します。
  なお、本設定は管理者による設定が優先されるため、環境変数による指定が有効かどうかは管理者に確認してください。

  ジョブスクリプトをホストOS上で実行する場合、実行するプログラムをmcexecコマンド経由で実行する必要があります。

  参照

  McKernelの詳しい情報については、以下のURLで公開されている「McKernel User's Guide」を参照してください。

  https://ihkmckernel.readthedocs.io

  - mcexecコマンドの使い方やMcKernelでのMPIプログラムを含むプログラムの実行方法については"Running Programs"の章を参照してください。
  - McKernelの構成については"Architectural Overview"の章を参照してください。

- ホストOS上で動作するプログラムに割り当てるジョブメモリ量

  "1.9 ジョブ実行環境"で説明したように、McKernelモードでは、ジョブ投入時に指定したメモリ量をMcKernel用とホストOS上で動作するプログラム向け(ここでは"ホストOS用"と表記)に分割して割り当てます。McKernel用のジョブメモリ量は、ジョブ投入時に指定したメモリ量から、ホストOS用ジョブメモリ量を減算した量となります。ホストOS用ジョブメモリ量はデフォルトでは128MiBですが、環境変数PJM_JOBENV_MCKERNEL_JOBMEMで変更できます。ただし、128MiB以上を指定してください(下記の「注意」参照)。環境変数PJM_JOBENV_MCKERNEL_JOBMEMには、10進数のByte単位の数値を指定してください。それ以外の文字列を指定した場合、または、ジョブメモリ量の総量を超える値を指定した場合は、ジョブはジョブ終了コード(PJMコード)が 29 で終了します。
  なお、ホストOS用ジョブメモリ量のデフォルト値は管理者が変更できるため、必要であれば管理者に確認してください。

- McKernelの起動パラメーター

  McKernelモードでは、McKernelを起動するためのパラメーターを指定できます。
  起動パラメーターとは、McKernelのOSインスタンスを作成する際に設定する項目(IHK_CPUSやIHK_MEMなど)のことです。詳細は、以下のURLを参照してください。

  https://ihkmckernel.readthedocs.io/en/latest/spec/ihk.html

  起動パラメーターは、設定項目と同じ名前の環境変数としてジョブ投入時に指定します。
  起動パラメーターの指定を有効にするには、環境変数PJM_JOBENV_MCKERNEL_CUSTOM_ENV(値は空文字を除く任意の文字列)もジョブ投入時に指定してください。起動パラメーターの指定を有効にした場合、メモリとCPUは起動パラメーターの値を使って割り

当てられます。このため、起動パラメーターIHK_CPUS(割り当てるCPUコア)とIHK_MEM(割り当てるメモリ量)の指定は必須です。また、pjsubコマンドの-L node-memオプションによるメモリ量の指定は使用しないでください。

以下は、McKernelにCPUコアをCPU IDが12から23を割り当て、メモリをNUMAノード4から6GB割り当てる例です。

```
$ pjsub -L jobenv=custom-mckernel -x PJM_JOBENV_MCKERNEL_CUSTOM_ENV=1 -x IHK_CPUS="12-23" -x IHK_MEM="6G@4" job.sh
```

起動パラメーターはジョブ運用ソフトウェアではチェックせず、直接McKernelに渡されます。

注意

- UDI指定で使用するMcKernelのイメージファイルは、計算ノードから参照できる必要があります。
- McKernelモードで実行できるのはノード専有ジョブだけです。

```
$ pjsub -L node=1,jobenv=<実行環境名> ...
```

なお、FXサーバの場合、仮想ノードを割り当てるジョブ(-L vnode=*N*)は、ノード共有ジョブになるため、実行できません。

- 環境変数PJM_JOBENV_MCKERNEL_JOBEXECとPJM_JOBENV_MCKERNEL_JOBMEMは、pjsubコマンドの-xオプションで指定してください。ジョブスクリプト内でこれらの環境変数を設定しても有効にはなりません。
- ホストOS上にジョブスクリプトを起動し、mcexecコマンドによりMcKernel上にプログラムを起動する場合、Technical Computing Suiteのcpu affinity(ジョブプロセスを特定のCPUコアにバインドする機能)は反映されません。mcexecコマンドのオプションを適切に指定してください。
- pjsigコマンドで、McKernel上のプロセスに対してSIGSTOP/SIGCONTシグナルは送信できません。
- ホストOS用ジョブメモリ量(環境変数PJM_JOBENV_MCKERNEL_JOBMEM)は、128MiB以上の量を指定してください。これ以下の場合は、ジョブが正常に起動しない可能性があります。
  以下は1073741824Byte(1GiB)を指定する例です。

```
$ pjsub -x PJM_JOBENV_MCKERNEL_JOBMEM=1073741824 ...
```

特に、MPIジョブを実行する場合は、ホストOS用ジョブメモリ量はデフォルトの128MiBでは不足する可能性があります。
McKernelモードでは、McKernel上にジョブプロセスを1つ起動するたびにホストOS上にデリゲーション用プロセス(mcexec)を1つ起動します。mcexecは1プロセス当り約10～15MiBのメモリを消費します。そのため、MPIでノード内プロセス並列実行をする場合などMcKernel上に多数のプロセスを起動する場合は、ホストOS上にmcexec用のメモリを必要量割り当てる必要があります。
具体的には、"30MiB(※)+(プロセス数)×15MiB"(ただし最低128MiB)以上の量のメモリを指定してください。
例えば、ノード内8並列のMPIプログラムを実行する場合、150MiB(=30MiB+8×15MiB)以上の値を指定してください。
(※) 30MiB:ジョブスクリプトやMPIプロセス起動のために必要な基本メモリ量

ホストOS用ジョブメモリが不足した場合、ジョブは次のどれかの状態になります。この場合は、該当のジョブを削除し、ホストOS用ジョブメモリ量を増やしてジョブを再実行してください。

a. ジョブ統計情報の項目REASONに"LIMIT OVER MEMORY"が設定され、かつ項目ASSISTANT CORE MAX MEMORY SIZE (USE)に環境変数PJM_JOBENV_MCKERNEL_JOBMEMの設定値に等しい値が設定される。

b. ジョブがPJM CODE 21で終了し、ERROR状態になる。

c. ジョブが終了せず、かつジョブ統計情報の項目RETRY NUMが1以上になる。

- McKernelの起動パラメーターを指定する場合は、以下に注意してください。

  ― 必須の起動パラメーターIHK_CPUSおよびIHK_MEMを指定しない場合、ジョブはPJMコードが29で終了します。

  ― メモリ使用量が起動パラメーターIHK_MEMで指定した量を超えた場合、ジョブはPJMコードが12で終了します。

  ― 起動パラメーターでカーネル引数IHK_KARGSを指定しない場合、管理者が設定したMcKernelの起動オプションが適用されます。必要に応じて、管理者にお問合せください。

  ― Development Studioの以下の機能については、動作の保証はしません。

  - MPIライブラリの以下のMCAパラメーター

    plm_ple_memory_allocation_policy (NUMAメモリポリシーを指定する)
    plm_ple_numanode_assign_policy (NUMAノードへのCPU(コア)割り当てポリシーを指定する)

- ジョブ運用ソフトウェア連携高速化機能

これらの機能については、Development Studioのマニュアル「MPI使用手引書」、「C言語使用手引書」、「C++言語使用手引書」、および「Fortran使用手引書」を参照してください。

ー 一部のジョブ統計情報は正しく出力されません。詳細は「A.2 ジョブ統計情報の出力」の"[McKernelモードでのジョブ統計情報について]"を参照してください。

[KVMモード]

UDI指定のジョブ実行環境にジョブを投入する場合は、管理者から通知されたUDI指定用のカスタム資源jobenvの値を使用し、環境変数でジョブ実行環境のイメージを指定します。
KVMモードにおけるUDI指定用のjobenvの値を"custom-kvm"とした場合、次のように環境変数に仮想マシンイメージファイルを指定して、ホームディレクトリ配下でジョブを投入します。

- QEMUイメージファイルmy-kvm.imgで起動したKVM上でjob5.shを実行

```
$ cd ~
$ pjsub -L jobenv=custom-kvm -x PJM_JOBENV_KVM_IMAGE=/directory/my-kvm.img job5.sh
```

UDI指定が可能かどうかはシステムの設定に依存します。このため、UDI指定を使用する場合は、指定方法を管理者に確認してください。

注意

- UDI指定で使用する仮想マシンイメージファイルは、計算ノードから参照できる必要があります。
  仮想マシンイメージファイルのそのほかの要件については、"E.1.3 KVM モードの場合"を参照してください。
- KVMモードでは、ジョブ運用ソフトウェアで設定されたメモリ資源量の超過は検出されません。仮想マシン内で引き起こされたメモリ超過時の動作は、使用した仮想マシンイメージファイルの設定に依存します。
- KVMモードでは、ジョブを構成するファイル(ジョブスクリプト、アプリケーション、および入出力データファイル)の配置場所およびジョブを実行するときのカレントディレクトリは、ホームディレクトリ配下にしてください。アプリケーションが、ゲスト環境(VM)内に存在しない固有のライブラリなどのモジュールを参照する必要がある場合には、それらもホームディレクトリに置き、参照できるようにしてください。
- 仮想マシンイメージ間の通信はできません。
- KVMモードで実行できるのは、1ノードのノード専有ジョブだけです。

  ー FXサーバの場合

  ```
  $ pjsub -L node=1,jobenv=〈実行環境名〉...
  ```

  なお、FXサーバの場合、仮想ノードを割り当てるジョブ(-L vnode=*N*)は、ノード共有ジョブになるため、実行できません。

  ー PRIMERGYサーバの場合

  ```
  $ pjsub -L node=1,jobenv=〈実行環境名〉...
  $ pjsub -L vnode=1,jobenv=〈実行環境名〉-P exec-policy=simplex ...
  ```

  また、KVMモードでは、HPCタグアドレスオーバーライド機能制御コマンド(fhetbo)、電力制御、コア間ハードウェアバリアおよびセクタキャッシュは使用できません。
  HPCタグアドレスオーバーライド機能についてはマニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド HPC拡張機能編」、電力制御についてはマニュアル「ジョブ運用ソフトウェア APIユーザーズガイド Power API編」をそれぞれ参照してください。コア間ハードウェアバリアとセクタキャッシュは、Development Studioのマニュアルを参照してください。
- KVMモードでは、pjshowip、pjrsh、pjpbind、およびpjexeコマンドは使用できません。
- 仮想マシン上では、ジョブに割り当てられた個数分のCPUが0番から昇順に番号付けされ、単一のNUMAノード構成となります。
- pjsigコマンドを使用する場合、シグナルは仮想マシンにジョブを投入するために接続しているsshプロセスに対して送信されます。
- KVMモードでは、仮想マシンにssh接続してからジョブを実行します。ジョブの経過時間には、仮想マシンに接続する時間も含まれます。

- 仮想マシンイメージファイルへの書き込みを有効にしたジョブ実行環境で、同じ仮想マシンイメージファイルを指定して複数のKVMモードのジョブを投入しないでください。仮想マシンイメージファイルが壊れるなど予期しない事象が発生することがあります。

参照

ジョブ実行環境のイメージファイルを作成する手順については、"E.1 ジョブ実行環境のイメージファイルの作成" を参照してください。
なお、ジョブ実行環境を自分で構築する場合は、Linuxコンテナなどの仮想化技術の知識が必要になります。
投入したジョブが正常に実行されなかった場合は、"E.2 トラブルシューティング"を参考にして対処してください。

2.4 ジョブの状況確認

ここでは、ジョブ投入後、ジョブの実行状況などを確認する方法について説明します。

2.4.1 ジョブの一覧表示

ユーザーが投入したジョブの状態などの情報は、pjstat コマンドで確認できます。

```
$ pjstat

JOB_ID      JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE    VNODE  CORE V_MEM
238         job.sh     NM RUN user1    11/17 09:01:41  0001:00:00          12:2x3x2        -      -    -
239         bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00          12:2x3x2        -      -    -
240         step.sh    ST RUN user1    11/17 09:01:42  -                   -               -      -    -
241         job2.sh    NM RUN user1    11/17 09:01:42  0001:00:00          2               -      -    -
```

参照

表示項目の意味は"A.1 pjstat や pjstat -v の出力"を参照してください。

注意

- デフォルトでは、pjstat コマンドを実行したユーザーが参照可能なジョブだけ表示されます。pjstat コマンドの -A オプションなどでユーザーが参照できないジョブも表示対象にした場合、それらについては、一部の情報は"********"のように表示されます。参照権限がないジョブを表示しないように管理者が設定している場合もあります。
- ステップジョブとバルクジョブの状態は、複数あるサブジョブの状態を総合した状態となります。このため、ジョブの状態は必ずしも個々のサブジョブの状態と一致しません。
  特に、バルクジョブでは様々な状況が発生します。例えば、ジョブが RUNOUT 状態でも、一部のサブジョブがまだ QUEUED や RUNNING 状態の場合があります。
  ステップジョブとバルクジョブの正確な状態を知りたい場合は、pjstat コマンドの -E オプションを使用して、サブジョブの状態も表示させてください。詳細は後述の "a. サブジョブも表示する" を参照してください。
- 表示される項目は管理者が変更できるため、ご利用のシステムでは表示が上記の例とは異なる場合があります。
- pjstat コマンドの -X オプション (RUNNING 状態のジョブが動作している計算ノードID とランク番号の表示) は、マスタ・ワーカ型ジョブには対応していません。実行した場合、マスタ・ワーカ型ジョブに対しては正しい結果になりません。

以下では、pjstat コマンドによるジョブ情報の表示例を示します。

a. サブジョブも表示する

   デフォルトでは、バルクジョブやステップジョブのサブジョブは表示されません。サブジョブを表示するには -E または --expand オプションを指定します。

```
$ pjstat -E

JOB_ID                  JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM  ... (*)
238                     job.sh     NM RUN user1    11/17 09:01:41  0001:00:00  ...
239                     bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00  ...
239[1]                  bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00  ...
239[2]                  bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00  ...
239[3]                  bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00  ...
239[4]                  bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00  ...
239[5]                  bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00  ...
240                     step.sh    ST RUN user1    11/17 09:01:42  -            ...
240_0                   step.sh    ST RUN user1    11/17 09:01:42  0001:00:00  ...
240_1                   step.sh    ST QUE user1    -               0001:00:00  ...
240_2                   step.sh    ST QUE user1    -               0001:00:00  ...
240_3                   step.sh    ST QUE user1    -               0001:00:00  ...
240_4                   step.sh    ST QUE user1    -               0001:00:00  ...
241                     job2.sh    NM RUN user1    11/17 09:01:42  0001:00:00   ...
(*) 紙面の都合上、右側の表示は省略しています。
```

上記の例では、バルクジョブ(ジョブID 239)は、サブジョブが5つあり(サブジョブID 239[1]から239[5])、すべてのサブジョブが実行中であることを示します。また、ステップジョブ (ジョブID 240) は、サブジョブが5つあり (サブジョブID 240_0 から 240_4)、サブジョブID 240_0 のサブジョブが実行中であることを示します。

参考

バルクジョブとステップジョブには、ジョブIDに対応する情報とサブジョブIDに対応する情報があります。本書では、前者を特にバルクジョブまたはステップジョブのサマリ情報と呼びます。
上記の例では、ジョブID 239 の行がバルクジョブのサマリ情報、ジョブID 240 の行がステップジョブのサマリ情報になります。

b. ジョブ数のサマリを表示する

--with-summary オプションを指定すると、pjstat コマンド実行者が投入したジョブの数を状態別にサマリ情報として表示します (-A オプションも指定している場合は、全ユーザーのジョブが対象)。

```
$ pjstat --with-summary

  ACCEPT QUEUED RUNING RUNOUT   HOLD  ERROR SUSPND REJECT   EXIT CANCEL   TOTAL    ← ジョブ数のサマリ
       0      0      4      0      0      0      0      0      0      0       4
s      0      4      8      0      0      0      0      0      0      0      12

JOB_ID      JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE  ... (*)
238         job.sh     NM RUN user1    11/17 09:01:41  0001:00:00          12:2x3x2      ...
239         bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00          12:2x3x2      ...
240         step.sh    ST RUN user1    11/17 09:01:42  -                   -             ...
241         job2.sh    NM RUN user1    11/17 09:01:42  0001:00:00          2             ...
(*) 紙面の都合上、右側の表示は省略しています。
```

ジョブ数のサマリは2行表示されます。上段では、バルクジョブおよびステップジョブについてはジョブ単位で集計します。下段(行頭に"s"がある行)では、バルクジョブおよびステップジョブについてはそれらのサブジョブ数を集計します。

参考

上記例では前述の箇条書きaに示すように、通常ジョブが2つ、バルクジョブが1つ (サブジョブは5つ)、およびステップジョブが1つ (サブジョブは5つ) 投入されているとします。

- ジョブ数のサマリの上段
  通常ジョブが2つ 、バルクジョブが1つ、およびステップジョブが1つなので、合計は4になります。
- ジョブ数のサマリの下段
  通常ジョブが2つ、バルクジョブのサブジョブが5つ、およびステップジョブのサブジョブが5つなので、合計は12になります。

c. 特定のジョブについてだけ表示する

pjstat コマンドにジョブ IDまたはサブジョブID を指定すると、そのジョブについてだけ表示します。ジョブID は複数指定できます。

```
$ pjstat 238

JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
238        job.sh     NM RUN user1    11/17 09:01:41  0001:00:00           12:2x3x2      ...

$ pjstat '239[1]'

JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
239[1]     bulk.sh    BU RUN user1    11/17 09:01:42  0001:00:00           12:2x3x2      ...
(*) 紙面の都合上、右側の表示は省略しています。
```

pjstat コマンドの引数にバルクジョブのサブジョブIDを指定する場合、括弧'[]'がシェルによって処理されないようにシングルクォートなどでエスケープしてください。

ジョブID やサブジョブID は複数指定でき、列挙する方式と範囲を指定する方式があります。

ー 列挙

```
$ pjstat 238 239                                          ← 複数のジョブID
$ pjstat '239[1]' '239[2]' '239[3]' '239[4]' '239[5]'     ← 複数のサブジョブID(バルクジョブ)
$ pjstat 240_0 240_1 240_2 240_3 240_4                    ← 複数のサブジョブID(ステップジョブ)
```

ー 範囲の指定

ジョブID または バルク番号、ステップ番号を ハイフンでつなげると、範囲指定となります。

```
$ pjstat 238-240          (ジョブID 238、239、240)
$ pjstat '239[1-5]'       (サブジョブID 239[1]、239[2]、239[3]、239[4]、239[5])
$ pjstat 240_0-4          (サブジョブID 240_0、240_1、240_2、240_3、240_4)
```

d. より詳しいジョブの情報を表示する

pjstat コマンドの -v オプションを指定すると、詳細表示部に各ジョブのより詳しい情報を表示できます。

```
$ pjstat -v

JOB_ID     JOB_NAME   MD ST  USER     GROUP    START_DATE      ELAPSE_TIM ELAPSE_LIM
(*)NODE_REQUIRE    VNODE  CORE V_MEM        V_POL E_POL RANK      LST EC  PC  SN PRI ACCEPT
(*)RSC_GRP REASON
238        job.sh     NM RUN user1    testgrp  11/17 09:01:41  0000:09:11 0001:00:00
(*)12:2x3x2        -      -    -           -     -     -         RNA 0   0   0  127 11/17 09:01:41
(*)rg001   -
239        bulk.sh    BU RUN user1    testgrp  11/17 09:01:42  -          0001:00:00
(*)12:2x3x2        -      -    -           -     -     -         QUE -   -   -  127 11/17 09:01:41
(*)rg001   -
240        step.sh    ST RUN user1    testgrp  11/17 09:01:42  -          -
(*)-               -      -    -           -     -     -         -   -   -   -  -   -
(*)-       -
241        job2.sh    NM RUN user1    testgrp  11/17 09:01:42  0000:09:10 0001:00:00
(*)2               -      -    -           -     -     -         RNA 0   0   0  127 11/17 09:01:41
(*)rg001   -

(*) 紙面の都合上、改行していますが、実際には、この行は直前の行の継続行として表示されます。
```

参照

-v オプション指定時に表示される項目の詳細については、"A.1 pjstat や pjstat -v の出力" を参照してください。

e. 特定の項目だけ表示する

pjstat コマンドの --choose オプションを使用して、特定の項目だけ表示できます。--choose オプションの引数に表示したい項目名を列挙すると、その順番に情報が表示されます。指定できる項目名はmanマニュアルpjstat(1)を参照してください。

以下は、ジョブID、ジョブ名、ジョブモデル、およびジョブ実行開始時刻を表示する例です。

```
$ pjstat --choose jid,jnam,jmdl,sdt

JOB_ID      JOB_NAME    MD START_DATE
238         job.sh      NM 11/17 09:01:41
239         bulk.sh     BU 11/17 09:01:42
240         step.sh     ST 11/17 09:01:42
241         job2.sh     NM 11/17 09:01:42
```

注意

- 項目によっては表示しないように管理者が設定している場合があります。それらを --choose オプションで指定しても情報は表示されません。
- item名 vpol、epol、rankm、および vnid は --choose オプションにだけ指定できます。

f. 特定の状況のジョブだけ表示する

pjstat コマンドの --filter オプションを使用して、指定した条件を満たすジョブだけ表示対象にできます。
--filter オプションでは以下の条件指定ができます。指定できる項目名はpjstatコマンドのmanマニュアルを参照してください。

条件	書式	説明
値指定	--filter "*項目=値*"	項目が指定した値と一致した場合に表示対象とします。 例) --filter "jmdl=NM" ジョブモデル(jmdl)が通常ジョブ(NM)のものを表示 例) --filter "st=RNA+RUN" 状態が RUNNING-A または RUNNING のジョブを表示 注: "+" 記号は、OR 条件を示します。"+" 記号を単なる文字として表現する場合はバックスラッシュを付加してください("¥+")。
範囲指定	--filter "*項目=範囲*"	項目が指定した範囲に含まれる場合に表示対象とします。 例) --filter "jid=1-10" ジョブIDが 1 から 10 までのジョブを表示 例) --filter "jid=10-" ジョブIDが 10 以上のジョブを表示 例) --filter "jid=-20" ジョブIDが 20 以下のジョブを表示 例) --filter "jid=1-10+21-30" ジョブIDが 1 以上 10 以下、または 21 以上 30 以下のジョブを表示
文字列パターン指定	--filter "*項目=文字列パターン*"	項目が指定した文字列パターンに一致する文字列の場合に表示対象とします。 文字列パターンは以下の2種類が使用できます。 . (ピリオド) : 改行文字以外の任意の1文字 * (アスタリスク) : 改行文字以外の0文字以上の任意の文字列

条件	書式	説明
		例) --filter "jnam=jobA*" ジョブ名が "jobA" で始まるジョブを表示 例) --filter "jnam=job." ジョブ名が "job" とそれに続く1文字で構成されるジョブを表示 例) --filter "jnam=jobA*+jobB*" ジョブ名が "jobA" または "jobB" で始まるジョブを表示

注意

-A オプションまたは --all オプションを指定した場合、ほかのユーザーのジョブも表示対象になりますが、ユーザー名のように隠される情報を --filter オプションに指定しても無視されます。

g. リソースユニットまたはリソースグループ単位で表示する

--rscunit (--ru) オプションや --rscgrp オプション (--rg) を使用することで、リソースユニットまたはリソースグループ単位で表示できます。

以下のようなジョブが投入されている場合の表示例を示します。

```
ジョブID 2927 : リソースユニット：unit1、リソースグループ：group1 で実行
ジョブID 2928 : リソースユニット：unit1、リソースグループ：group2 で実行
ジョブID 2929 : リソースユニット：unit2、リソースグループ：group1 で実行
```

― リソースユニット単位で表示する場合

--rscunit (--ru) オプションを引数なしで指定します。--rscgrp (--rg) オプションは指定しません。

```
$ pjstat --rscunit

[ RSCUNIT: unit1 ]
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
2927       jobA       NM RUN user1    08/06 10:41:45  0001:00:00           2             ...
2928       jobB       NM RUN user1    08/06 10:42:03  0001:00:00           2             ...

[ RSCUNIT: unit2 ]
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
2929       jobC       NM RUN user1    08/06 10:43:21  0001:00:00           2             ...
(*) 紙面の都合上、右側の表示は省略しています。
```

― リソースユニットおよびリソースグループ単位で表示する場合

--rscgrp (--rg) オプションに加えて、--rscunit (--ru) オプションも引数なしで指定します。

```
$ pjstat --rscunit --rscgrp

[ RSCUNIT: unit1 ]
[ RSCGRP: group1 ]
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
2927       jobA       NM RUN user1    08/06 10:41:45  0001:00:00           2             ...

[ RSCUNIT: unit1 ]
[ RSCGRP: group2 ]
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
2928       jobB       NM RUN user1    08/06 10:42:03  0001:00:00           2             ...

[ RSCUNIT: unit2 ]
[ RSCGRP: group1 ]
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE  ... (*)
2929       jobC       NM RUN user1    08/06 10:43:21  0001:00:00           2             ...
(*) 紙面の都合上、右側の表示は省略しています。
```

参考

--data オプションを指定することで表示データを列挙形式で出力できます。このオプションとともに --rscunit (--ru) オプションや --rscgrp (--rg) オプションを指定すると、以下の下線部分のように RSCUNIT (リソースユニット) や RSCGRP (リソースグループ) フィールドが追加されます。これにより、出力されるレコードがどのリソースユニットまたはリソースグループの内容かを判断できます。

```
$ pjstat --rscunit unit1 --rscgrp group1 --data
H,RSCUNIT,RSCGRP,,JOB_ID,JOB_NAME,MD,ST,USER,START_DATE,ELAPSE_LIM,NODE_REQUIRE,VNODE,CORE,V_MEM
,unit1,group1,2927,jobA,NM,RUN,user1,  08/06 10:41:45,0001:00:00,2,-,-,-
```

pjstat コマンドのオプションは上記以外にもあります。詳細は pjstat コマンドの man マニュアルを参照してください。

2.4.2 ジョブ統計情報の表示

実行中のジョブのジョブ統計情報を pjstat コマンドの -s または -S オプションで表示できます。-S オプションは -s オプションよりも詳細なジョブ統計情報を表示します。

以下は、ジョブ統計情報を表示する例です。

```
$ pjstat -s

JOB ID                : 9417
JOB NAME              : script.sh
JOB TYPE              : BATCH
JOB MODEL             : NM
RETRY NUM             : 0
SUB JOB NUM           : -
USER                  : user
GROUP                 : group
RESOURCE UNIT         : rsc_unit
RESOURCE GROUP        : rsc_grp
APRIORITY             : 127
PRIORITY              : 127
SHELL                 : /bin/sh
COMMENT               :
...
```

注意

- ジョブIDを指定しない場合、pjstat コマンドを実行したユーザーのすべてのジョブについてジョブ統計情報が表示されます。ジョブ統計情報は表示される内容が多いため、大量にジョブがある場合は注意してください。
- 実行中のジョブのジョブ統計情報には、ジョブ開始時に実行されるプロローグ処理の情報が加算されています。
  ただし、実行中のジョブの実行経過時間にプロローグ処理が含まれるかどうかはシステムの設定によります。詳細は管理者にお問い合わせください。
- 実行中のジョブのジョブ統計情報のうち、資源使用状況に関する情報(メモリ使用量やCPU使用時間など)は更新が最大で10分程度遅れる場合があります。
- ジョブ統計情報は情報量が多いため、システム内に保存するジョブ統計情報を管理者が一部に限定している場合があります。pjstat コマンドの -s や -S オプションで表示されるのは、管理者が保存するように設定した情報だけです。
- FXサーバには、ジョブに割り当てた CPU コア以外に、アシスタントコアと呼ぶ CPU コアがあります。アシスタントコアは、ジョブの実行性能を劣化させる OS の割り込み処理やデーモンの処理などを受け持ちます。
  ジョブでは MPI 非同期通信処理をこのアシスタントコア上で実行します。このため、FXサーバで実行中のジョブのジョブ統計情報には、ジョブに割り当てた資源以外に、MPI 非同期通信処理に関するアシスタントコアの使用情報も出力されます。

参照

表示されるジョブ統計情報は、man マニュアルpjstatsinfo(7)を参照してください。

2.4.3 ジョブのノード利用状況の表示

ノードで実行されているジョブを確認するには、pjshowrsc コマンドの -v 1 オプションを使用します。

```
$ pjshowrsc -c cluster -v 1
[ CLST: cluster ]
[ NODE: 0x01010010 ]
      RSC      TOTAL     FREE    ALLOC
      cpu         32        0       32
      mem       57Gi        0     57Gi

 RUNNING_JOBS: 1022                                  ← ノード 0x01010010 で実行中のジョブのジョブID

[ NODE: 0x0101011 ]
      RSC      TOTAL     FREE    ALLOC
      cpu         32       16       16
      mem       57Gi     30Gi     27Gi

 RUNNING_JOBS: 2551                                  ← ノード 0x0101011 で実行中のジョブのジョブID

[ NODE: 0x0101012 ]
      RSC      TOTAL     FREE    ALLOC
      cpu         32       16       16
      mem       57Gi     30Gi     27Gi

 RUNNING_JOBS: 2551                                  ← ノード 0x0101012 で実行中のジョブのジョブID
```

注意

"RUNNING_JOBS" 行にほかのユーザーのジョブを表示できないように管理者がアクセス制限をしている場合があります。

pjshowrsc コマンドの -v 2 オプションを使用すると、ノードで実行中のジョブだけでなく、そのノードにおけるカスタム資源の利用状況も確認できます。

```
$ pjshowrsc -c clst1 -v 2
[ CLST: clst1 ]
CUSTOM_RESOURCE                                     ← 全体のカスタム資源の利用状況
RSCNAME                        TOTAL       FREE      ALLOC
customrscname1               1000000     300000     700000
customrscname2                     5          5          0
customrscname3/type1       unlimited  unlimited          2
customrscname3/type2       unlimited  unlimited          1

[ NODE: 0x01010010 ]
      RSC      TOTAL     FREE    ALLOC
      cpu         32        0       32
      mem       57Gi        0     57Gi

 RUNNING_JOBS: 1022                                  ← ノード 0x01010010 で実行中のジョブのジョブID

CUSTOM RESOURCE(PER NODE)                           ← ノード 0x01010010 のカスタム資源の利用状況
RSCNAME                      TOTAL         FREE      ALLOC
customrscname4                   1            0          1

 [ NODE: 0x0101011 ]
      RSC      TOTAL     FREE    ALLOC
```

```
      cpu         32      16      16
      mem       57Gi    30Gi    27Gi

 RUNNING_JOBS: 2551                                  ← ノード 0x01010011 で実行中のジョブのジョブID

CUSTOM RESOURCE(PER NODE)                            ← ノード 0x0101011 のカスタム資源の利用状況
RSCNAME                      TOTAL          FREE      ALLOC
customrscname4                   1             0          1

 [ NODE: 0x0101012 ]
      RSC      TOTAL    FREE   ALLOC
      cpu         32      16      16
      mem       57Gi    30Gi    27Gi

 RUNNING_JOBS: 2552                                  ← ノード 0x01010012 で実行中のジョブのジョブID

CUSTOM RESOURCE(PER NODE)                            ← ノード 0x0101012 のカスタム資源の利用状況
RSCNAME                      TOTAL          FREE      ALLOC
customrscname4                   1             0          1
```

(*1) *customrscname1* から *customrscname4* には、定義されているカスタム資源名が表示されます。
(*2) 種別で定義されているカスタム資源の場合は、カスタム資源名/種別 *customrscname*/*type* の形式でカスタム資源名が表示されます。
また、利用状況の TOTAL (カスタム資源の総数 を表示) と FREE (ジョブに割り当てられていないカスタム資源の総数を表示) には、unlimited が表示されます。

pjshowrsc コマンドの -v 3 オプションを使用すると、ノードで実行中のジョブおよびそのノードにおけるカスタム資源の利用状況だけでなく、ノードを通信経路として使用しているジョブも確認できます。

```
$ pjshowrsc -c cluster -v 3
[ CLST: cluster ]
[ NODE: 0x01010010 ]
      RSC      TOTAL    FREE   ALLOC
      cpu         32       0      32
      mem       57Gi       0    57Gi

 RUNNING_JOBS: 1022                                  ← ノード 0x01010010 で実行中のジョブのジョブID

 CUSTOM RESOURCE(PER NODE)                           ← ノード 0x01010010 のカスタム資源の利用状況
 RSCNAME                      TOTAL          FREE      ALLOC
 customrscname4                   1             0          1

 JOBS_USING_ROUTE: 1030,1031,1032,1033,1034          ← ノードを通信経路として使用しているジョブのジョブID

[ NODE: 0x0101011 ]
      RSC      TOTAL    FREE   ALLOC
      cpu         32      16      16
      mem       57Gi    30Gi    27Gi

 RUNNING_JOBS: 2551                                  ← ノード 0x01010011 で実行中のジョブのジョブID

 CUSTOM RESOURCE(PER NODE)                           ← ノード 0x0101011 のカスタム資源の利用状況
 RSCNAME                      TOTAL          FREE      ALLOC
 customrscname4                   1             0          1

 JOBS_USING_ROUTE: 1030,1031,1032,1033,1034          ← ノードを通信経路として使用しているジョブのジョブID

[ NODE: 0x0101012 ]
      RSC      TOTAL    FREE   ALLOC
      cpu         32      16      16
      mem       57Gi    30Gi    27Gi

 RUNNING_JOBS: 2551                                  ← ノード 0x01010012 で実行中のジョブのジョブID
```

```
CUSTOM RESOURCE(PER NODE)                          ← ノード 0x0101012 のカスタム資源の利用状況
RSCNAME                    TOTAL        FREE      ALLOC
customrscname4                 1           0          1

JOBS_USING_ROUTE: 1030,1031,1032,1033,1034         ← ノードを通信経路として使用しているジョブのジョブID
```

参考

FXサーバで実行されるジョブでは、ジョブプロセスのノード間通信がジョブに割り当てられていないノードを経由する場合があります。通信経路となっているノードがソフトウェアの異常などでダウンしても、通信経路のインターコネクトや、その制御をするICC(Interconnect Controller)が正常であればジョブの実行に影響はありません。しかし、ICCの故障などで通信が行えなくなった場合は、そのノードを通信経路としているすべてのジョブが影響を受け、中断させられます。

2.4.4 ジョブの終了の確認

ジョブが終了したことは以下の方法でわかります。

- pjstatコマンドによるジョブの状態の確認

  ジョブが終了すると(EXIT、REJECTまたはCANCEL状態)、pjstatコマンドの通常の表示対象外となります。-Hまたは--historyオプションを指定すると、終了したジョブだけ表示されます。

```
$ pjstat -H

JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE ...  (*)
1234567    jobname1   NM EXT usrname  01/01 00:00:00  0100:00:00          12:2x3x2     ...  (*1)
1234570    jobname1   NM RJT usrname  01/01 00:00:00  0100:00:00          12:2x3x2     ...  (*2)
1234590    jobname1   NM CCL usrname  01/01 00:00:00  0100:00:00          12:2x3x2     ...  (*3)

(*) 紙面の都合上、右側の表示は省略しています。
(*1) EXIT 状態のジョブ   (*2) REJECT 状態のジョブ   (*3) CANCEL 状態のジョブ
```

注意

- 終了したジョブの状態は、管理者が設定した期間だけ保持されます。このため、上記操作をしたときに状態の保持期間を過ぎているジョブについては状態が表示されません。
- バルクジョブとステップジョブの終了したサブジョブは、同一ジョブ内のすべてのサブジョブが終了すると-Hオプション指定時の表示に含まれます。それまでは-Hオプションを指定しないときの表示に含まれます。
  下記の例では、ステップジョブのサブジョブ4_0は終了していますが、ほかのサブジョブが実行中のため(ジョブ4がRUNNING状態)、-Hオプションを指定しない場合に表示されます。

```
$ pjstat -E

JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE ...  (*)
4          t1.sh      ST RUN user1    07/12 13:09:43  -                   -            ...
4_0        t1.sh      ST EXT user1    07/12 13:09:43  0001:00:00          32           ...
4_1        t2.sh      ST RUN user1    07/12 13:50:32  0001:00:00          32           ...
...
$ pjstat -E -H
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE ...  (*)
1          t1.sh       NM EXT user1   07/05 08:05:11  0001:00:00          32           ...
2          t2.sh       NM EXT user1   07/06 12:30:50  0001:00:00          32           ...
(*) 紙面の都合上、右側の表示は省略しています。
```

- メールによる通知

  ジョブ投入時にpjsubコマンドの-mオプションでメール通知の指定をしている場合、ジョブが終了するとユーザー宛にメールが送信されます。
  なお、ジョブが異常終了した場合は、必ずメールで通知されます。詳細は"2.6.1 ジョブ実行結果の参照"を参照してください。

- pjwait コマンドによるジョブ終了の待ち合わせ

  ユーザーは、特定のジョブが終了するのを pjwait コマンドで待ち合わせることができます。pjwait コマンドは、指定されたジョブが終了するまで復帰しません。
  以下は、3つのジョブを投入し、それらの終了を pjwait コマンドで待ち合わせる例です。

```
$ pjsub job1.sh
[INFO] PJM 0000 pjsub Job 5300 submitted.
$ pjsub job2.sh
[INFO] PJM 0000 pjsub Job 5301 submitted.
$ pjsub job3.sh
[INFO] PJM 0000 pjsub Job 5302 submitted.
$ pjwait 5300 5301 5302                    ← 指定したジョブすべてが終了するまで待つ
5300 0 0 -                                 ← 各ジョブのジョブID、ジョブ終了コード、
5301 0 1 -                                   ジョブスクリプトの終了ステータス、シグナル番号
5302 0 0 -
```

2.5 ジョブの操作

ここでは、ジョブに対する操作を説明します。

参照

以降で説明する操作は、ジョブの状態によってはできない場合があります。詳細は "付録D ジョブに対する操作について" を参照してください。

2.5.1 ジョブの削除

投入したジョブを取り消すことを「ジョブの削除」と呼びます。ジョブを削除するには、pjdel コマンドでジョブIDを指定します。実行中のジョブを削除すると、ジョブは中止されます。

```
$ pjdel jobid [ jobid...]
```

ジョブが正常に削除された場合、以下のメッセージが表示されます。

```
[INFO] PJM 0100 pjdel Accepted job jobid.
```

存在しないジョブを指定した場合、以下のメッセージが表示されます。

```
[ERR.] PJM 0112 pjdel Job 存在しないジョブID does not exist.
```

階層化ストレージを利用するシステムでは、--llio-flushオプションを指定すると、第1階層ストレージ上から第2階層ストレージへの書出しが完了するのを待ち合わせてからジョブが削除されます。ただし、待ち合わせは最長でジョブの経過時間制限値に達するまでです。経過時間制限値を範囲で指定している場合(pjsub -L elapse=*min*-*max*)は、経過時間制限値の最大値(*max*)に達するまで、または次のジョブの実行のために終了させられるまで、ファイルの書出し完了を待ち合わせます。

```
$ pjdel --llio-flush jobid [ jobid...]
```

注意

- 削除する権限がないジョブを指定した場合は、存在しないジョブを指定した場合と同じ動作になります。
- ACCEPT 状態のジョブを削除すると、REJECT 状態に遷移します。ACCEPT 状態以外で削除すると、CANCEL 状態に遷移します。
- プロローグスクリプト実行中(RUNNING-P状態)やエピローグスクリプト実行中(RUNNING-E状態)に処理を中断して、ジョブを削除する場合は、--enforce オプションを指定してください。また、エピローグスクリプト実行前のジョブを--enforce オプションを指定して削除した場合は、エピローグスクリプトは実行されません。
  なお、--enforce オプションを指定できないように管理者が設定している場合があります。pjaclコマンドの出力項目 execute pjdel(--enforce) を確認してください。

- pjdelコマンドはジョブ運用管理機能へジョブの削除依頼をすると復帰します。ファイルの書出し完了の待ち合わせを含む、ジョブの削除処理はpjdelコマンドの復帰とは非同期で処理されます。
- 操作ミスなどによって間違って投入した大量の実行待ちジョブ(QUEUED状態)を削除する場合、ジョブ統計情報ファイルやpjstatコマンドの-Hオプションで表示されるジョブの履歴情報が大量に出力される可能性があります。これを避けたい場合は、以下のオプションを指定してください。

  --no-stats

  削除するジョブがQUEUED状態の場合、そのジョブのジョブ統計情報ファイル(.statsファイル)の出力を抑止します。本オプションを指定した場合、ジョブ投入時にpjsubコマンドの--statsまたは--STATSオプションを指定していても、ジョブ統計情報ファイルは出力されません。

  --no-history

  削除するジョブがQUEUED状態の場合、pjstatコマンドの-Hオプションで出力されるジョブの履歴情報に、そのジョブの情報を出力することを抑止します。なお、--no-historyオプションを指定できないように管理者が設定している場合があります。pjaclコマンドの出力項目execute pjdel(--no-history)を確認してください。

  上記のオプションの指定に関係なくジョブ統計情報ファイルやジョブの履歴情報を出力しないように管理者が設定している場合があります。設定については管理者にお問い合わせください。

2.5.2 ジョブへのシグナル送信

ユーザーは実行中 (RUNNING 状態) のジョブに対し、pjsig コマンドでシグナルを送信できます。

```
$ pjsig -s シグナル ジョブID
```

シグナルは、シグナル名(SIGHUP など)またはシグナル番号(1から64まで)で指定します。
以下は、ジョブに対してシグナル SIGKILL (シグナル番号9)を送信する例です。

```
$ pjsig -s 9 1                                  ← シグナルをシグナル番号で指定
[INFO] PJM 0700 pjsig Accepted job 1.

$ pjsig -s SIGKILL 2                            ← シグナルをシグナル名で指定
[INFO] PJM 0700 pjsig Accepted job 2.
```

2.5.3 ジョブの固定と固定解除

実行中のジョブを pjdel コマンドで削除すると、ジョブは終了し、再度実行するためには、ジョブの投入をやり直さなければいけません。pjhold コマンドを使用すると、ジョブは中断し、実行結果は破棄されますが、投入された状態は保持されます。これを「ジョブの固定」と呼び、この状態を HOLD 状態と呼びます。

HOLD 状態を解除するには pjrls コマンドを使用します。これにより、ジョブは再度スケジューリングされ、最初から実行されます。
ジョブ実行中にシステムの停止があるような場合、一旦、ジョブを固定し、運用再開後に固定を解除することでジョブ投入の手間を省けます。

注意

プロローグスクリプト実行中(RUNNING-P状態)やエピローグスクリプト実行中(RUNNING-E状態)に処理を中断して、ジョブを固定する場合は、--enforce オプションを指定してください。また、エピローグスクリプト実行前のジョブを--enforce オプションを指定して固定した場合は、エピローグスクリプトは実行されません。
なお、--enforce オプションを指定できないように管理者が設定している場合があります。pjaclコマンドの出力項目 execute pjhold(--enforce)を確認してください。

以下は、ジョブの固定と固定解除の例です。

```
$ pjhold 1 2
[INFO] PJM 0300 pjhold Accepted job 1.
[INFO] PJM 0300 pjhold Accepted job 2.

$ pjstat -v 1-2
```

```
JOB_ID     JOB_NAME   MD ST  USER   ...     REASON
1          jobname1   NM HLD user1  ...     user1
2          jobname2   NM HLD user1  ...     user1

$ pjrls 1 2
[INFO] PJM 0400 pjrls Job 1 released.
[INFO] PJM 0400 pjrls Job 2 released.

$ pjstat -v 1-2
JOB_ID     JOB_NAME   MD ST  USER   ...  LST ...   REASON
1          jobname1   NM QUE user1  ...  HLD ...   -
2          jobname2   NM QUE user1  ...  HLD ...   -
```

階層化ストレージを利用するシステムでは、--llio-flushオプションを指定すると、第1階層ストレージ上から第2階層ストレージへの書出しが完了するのを待ち合わせてからジョブが固定されます。ただし、待ち合わせは最長でジョブの経過時間制限値に達するまでです。経過時間制限値を範囲で指定している場合(pjsub -L elapse=*min*-*max*)は、経過時間制限値の最大値(*max*)に達するまで、または次のジョブの実行のために終了させられるまで、ファイルの書出し完了を待ち合わせます。

```
$ pjhold --llio-flush jobid [ jobid..]
```

注意

pjholdコマンドはジョブ運用管理機能へジョブの固定依頼をすると復帰します。ファイルの書出し完了の待ち合わせを含む、ジョブの固定処理はpjholdコマンドの復帰とは非同期で処理されます。

2.5.4 ジョブのパラメーター変更

投入したジョブに対して、pjalter コマンドで変更できるパラメーターは以下です。

- -L オプション
  - ジョブの経過時間制限値 (elapse)
  - ジョブを実行するリソースユニット (rscunit)
  - ジョブを実行するリソースグループ (rscgrp)

```
$ pjalter [-L | --rsc-list] 資源名=値  jobid
```

- -p オプション
  - 同一ユーザーのジョブにおける優先度 (priority)

```
$ pjalter -p priority jobid
```

注意

- ジョブのパラメーターは、ジョブの状態が QUEUED、HOLD、または ERROR の場合に変更できます。
  FXサーバで実行するジョブについては、管理者がジョブ運用管理機能の設定で許可していれば、ジョブの状態がRUNNINGでも経過時間制限値を変更できます。ただし、エンドユーザの権限では、経過時間制限値の変更は短縮だけができます。すでに経過した時間よりも小さい値には変更できません。
- 管理者によって、ユーザーが pjalter コマンドを実行できないように設定されている場合があります。
- バルクジョブのサブジョブID指定は、経過時間制限値の変更時だけ可能です。
- ジョブを実行するリソースユニットやリソースグループを変更する場合、ジョブを投入したユーザーに対してジョブACL機能の設定が以下になっている必要があります(括弧内はpjaclコマンドの表示)。
  - 変更先のリソースユニットやリソースグループに対してジョブを投入できること('execute pjsub'が'enable')。

— ジョブ投入時に指定されているオプションは、変更先のリソースユニットやリソースグループにおいても指定が許可されていること('execute pjsub(*opt*)'が'enable')。

— ジョブの投入時に指定されている資源量は、変更先のリソースユニットやソースグループにおいても上限値と下限値('pjsub option parameters'の'upper'と'lower')の範囲に収まっていること。

— ジョブがカスタム資源を使用する場合、変更先のリソースユニットやリソースグループにも使用するカスタム資源が定義されていること('pjsub option parameters'に使用するカスタム資源が表示される)。

上記をpjaclコマンドで確認するときは、変更先のリソースユニットやリソースグループを指定してください。

```
$ pjacl --rscunit リソースユニット
$ pjacl --rscgrp リソースグループ
```

- 経過時間制限値が範囲で指定されているジョブ ("2.3.2.3 ジョブの経過時間制限値の指定"を参照) については、どのパラメーターも変更できません。
- パラメーターを変更すると、管理者が設定したタイミングでジョブは再度スケジューリングされ、変更したパラメーターが適用されます。ジョブにパラメーターが適用されるとpjstatコマンドで表示される情報が更新されます。

2.6 ジョブの結果の確認

ここではジョブの実行結果を知る方法について説明します。

2.6.1 ジョブ実行結果の参照

ジョブの標準出力、標準エラー出力は、ジョブ投入時のカレントディレクトリにそれぞれファイルとして作成されます。ただし、ジョブがバルクジョブまたはステップジョブの場合、標準出力、標準エラー出力はサブジョブごとにファイルへ出力されます。
デフォルトの出力先ファイル名は以下のとおりです。これらはジョブ投入時に変更できます("2.3.2.10 バッチジョブの標準出力、標準エラー出力ファイルの指定" 参照)。

ジョブの出力	出力先ファイル名
標準出力	*ジョブ名*.*ジョブID*.out ただし、バルクジョブ、ステップジョブではジョブID部分はサブジョブIDとなります。 例1) ジョブ名が job.sh、ジョブID が123456 の場合 job.sh.123456.out 例2) ジョブ名が bulkjob.sh、サブジョブID が 123456[0]、123456[1]、... のバルクジョブ場合 bulkjob.sh.123456[0].out、bulkjob.sh.123456[1].out、... 例3) ジョブ名が stepjob.sh、サブジョブID が 123456_0、123456_1、... のステップジョブの場合 stepjob.sh.123456_0.out、stepjob.sh.123456_1.out、...
標準エラー出力	*ジョブ名*.*ジョブID*.err ただし、バルクジョブ、ステップジョブではジョブID部分はサブジョブIDとなります。 例1) ジョブ名が job.sh、ジョブID が123456 の場合 job.sh.123456.err 例2) ジョブ名が bulkjob.sh、サブジョブID が 123456[0]、123456[1]、... のバルクジョブ場合 bulkjob.sh.123456[0].err、bulkjob.sh.123456[1].err、... 例3) ジョブ名が stepjob.sh、サブジョブID が 123456_0、123456_1、... のステップジョブの場合 stepjob.sh.123456_0.err、stepjob.sh.123456_1.err、...

注意

- ジョブ名の先頭が半角数字の場合、出力ファイル名の先頭に文字 "J" が付加されます。

- 出力ファイル名で、ジョブ名の部分(先頭に付加される文字"J"も含む)は、63 文字までです。
- ジョブスクリプトではなく、標準入力からジョブを投入した場合は、ジョブ名の部分は "STDIN" となります。
- バルクジョブやステップジョブに対し、標準出力や標準エラー出力を同じファイル名に指定した場合、各サブジョブの出力が混在します。
- FXサーバ上で実行するジョブ内でmpiexecコマンドを実行すると、その標準出力および標準エラー出力は、mpiexecコマンドのオプションで指定しなければジョブACL機能の項目mpiexec-*("2.2.2 制限情報の確認"参照)で定義されるファイルへ出力されます。mpiexecコマンドの標準出力および標準エラー出力については"2.3.6.9 mpiexecコマンドの標準出力/標準エラー出力 [FX]"やDevelopment Studioのマニュアル「MPI使用手引書」を参照してください。

ジョブの標準出力や標準エラー出力にジョブ運用ソフトウェアから以下のようなメッセージが出力される場合があります。

```
[プライオリティ] PJM nnnn メッセージ文          (標準エラー出力)
[プライオリティ] PLE nnnn メッセージ文          (標準出力または標準エラー出力)
[プライオリティ] PLE nnnn plexec メッセージ文   (標準出力または標準エラー出力)
```

書式	意味
[*プライオリティ*]	[ERR.]、[WARN]、または[INFO]で、メッセージの重要度を表します。
PJM	ジョブマネージャー機能のメッセージを表します。 管理者はこのメッセージを出力するかどうかを設定できるため、事象が発生しても必ずしもメッセージが出力されるとは限りません。
PLE	並列実行環境の機能のメッセージを表します。
plexec	並列実行環境の内部機能 plexec コマンドのメッセージを表します。
nnnn	メッセージIDで、4桁の整数です。

参照

これらのメッセージについては、マニュアル「ジョブ運用ソフトウェア コマンドリファレンス」の "第3章 エンドユーザ向けコマンドリファレンス" にある "ジョブ実行時に表示されるメッセージ" を参照してください。

ジョブに異常が発生した場合、pjsub コマンドを実行したユーザーに、異常の内容を示すメールが通知されます。ジョブの受付けが拒否された場合、この通知はありません。
メールの通知例を以下に示します。

```
Subject: PJM job: 6814. error.                ← ジョブID 6814 のジョブが異常終了したことを示す

Job name:       pjmtest.sh                    ← ジョブ名
Job owner:      user1                         ← ジョブ投入ユーザー名
Mail sent at:   Tue Jun 11 21:33:55 JST 2019  ← 本メール送信時刻


Reason: Node down.                            ← 異常終了の原因
```

参照

- メールで通知されるメッセージについては "付録B ジョブ実行に関する通知メッセージ" を参照してください。
- ジョブ実行環境を指定して投入したジョブが正常に終了しなかった場合は、"E.2トラブルシューティング"を参考にして対処してください。

注意

pjsub コマンドの --mail-list オプションでメール送信先を指定していない場合は、計算クラスタ管理ノードのユーザーアカウントに対してメールが送信されます。ユーザーへのメール配送についてはシステムの設計により異なりますので、管理者にお問い合わせください。

参考

ジョブの異常が発生した場合、その原因はジョブ統計情報に出力されます。

2.6.2 ジョブ統計情報の出力

ジョブ投入時に pjsub コマンドの -s または -S オプションを指定している場合は、ジョブ投入時のカレントディレクトリ配下のファイル (ファイル名 : *ジョブ名.ジョブID*.stats) または、--spath オプションで指定したファイルにジョブ統計情報が出力されます。詳細は "A.2 ジョブ統計情報の出力"やmanマニュアルpjstatsinfo(7)を参照してください。
これ以外に、pjstat コマンドの -H オプションと、-s または -S オプションを指定することで、終了したジョブのジョブ統計情報を参照できます。

```
$ pjstat -H                    ← 終了したジョブのジョブIDを確認
...
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM           NODE_REQUIRE ... (*)
1234567    jobname1   NM EXT usrname  01/01 00:00:00  0100:00:00           12:2x3x2     ...
...

$ pjstat -H  -s 1234567        ← ジョブID を指定し、ジョブ統計情報を表示 (-S オプションの場合は詳細表示)
...

(*) 紙面の都合上、右側の表示は省略しています。
```

表示される項目の意味については pjstat コマンドの man マニュアルを参照してください。

注意

- 管理者が設定するジョブ状態の保持期間を過ぎているジョブは参照できず、pjstat コマンドの表示には登場しません。
- ジョブを FXサーバで実行した場合と、PRIMERGYサーバで実行した場合では、出力される統計情報は一部異なります。
- 実行したジョブのジョブ統計情報には、管理者が設定するプロローグ・エピローグ処理の情報が加算されています。
  ただし、実行したジョブの実行経過時間にプロローグ・エピローグ処理が含まれるかどうかはシステムの設定によります。詳細は管理者にお問い合わせください。
- FXサーバで実行したジョブのジョブ統計情報には、ジョブに割り当てた資源以外に、MPI 非同期通信処理に関するアシスタントコアの使用情報も出力されます。

参考

ノードの故障などシステムの問題でジョブが中断し、自動的に再実行したジョブは ("2.3.2.7 ジョブの自動再実行についての指定"参照)、再実行時の結果がジョブの終了結果となります。

2.7 ノードがダウンした場合のジョブへの影響

ここでは、ジョブの実行中にノードがダウンした場合のジョブへの影響について説明します。

表2.37 計算ノードがダウンした場合のジョブへの影響

対象	自動再実行が有効なジョブ(*)	自動再実行が無効なジョブ(*) および会話型ジョブ
ジョブ	ジョブは中断し、QUEUED状態に遷移します。 計算機資源の割り当て後、再度実行されます。	ジョブは中断し、終了します。
ジョブ統計情報ファイル (*.statsファイル)	ジョブが中断した時点のジョブ統計情報は出力されず、再実行時の結果が出力されます。	ジョブが中断した時点のジョブ統計情報が出力されます。
ジョブの標準出力 (*.outファイル) 標準エラー出力 (*.errファイル)	ジョブが中断するまでの結果と再実行時の結果が出力されます。	ジョブが中断するまでの結果が出力されます。
ジョブが出力したファイル	ジョブが中断するまでに出力された結果になります。 再実行時に同名ファイルが存在する場合、その扱いはジョブのファイル操作の仕様に依存します。	ジョブが中断するまでに出力された結果になります。

(*)自動再実行の有効/無効は、ジョブ投入時に指定する--restartオプションや--norestartオプション、または管理者によるジョブ運用の設定に従います。

付録A　ジョブの情報

ここでは、pjstat コマンドや pjsub コマンドが出力するジョブの情報について説明します。

A.1　pjstat や pjstat -v の出力

pjstat コマンドは、ジョブの情報を出力します。

```
$ pjstat
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM            NODE_REQUIRE     VNODE  CORE V_MEM
XXXXXXXXXX XXXXXXXXXX XX XXX XXXXXXXX MM/DD hh:mm:ss  hhhh:mm:ss-hhhh:mm:ss nnnnnn:XXxYYxZZ  nnnnnn nnn  nnnnnnnnnnMiB
...
```

-vオプションを指定すると、表示される情報が追加されます。

```
$ pjstat -v
JOB_ID     JOB_NAME   MD ST  USER     GROUP   START_DATE      ELAPSE_TIM ELAPSE_LIM            NODE_REQUIRE (*)
VNODE  CORE V_MEM        V_POL E_POL RANK       LST EC  PC  SN PRI ACCEPT         RSC_GRP  REASON
XXXXXXXXXX XXXXXXXXXX XX XXX XXXXXXXX XXXXXXX MM/DD hh:mm:ss< hhhh:mm:ss hhhh:mm:ss-hhhh:mm:ss nnnnnn:XXxYYxZZ (*)
nnnnnn nnn nnnnnnnnnnMiB XXXXX XXXXX XXXXXXX    XXX XXX XXX XX XXX MM/DD hh:mm:ss XXXXXXXX XXXXXXXXXXXXXXXX

備考）紙面の都合で、上記表示例は(*)の箇所で改行しています。実際には1行として表示されます。
```

表示される情報とその意味を以下の表に示します。

参考

- 項目名欄に [-v] が付いているものは、-v オプション指定時に表示される項目です。
- バルクジョブとステップジョブには、ジョブIDに対応する情報とサブジョブIDに対応する情報があります。前者を特にバルクジョブまたはステップジョブのサマリ情報と呼びます。

表A.1 pjstat コマンドの出力項目

項目名	説明
JOB_ID	ジョブID (10桁の 10進数) サブジョブの場合は、サブジョブID (*ジョブID*[*バルク番号*]、または *ジョブID_ステップ番号*)
JOB_NAME	ジョブ名 (先頭から 10文字のみ)
MD	ジョブモデル NM:通常ジョブ ST:ステップジョブ BU:バルクジョブ MW:マスタ・ワーカ型ジョブ
ST	ジョブの現在の処理状態 ACC:ジョブの投入が受け入れられた状態 CCL:ジョブ実行中止による終了 ERR:エラーによる固定状態 EXT:ジョブ終了処理完了 HLD:ユーザーによる固定状態 QUE:ジョブ実行待ち状態 RJT:投入が受け付けられなかった状態 RNA:ジョブ実行に必要な資源を獲得中 RNE:エピローグ処理中 RNO:ジョブ終了処理完了待ち状態 RNP:プロローグ処理中 RSM:リジューム処理中

項目名	説明
	RUN:ジョブ実行中 SPD:サスペンド済み SPP:サスペンド処理中
USER	実行ユーザー名 (先頭の8文字だけ) ユーザーがOSに登録されていない場合は、ユーザーIDを出力します。
GROUP [-v]	実行ユーザーのグループ名 (先頭の8文字だけ) ステップジョブのサマリ情報の場合は、ステップジョブの最初のサブジョブ投入時に指定したグループ名を出力します。 グループがOSに登録されていない場合は、グループIDを出力します。 実行中のサブジョブがない場合、次に実行される予定のサブジョブの情報を出力します。
START_DATE	ジョブの実行開始予定時刻または実行開始時刻 ・ (*MM*/*DD hh*:*mm*) ジョブの実行開始予定時刻。実行開始予定時刻は括弧で囲まれます。 ・ (*YYYY*/*MM*/*DD*) ジョブの実行開始予定時刻(1年後以降の場合) ・ (*MM*/*DD hh*:*mm*)# 実行開始予定時刻がスケジューリング期間を超えた場合、時刻の後ろに"#"が付きます。 この場合、実行開始予定時刻は表示された時刻以降になります。 ・ *MM*/*DD hh*:*mm*:*ss* ジョブが開始した時刻 ・ (*MM*/*DD hh*:*mm*)< または *MM*/*DD hh*:*mm*:*ss*< バックフィルが適用されたジョブには、時刻の後ろに "<" が付きます。 ・ (*MM*/*DD hh*:*mm*)@ または *MM*/*DD hh*:*mm*:*ss*@ 実行開始時刻が指定されたジョブには、時刻の後ろに "@" が付きます。 ステップジョブのサマリ情報の場合は、実行中のサブジョブの情報を出力します。 実行中のサブジョブがない場合、次に実行される予定のサブジョブの情報を出力します。 実行開始予定時刻の表示は、管理者が以下のように設定している場合があります。 ・ 時刻の代わりに、管理者が決めた文字列を出力する。 ・ 実行開始予定時刻の後ろに付加される文字 "#"、"<"、および "@" のすべて、または一部を出力しない。
ELAPSE_TIM [-v]	実行経過時間 "*hhhh*:*mm*:*ss*" 桁が溢れる場合は、ss を省略して出力します。 ジョブの実行経過時間にはサスペンド期間は含まれません。 バルクジョブ、ステップジョブのサマリ情報の場合は、"-" を出力します。
ELAPSE_LIM	経過時間制限 表示書式には2種類あります。 ・ *時間* (*hhhh*:*mm*:*ss*) ジョブは最長で *hhhh*:*mm*:*ss* の時間だけ実行可能です。 ・ *時間1*-*時間2* (*hhhh*:*mm*:*ss*-*hhhh*:*mm*:*ss*) ジョブの実行経過時間が "*時間1*" を超えると、資源に空きがあれば最大 "*時間2*" に達するまでは実行できることを示します。 桁が溢れる場合は、*ss* を省略して出力します。 ステップジョブのサマリ情報の場合は、"-" を出力します。 バルクジョブのサマリ情報の場合、サブジョブごとに経過時間制限値が異なるときは、"-"を出力します。
NODE_REQUIRE	・ FXサーバ ジョブの投入時に指定したノード数とノード形状 "*nnnnnn*:*XX*x*YY*x*ZZ*" 上記のフォーマットに収まりきらない場合は、ノード形状のみ出力します。

項目名	説明
	・ PRIMERGYサーバ ジョブ投入時のノード数 "*nnnnnnn*" 指定がない場合は "-" を出力 ステップジョブのサマリ情報の場合は、"-" を出力
VNODE	仮想ノード数 "*nnnnnnn*" 以下のジョブの場合は "-" を出力します。 ・FXサーバ上のジョブ ・PRIMERGYサーバのノード割り当てジョブ 本項目を表示しないように管理者が設定している場合があります。
CORE	仮想ノード当たりのCPUコア数 "*nnn*" 以下のジョブの場合は "-" を出力します。 ・FXサーバ上のジョブ ・PRIMERGYサーバ上のノード割り当てジョブ 本項目を表示しないように管理者が設定している場合があります。
V_MEM	仮想ノード当たりのメモリ量 "*nnnnnnnnnn*MiB" ジョブ投入時に指定したパラメーター vnode-mem または mem の値です。 CPUコア当たりのメモリ量 core-mem を指定したジョブは、仮想ノード当たりのCPUコア数を乗じた値になります。 以下のジョブの場合は "-" を出力します。 ・FXサーバ上のジョブ ・PRIMERGYサーバ上のノード割り当てジョブ 本項目を表示しないように管理者が設定している場合があります。
V_POL [PG][-v]	仮想ノード配置ポリシー A_PCK : Absolutely PACK PACK : PACK A_UPK : Absolutely UNPACK UPCK : UNPACK 以下の場合は "-" を出力します。 ・FXサーバ上のジョブ ・PRIMERGYサーバ上のノード割り当てジョブ ・ステップジョブのサマリ情報 本項目を表示しないように管理者が設定している場合があります。
E_POL [PG][-v]	実行モードポリシー SHARE : SHARE SMPLX : SIMPLEX 以下の場合は "-" を出力します。 ・FXサーバ上のジョブ ・PRIMERGYサーバ上のノード割り当てジョブ ・ステップジョブのサマリ情報 本項目を表示しないように管理者が設定している場合があります。
RANK [PG][-v]	ランクマップの指定方法 bynode : rank-map-bynode bychip : rank-map-bychip 以下の場合は "-" を出力します。 ・FXサーバ上のジョブ ・PRIMERGYサーバ上のノード割り当てジョブ ・ステップジョブのサマリ情報 本項目を表示しないように管理者が設定している場合があります。
LST [-v]	ジョブの以前 (「ジョブの現在の処理状態」に遷移する前) の処理状態 値の意味は、項目STを参照してください。

<table>
<tr><th>項目名</th><th>説明</th></tr>
<tr><td>EC
[-v]</td><td>ジョブスクリプトの終了コード
バルクジョブ、ステップジョブのサマリ情報の場合は、"-" を出力します。</td></tr>
<tr><td>PC
[-v]</td><td>ジョブ終了コード (PJM コード)
ジョブ実行における、ジョブマネージャーの処理結果を示すコードです。
バルクジョブ、ステップジョブのサマリ情報の場合は、"-" を出力します。
コードの意味は以下のとおりです。
一部のジョブ終了コードについては、それに対応したメッセージが、ジョブの標準エラー出力に出力されます。
意味や対処の詳細については、「コマンドリファレンス」の"第3章 エンドユーザ向けコマンドリファレンス" にある "ジョブ実行時に表示されるメッセージ" を参照してください。(*)印のジョブ終了コードの対処については管理者にお問い合わせください。
<table>
<tr><th>ジョブ終了コード
(PJM コード)</th><th>意味</th></tr>
<tr><td>0</td><td>ジョブの正常終了</td></tr>
<tr><td>1</td><td>ユーザーが操作した pjdelコマンドによる CANCEL</td></tr>
<tr><td>2</td><td>ジョブの受付け拒否判定による REJECT
pjsub コマンドがエラーになります。</td></tr>
<tr><td>3 (*)</td><td>ジョブマネージャー出口機能による実行拒否
ジョブは実行されていません。</td></tr>
<tr><td>4</td><td>ユーザーが操作した pjhold コマンドによる HOLD</td></tr>
<tr><td>6</td><td>ステップジョブ依存関係式によるCANCEL
ジョブは実行されていません。</td></tr>
<tr><td>7</td><td>デッドライン強制指定によりCANCEL</td></tr>
<tr><td>8 (*)</td><td>ジョブマネージャー出口機能による CANCEL
ジョブは実行されていません。</td></tr>
<tr><td>9</td><td>再実行不可指定のため、ジョブ再構築時に EXIT</td></tr>
<tr><td>11</td><td>経過時間制限違反によるジョブ実行タイムアウト</td></tr>
<tr><td>12</td><td>メモリ使用量超過による強制終了</td></tr>
<tr><td>16</td><td>カレントディレクトリまたは標準入力/標準出力/標準エラー出力ファイルへのアクセス不可による終了</td></tr>
<tr><td>18</td><td>ジョブの実行可能時間の最小値を超えて実行していたジョブが、後続のジョブの実行、またはデッドラインスケジュールの開始により終了。
前者が原因の場合は、項目REASONが"ANOTHER JOB STARTED"になり、後者の場合は"DEADLINE SCHEDULE STARTED"になります。</td></tr>
<tr><td>20</td><td>ノードダウン</td></tr>
<tr><td>21</td><td>シェルの実行失敗</td></tr>
<tr><td>22</td><td>ICCエラー
注: Tofuインターコネクトのリンクダウン時に通信経路を変更しても("2.3.2.12 Tofuインターコネクトのリンクダウンに対する動作の指定 [FX]"参照)、ジョブの実行を継続できないと判断された場合も該当します。</td></tr>
<tr><td>23</td><td>OOM Killer動作による終了</td></tr>
<tr><td>25</td><td>HA失敗</td></tr>
<tr><td>26 (*)</td><td>プロローグ、エピローグ処理のエラー</td></tr>
<tr><td>27 (*)</td><td>資源管理出口処理のエラー</td></tr>
<tr><td>28</td><td>ジョブ実行環境の異常</td></tr>
<tr><td>29</td><td>指定したジョブ実行環境情報が不正</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>項目名</th><th colspan="2">説明</th></tr>
<tr><td></td><td>30</td><td>サスペンドまたはリジューム処理失敗による中断</td></tr>
<tr><td></td><td>100 (*)</td><td>ジョブマネージャーの内部エラー</td></tr>
<tr><td></td><td>120 (*)</td><td>ジョブスケジューラーの内部エラー</td></tr>
<tr><td></td><td>140 (*)</td><td>ジョブ資源管理の内部エラー</td></tr>
<tr><td></td><td>160 (*)</td><td>Tofuライブラリの内部エラー [FX]
ジョブは実行されていません。</td></tr>
<tr><td></td><td>180</td><td>階層化ストレージの内部エラー</td></tr>
<tr><td>SN
[-v]</td><td colspan="2">シグナル番号
バルクジョブ、ステップジョブのサマリ情報の場合は、"-" を出力します。</td></tr>
<tr><td>PRI
[-v]</td><td colspan="2">ジョブの優先度 (0から255)
数字が大きいほど優先度が高いことを示します。
ステップジョブのサマリ情報の場合は、"-" を出力します。</td></tr>
<tr><td>ACCEPT
[-v]</td><td colspan="2">ジョブの投入日時 "MM/DD hh:mm:ss"</td></tr>
<tr><td>RSC_GRP
[-v]</td><td colspan="2">ジョブ投入時のリソースグループ</td></tr>
<tr><td>REASON
[-v]</td><td colspan="2">エラーメッセージ
ジョブを実行する、しないにかかわらず、そのジョブの何らかの処理に対する結果コードに対応するメッセージです。
バルクジョブ、ステップジョブのサマリ情報の場合は、"-" を出力します。
出力されるメッセージの意味は以下のとおりです。
<table>
<tr><th>エラーメッセージ</th><th>意味</th></tr>
<tr><td>-</td><td>エラーなし。</td></tr>
<tr><td>MM/DD hh:mm DELAY</td><td>実行開始予定時刻が、指定した実行開始時刻以降になりました。</td></tr>
<tr><td>ANOTHER JOB STARTED</td><td>ジョブの実行可能時間の最小値を超えて実行していたジョブが、後続のジョブを実行するために終了させられました。</td></tr>
<tr><td>DEADLINE SCHEDULE STARTED</td><td>ジョブの実行可能時間の最小値を超えて実行していたジョブが、デッドラインスケジュールの開始により終了させられました。</td></tr>
<tr><td>ELAPSE LIMIT EXCEEDED</td><td>経過時間制限を超過しました。</td></tr>
<tr><td>FILE IO ERROR</td><td>ユーザーのジョブ投入時のカレントディレクトリにアクセス不能です。</td></tr>
<tr><td>GATE CHECK</td><td>ジョブマネージャー出口機能によってキャンセルされました。</td></tr>
<tr><td>IMPOSSIBLE SCHED</td><td>スケジューリングができませんでした。</td></tr>
<tr><td>INSUFF CPU</td><td>物理的にCPU数が不足しています。</td></tr>
<tr><td>INSUFF MEMORY</td><td>物理的にメモリ量が不足しています。</td></tr>
<tr><td>INSUFF NODE</td><td>物理的にノード数が不足しています。</td></tr>
<tr><td>INSUFF CustomResourceName</td><td>資源名 CustomResourceName で定義されているカスタム資源が不足しています。</td></tr>
<tr><td>INTERNAL ERROR</td><td>内部エラー</td></tr>
<tr><td>INVALID HOSTFILE</td><td>pjsub コマンドの rank-map-hostfile パラメーターで指定したホストファイルが不正です。[FX]</td></tr>
<tr><td>LIMIT OVER MEMORY</td><td>ジョブ実行中にメモリ量制限を超過しました。</td></tr>
<tr><td>LOST COMM</td><td>並列プロセスの全対全通信が保証されません。</td></tr>
<tr><td>NO CURRENT DIR</td><td>ユーザーのジョブ投入時のカレントディレクトリまたは標準入力/標準出力/標準エラー出力ファイルにアクセスできませんでした。</td></tr>
</table></td></tr>
</table>

項目名	説明	
	NOT EXIST *CustomResourceName*	資源名 *CustomResourceName* のカスタム資源は定義されていません。
	RESUME FAIL	リジュームに失敗しました。
	RSCGRP NOT EXIST	リソースグループが存在しません。
	RSCGRP STOP	リソースグループが停止しています。
	RSCUNIT NOT EXIST	リソースユニットが存在しません。
	RSCUNIT STOP	リソースユニットが停止しています。
	RUNLIMIT EXCEED	ジョブの同時実行制限数を超過しました。
	SUSPEND FAIL	サスペンドに失敗しました。
	USELIMIT EXCEED	同時使用ノード数制限または同時使用CPUコア数制限による実行待ちです。
	USER NOT EXIST	ジョブの実行ユーザーがシステムに存在しません。
	WAIT SCHED	スケジューリング対象ジョブ数の制限に達したため、スケジューリングの対象外になりました。
	そのほかの文字列	• pjdel、pjhold、または pmsuspend コマンドの --reason オプションで指定したメッセージ • ジョブマネージャー出口機能、ジョブスケジューラー出口機能、またはジョブ資源管理出口機能で管理者が設定したメッセージ pjhold および pmsuspend コマンドの --reason オプションによるメッセージの場合は、"*コマンド実行ユーザー名*: *message*" の形式で表示されます。--reason オプションの指定がない場合は、"*コマンド実行ユーザー名*:" が表示されます。

pjstat コマンドの --with-summary または --summary オプションを指定すると、状態別のジョブ数が表示されます。表示される情報とその意味を以下の表に示します。

```
$ pjstat --with-summary

  ACCEPT QUEUED RUNING RUNOUT   HOLD  ERROR SUSPND REJECT   EXIT CANCEL   TOTAL
  nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnnn
s nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnn nnnnnnn

JOB_ID      JOB_NAME    MD ST  USER      START_DATE       ELAPSE_LIM             NODE_REQUIRE      VNODE  CORE V_MEM
XXXXXXXXXX XXXXXXXXXX XX XXX XXXXXXXX MM/DD hh:mm:ss  hhhh:mm:ss-hhhh:mm:ss nnnnnn:XXxYYxZZ  nnnnnn nnn  nnnnnnnnnnMiB
```

表A.2 pjstat コマンドの出力項目 (状態別のジョブ数)

項目	説明
ACCEPT	ジョブ受付け待ち状態のジョブ数
QUEUED	ジョブ実行待ち状態のジョブ数
RUNING	実行中のジョブ数
RUNOUT	ジョブの終了待ち状態のジョブ数
HOLD	ユーザーによる固定状態のジョブ数
ERROR	エラーによる固定状態のジョブ数
SUSPND	サスペンド期間中(状態が SUSPEND、SUSPENDED、RESUME)のジョブ数 なお、管理者が本項目を表示しないように設定している場合があります。
REJECT	受付けが拒否された状態のジョブ数
EXIT	終了したジョブ数

項目	説明
CANCEL	中止されたジョブ数
TOTAL	表示されている状態別ジョブ数の合計

A.2 ジョブ統計情報の出力

pjstat コマンドや pjsub コマンドの -s、-S オプションを指定するとジョブ統計情報が出力されます。

注意

出力されるジョブ統計情報の項目とその出力順序は管理者が変更できます。このため、出力されるジョブ統計情報は以下の例と異なる場合があります。

- pjstat -s の出力例

```
$ pjstat -s
[Job Statistical Information]
 JOB ID                    : 100
 JOB NAME                  : job.sh
 JOB TYPE                  : BATCH
 JOB MODEL                 : NM
 ...
<中略>
 ...
 START BULKNO              : -
 END BULKNO                : -
```

- pjstat -S の出力例

```
$ pjstat -S
[Job Statistical Information]
 JOB ID                    : 100
 JOB NAME                  : job.sh
 JOB TYPE                  : BATCH
 JOB MODEL                 : NM
 ...
 START BULKNO              : -
 END BULKNO                : -

[Node Statistical Information]          ← ノードごとまたは仮想ノードごとの情報
 VNODE ID                  : -
 NODE ID                   : 0x00014DBF
 ...
 CPU BITMAP (USE)          : 0xF
 RANK NO                   : 0
```

- pjsub -s の出力例 (ジョブ統計情報ファイル)

```
Job Statistical Information

 JOB ID                    : 100
 JOB NAME                  : job.sh
 JOB TYPE                  : BATCH
 ...
 <中略>
 ...
 START BULKNO              : -
 END BULKNO                : -
```

- pjsub -S の出力例 (ジョブ統計情報ファイル)

```
Job Statistical Information

 JOB ID                    : 100
 JOB NAME                  : job.sh
 JOB TYPE                  : BATCH
 ...
 <中略>
 ...
 START BULKNO              : -
 END BULKNO                : -

--------------------------------------------------------------------------------
Node Statistical Information          ← ノードまたは仮想ノードごとの統計情報

 NODE ID                   : 0x00014DBE
 TOFU COORDINATE           : (23,17,16)
 ...
 CPU BITMAP (USE)          : 0xF
 RANK NO                   : 0

--------------------------------------------------------------------------------
Node Statistical Information          ← ノードまたは仮想ノードごとの統計情報

 NODE ID                   : 0x00014DBF
 ...
 CPU BITMAP (USE)          : 0xF
 RANK NO                   : 1
```

参照

pjstat コマンドや pjsub コマンドで出力されるジョブ統計情報の項目については、manマニュアルpjstatsinfo(7)を参照してください。

注意

- ジョブ統計情報の項目PERF COUNT *n*について
  ジョブ統計情報には、FXサーバのCPUの性能情報に関する項目PERF COUNT 1～PERF COUNT 9があります。以下の場合、これらの項目は値が0になります。

  ー ジョブ内でDevelopment Studioのプロファイラ機能または実行時情報出力機能を使用した場合

  ー ジョブ内で xospastop コマンドを実行した場合(xospastop コマンドについては"2.1.1.7 PAPI ライブラリを使用するジョブの作成 [FX]"参照)

  ジョブ統計情報には、FXサーバのCPUの性能情報に関する項目PERF COUNT 1～PERF COUNT 9があります。会話型ジョブの場合、これらの項目は値が"-"になります。

  Development Studio以外の MPI 処理系については、項目PERF COUNT *n*は取得できず、ジョブ統計情報に計上されません。

  項目PERF COUNT *n*から、ジョブの性能情報を算出できます。詳細は"A.3 ジョブの性能情報の算出について [FX]"を参照してください。
  ジョブが富士通プロファイラを利用した場合、PMUカウンタ(Performance Monitoring Unit Counter)の採取は停止され、停止するまでにジョブ資源管理機能が定期的に収集した最新の情報が本項目に設定されます。ジョブが富士通プロファイラを利用したかどうかは項目FJ PROFILERで確認できます。

- FXサーバで実行したジョブのジョブ統計情報には、ジョブに割り当てた CPU コアでの処理だけでなく、アシスタントコアで処理したMPI非同期通信処理も計上されます。
  このため、FXサーバのジョブでは、項目 CPU NUM(ALLOC)、CPU NUM(USE)は項目 CPU NUM(REQUIRE) よりも大きくなる場合があります。また、項目 USER CPU TIME (USE)、SYSTEM CPU TIME (USE)、および CPU TIME(TOTAL) にはアシスタントコアで実行された CPU 時間が含まれます。

- FXサーバのジョブで計上するCPU時間の最小単位は10ms(10000000ns)です。動作したCPU時間がこの最小時間より短い場合、ジョブ統計情報には0に丸めて計上します。

- FXサーバのノード平均消費電力は、「ノード平均消費電力 (推定)」と「ノード平均消費電力 (実測)」の2種類を出力します。
「ノード平均消費電力 (推定)」はCPUの命令発行数などに基づいてハードウェアにより算出されます。計算ノードとI/Oノードに依存して構成が異なるアシスタントコア、PCI Expressデバイスなどの消費電力は計上されません。
「ノード平均消費電力 (実測)」は電力計測素子からハードウェアにより収集されます。計算ノードとI/Oノードに依存して構成が異なるアシスタントコア、PCI Expressデバイスなどの消費電力も計上されます。ノード個体差や処理するデータパターンの違いによる消費電力の差も含まれます。
同一ジョブを異なるノードで実行する場合、「ノード平均消費電力 (推定)」はジョブに割り当てられるノードによらず同一であるのに対し、「ノード平均消費電力 (実測)」はジョブに割り当てられるノードによって差があります。2種類のノード平均消費電力を比較すると±40%程度の差がある場合があります。
ノード最大消費電力、ノード最小消費電力、ノード消費電力量についても、それぞれ(推定)、(実測)があり上記と同様です。

[McKernelモードでのジョブ統計情報について]

McKernelモードのジョブ統計情報については、以下に注意してください。

- McKernelモードのジョブを実行する環境は、ホストOSとMcKernelの両方になります("1.9 ジョブ実行環境"の"McKernel モード"参照)。したがって、McKernelモードのジョブ統計情報は基本的にはホストOSとMcKernel両方の情報になりますが、一部は以下に示す環境の情報になります。

表A.3 McKernelモードのジョブ統計情報の対象となる環境

ジョブ統計情報	対象となる環境
USER CPU TIME (USE)	McKernel
SYSTEM CPU TIME (USE)	McKernel
CPU TIME (TOTAL)	McKernel
MAX MEMORY SIZE (USE)	McKernel
ASSISTANT CORE (USE)	ホストOS
ASSISTANT CORE USER CPU TIME (USE)	ホストOS
ASSISTANT CORE SYSTEM CPU TIME (USE)	ホストOS
ASSISTANT CORE MAX MEMORY SIZE (USE)	ホストOS
FJ PROFILER	McKernel
SECTOR CACHE	McKernel
INTRA NODE BARRIER	McKernel
UTILIZATION INFO OF POWER API	McKernel
CPU BITMAP (USE)	ホストOS (*1)
PERF COUNT 1 ～ PERF COUNT 9	McKernel (*2)

(*1)
項目CPU BITMAP (USE)は、ホストOS上でのCPU番号体系に基づいたCPU IDのビットマップ情報が表示されます。

(*2)
項目PERF COUNT 1～PERF COUNT 9は、McKernel上で使用したCPU統計情報の値が表示されます。ホストOS上で起動したプロセスが使用したCPU統計情報は含まれません。

- McKernelモードで実行されたジョブに対するジョブ統計情報のCPU時間は、ユーザーがジョブプロセス内でgetrusage()システムコールを呼んで計測したCPU時間より通常は大きくなります。これは、McKernelからジョブプロセスを起動するために呼び出す補助的なプログラム(mcexec)のCPU時間も計上されるためです。

- McKernelの起動パラメーターを指定しているときは("2.3.8 ジョブの実行環境の指定"参照)、以下のジョブ統計情報は正しくない場合があります。

  ー CPU BITMAP (USE)

- USER CPU TIME (USE)
- SYSTEM CPU TIME(USE)
- CPU TIME (TOTAL)
- MEMORY SIZE(ALLOC)
- MAX MEMORY SIZE(USE)
- CPU NUM (USE)

[KVMモードでのジョブ統計情報について]

KVMモードでは、ジョブ運用ソフトウェアは仮想マシン内で使用された一部の資源情報を取得できません。このため、これらに関するジョブ統計情報の出力は以下になります。

表A.4 KVMモードのジョブ統計情報について

項目	説明
USER CPU TIME (USE)	仮想マシンに接続するsshプロセスに関する情報を表します。
SYSTEM CPU TIME (USE)	仮想マシンに接続するsshプロセスに関する情報を表します。
CPU TIME (TOTAL)	仮想マシンに接続するsshプロセスに関する情報を表します。
MAX MEMORY SIZE (USE)	QEMUプロセスと、仮想マシンに接続するsshプロセスの最大メモリ使用量を表します。
CPU NUM (USE)	仮想マシンに接続するsshプロセスのCPU使用数を表します。
ASSISTANT CORE (USE)	FXサーバでは0になります。 PRIMERGYサーバではKVMモードかどうかに関わらず、常に"-"になります。
ASSISTANT CORE USER CPU TIME (USE)	FXサーバでは0になります。 PRIMERGYサーバではKVMモードかどうかに関わらず、常に"-"になります。
ASSISTANT CORE SYSTEM CPU TIME (USE)	FXサーバでは0になります。 PRIMERGYサーバではKVMモードかどうかに関わらず、常に"-"になります。
PERF COUNT 1 ～ PERF COUNT 9	"-"になります。

A.3 ジョブの性能情報の算出について [FX]

ジョブ統計情報から、FXサーバで実行したジョブに関する性能情報を以下のようにして算出できます。

表A.5 ジョブの性能情報の算出方法

性能情報	算出方法
実行サイクル数	SUM(PERF COUNT 1)
浮動小数点命令演算数	SUM(PERF COUNT 2)+SUM(PERF COUNT 3)×4
メモリ読出し要求数	SUM(PERF COUNT 4)/12
メモリ書出し要求数	SUM(PERF COUNT 5)/12
スリープサイクル数	SUM(PERF COUNT 6)

PERF COUNT *N*: ジョブ統計情報の項目
SUM(*perf*): ノードごとのジョブ統計情報に出力される項目*perf*の合計値

付録B　ジョブ実行に関する通知メッセージ

ジョブ実行が正常に行われなかった場合や、ジョブの開始・終了を通知するようにユーザーが指示した場合は、ユーザーにはメールでその詳細が通知されます。
ここでは、その通知内容に含まれるメッセージと意味について説明します。メール通知に関しては "2.6.1 ジョブ実行結果の参照" を参照してください。

表B.1 メールで通知されるメッセージ

メッセージ	意味	ジョブの状態	対処
Reason: Internal error: *code*.	内部エラーを検出しました。*code* は内部コードが出力されます。	ERRORに遷移	管理者に連絡してください。管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社 Support Desk に連絡してください。
Reason: Node down.	ジョブ実行依頼時または、実行中に計算ノードがノードダウンしました。	QUEUEDに遷移または終了	管理者に連絡してください。管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社 Support Desk に連絡してください。 中断したジョブは自動的に再実行されるため、ユーザーが対処する必要はありません。ただし、自動再実行が無効になっているジョブは、再投入が必要です。
Reason: Unable to analyze pjm *code*.	ジョブ資源管理機能、ジョブスケジューラー機能、または階層化ストレージ機能から不正なコードを受け取りました。	ERRORに遷移	管理者に連絡してください。管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社 Support Desk に連絡してください。
Reason: Fail to write the jobinfo file. *詳細* path:*pathname*	統計情報の出力に失敗しました。*詳細*には、出力に失敗した原因が表示されます。*pathname* には、書き込めなかったパス名が表示されます。	終了	詳細に internal error と表示されている場合は、管理者に連絡してください。管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社 Support Desk に連絡してください。 上記以外は、*pathname* にファイルが書き込めるかどうかを確認してください。 また、当該ジョブの統計情報を、システムに記録されている情報から得ることができないか、ジョブIDを示して、管理者に確認してください。 管理者は、pmdumpjobinfo コマンドでジョブの統計情報が取得できる場合、ユーザーへの情報提供を検討してください。
Reason: Fail to write the jobinfo record. *詳細* path:*pathname*			
Fail to get user name or group name.	ジョブを実行するユーザー、グループの取得に失敗しました。	ERRORに遷移	ジョブの投入ユーザーやグループの存在を確認してください。
Host file is not correct.	pjsub コマンドの rank-map-hostfile パラメーターで指定したホストファイルが正しくありません。	ERRORに遷移	pjstat コマンドの -vオプション を実行して、REASONを確認してください。 REASONにINVALID HOSTFILE が表示されている場合、ホストファイルを見直してください。 上記以外の場合には、管理者に連絡してください。 管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合

メッセージ	意味	ジョブの状態	対処
			わせて、担当保守員 (SE)、または当社 Support Desk に連絡してください。
Fail to access current directory.	ジョブ投入時のカレントディレクトリにアクセスできませんでした。	ERRORに遷移	ジョブのカレントディレクトリの存在を確認してください。
The current directory does not exist or have permission. path: *pathname*.	ジョブ投入時のカレントディレクトリまたは標準入力/標準出力/標準エラー出力ファイルにアクセスできません。*pathname* にはジョブ投入時のカレントディレクトリのパス名が表示されます。	終了	ジョブ投入時のカレントディレクトリの存在や、標準入力/標準出力/標準エラー出力ファイルのパス名を確認してください。 または、ジョブ投入ユーザーにそれらへのアクセス権があるかを確認してください。
Job *jobid* was canceled due to the suspend processing failure of the system.	サスペンド処理が失敗してジョブが中断されました。	QUEUEDに遷移または終了	中断したジョブは自動的に再実行されるため、ユーザーが対処する必要はありません。ただし、自動再実行が無効になっているジョブは、再投入が必要です。
Job *jobid* canceled by pjdel command from *username*.	ユーザー *username* が実行した pjdel コマンドにより、削除させられました。	削除	対処不要です。
Job *jobid* canceled, because of norestart job.	ジョブは自動再実行が無効に設定されているため、再実行されません。	削除	ジョブを実行するには、再度投入してください。
The job was canceled because of the deadline period.	ジョブはデッドラインスケジュールにより中断させられました。	QUEUEDに遷移	空き資源がある場合は、ジョブは再実行されます。空き資源がない場合は、デッドラインスケジュールの終了後、ジョブが再実行されます。
CPU time limit exceeded.	CPU 使用時間が制限値を越えました。	終了	ジョブに対する資源制限値を確認してください。
Elapse time limit exceeded.	実行可能時間が制限値を越えました。	終了	ジョブに対する資源制限値を確認してください。
Memory size limit exceeded.	メモリ使用量が制限値を越えました。	終了	ジョブに対する資源制限値を確認してください。
Killed by OOM killer.	OSのOOM Killerによって強制終了させられました。	終了	OOM Killerはジョブの動作に起因して発生する可能性があるため、以下2つの観点でジョブの内容を確認してください。 これらに該当しない場合は、管理者に連絡してください。管理者は「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社Support Deskに連絡してください。 1. メモリ割り当てを特定NUMAノードに限定している場合 システムコール set_mempolicy でMPOL_BINDを指定し、プロセスのメモリ獲得対象を特定NUMAノードに限定している場合、対象NUMAノードのメモリを使い切り、メモリが枯渇すると、OOM Killerにより強制終了させられます。 この場合、メモリ獲得対象NUMAノードの限定を止めるか、または、ジョブのメモリ獲得量を削減してください。

メッセージ	意味	ジョブの状態	対処
			2. 計算ノードのジョブ用のメモリ資源量(*)近くまで使用した場合 ジョブの動作に依存して、その時のジョブ用メモリの断片化の状況や、ジョブのメモリ獲得時にジョブ用のメモリ資源量がジョブの投入時に指定したメモリ使用量より少なくなっているような状況では、"Memory size limit exceeded."(PJM CODE 12)ではなく"Killed by OOM killer."(PJM CODE 23)として通知される場合があります。 この場合、ジョブのメモリ獲得量を削減してください。 (*) pjshowrscコマンドで表示される計算ノードのメモリ資源量
Job *jobid* was deleted by system factor.	システム要因によってジョブは削除されました。	削除	管理者に連絡してください。管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社Support Deskに連絡してください。
Abnormal end.	ジョブ実行に異常が発生しました。	終了	管理者に連絡してください。管理者は、「管理者向けガイド保守編」に従って、調査資料を採取し、出力されたメッセージと合わせて、担当保守員(SE)、または当社Support Deskに連絡してください。
Job *jobid* is started.	ジョブが開始しました。	-	対処不要です。 pjsubコマンドの-m bオプション指定時のみ出力される通知メッセージです。
Job *jobid* is completed.	ジョブが正常に終了しました。	-	対処不要です。 pjsubコマンドの-m eオプション指定時のみ出力される通知メッセージです。
Job *jobid* restarted.	ジョブが再実行されました。	-	対処不要です。 pjsubコマンドの-m rオプション指定時のみ出力される通知メッセージです。
Another job started.	ジョブの経過時間制限値の最小値を超えて実行していたジョブが、後続のジョブの実行より中断されました。	-	対処不要です。 pjsubコマンドの-m eオプション指定時のみ出力される通知メッセージです。
Deadline schedule started.	ジョブの経過時間制限値の最小値を超えて実行していたジョブが、デッドラインスケジュールの開始により中断されました。	-	対処不要です。 pjsubコマンドの-m eオプション指定時のみ出力される通知メッセージです。
Reason: Gate check. または Reason: *管理者が設定したメッセージ*	管理者が設定するジョブマネージャー出口機能、ジョブスケジューラー出口機能、またはジョブ資源管理出口機能によってジョブが中断されました。	削除	中断された理由は管理者に確認してください。
Reason: *rscname*=*value* is less than the lower limit (*limit-value*).	ジョブ投入時に指定した*rscname*=*value*が下限値*limit-value*より小さい値です。	終了	pjaclコマンドで資源 *rscname* の下限値を確認し、それを下回らない値を指定してください。

メッセージ	意味	ジョブの状態	対処
Reason: *rscname*=*value* is greater than the upper limit (*limit-value*).	ジョブ投入時に指定した *rscname*=*value*が上限値 (*limit-value*)より大きい値です。	終了	pjacl コマンドで資源 *rscname* の上限値を確認し、それを超えない値を指定してください。

付録C　Development Studio以外の MPI 処理系の実行について

C.1　MPI 処理系ごとの注意事項と実行例

Intel MPI、Open MPI、MPICH、Platform MPIなど、Development Studio以外のMPI処理系のMPIプログラムをジョブ運用ソフトウェア上で実行する際の注意事項と実行例について説明します。

[注意事項]

- ノードまたは仮想ノード上でMPIプロセスを実行するために、rsh、sshコマンドの代わりにジョブ運用ソフトウェアが提供するpjrshコマンドを使用してください。
  ただし、Intel MPI (Hydra)などでmpiexecプロセスの標準入力を利用した場合、pjrshコマンドがMPIプロセスへ標準入力を与えますが、標準入力のデータ量がおよそ8MBを超える場合、rsh、sshコマンドと比べて、転送性能が劣化する可能性があります。このようなときは、mpiexecプロセスの標準入力を利用するのではなく、各MPIプロセスからファイルで読み込んでください。
- ジョブに割り当てるノード資源を、以下のように指定してください。

  ー ノード単位で割り当てる場合

  ```
  pjsub -L "node=N" ...
  ```

  ー 仮想ノード単位で割り当てる場合
  仮想ノード配置ポリシーはAbsolutely UNPACK (abs-unpack)を指定してください。

  ```
  pjsub -L "vnode=N,vnode-core=M" -P "vn-policy=abs-unpack" ...
  または
  pjsub -L "vnode=N(core=M)" -P "vn-policy=abs-unpack" ...
  ```

- MPIプログラムがCPUコア内のすべての論理CPUを使用する場合は、pjaclコマンドでジョブACL機能の設定項目define assign-logical-cpuの値が"all"になっていることを確認してください。
- ホスト名を指定してMPIプログラムを実行する場合は、以下のように実施してください。

  a. FXサーバで実行する場合
  pjshowipコマンドを使用してください。
  pjshowipコマンドは、ジョブスクリプト内から実行することによって、ジョブに割り当てたノードリスト(ランク番号ごとのIPアドレス)をランク番号ごとに標準出力へ出力します。

  b. PRIMERGYサーバで実行する場合
  環境変数PJM_O_NODEINFを使用してください。
  環境変数PJM_O_NODEINFはジョブ運用ソフトウェアによって設定され、ジョブに割り当てたノードリスト(IPアドレスのリスト)を格納したファイルのパスを示します。

  具体的な実行方法は、後述の実行例を参考にしてください。

- MPIジョブを削除する場合は、そのMPI処理系がクリーンアップ処理(各種資源の解放など)をするシグナルを送信してから、ジョブを削除してください。クリーンアップ処理をするシグナルを送信せずにジョブを削除した場合、後続のジョブが実行できないなどの影響をおよぼす可能性があります。

  クリーンアップ処理をするシグナルは各MPI処理系の仕様を確認してください。一般的には、シグナルSIGINTまたはSIGTERMによりクリーンアップ処理が行われます。

[実行例]

以下は、MPI処理系ごとの実行例です。なお、記載されているバージョンは例であり、特に限定するものではありません。
MPIプログラムの一般的な実行方法やバージョンによる違いについては、各MPI処理系の仕様を確認してください。

- Intel MPI 2021 [PG]

  以下の一連の操作をするジョブスクリプトを投入することで、Intel MPIのMPIプログラムをジョブの仮想ノード上で実行できます。

  [Scalable Process Management System (Hydra)を利用する場合]

1. 環境変数の設定

   Hydra の環境変数を以下のように設定します。

   a. I_MPI_HYDRA_BOOTSTRAP=rsh

   b. I_MPI_HYDRA_BOOTSTRAP_EXEC=/bin/pjrsh

   c. I_MPI_HYDRA_HOST_FILE="${PJM_O_NODEINF}"

   これにより、rsh、ssh コマンドの代わりに pjrsh コマンドを使用するようになります。

2. MPIプログラム実行

   Intel MPIで作成されたMPIプログラムを実行します。プログラム実行にはmpiexec.hydraコマンドを利用します。

[例] プロセス並列のMPIプログラム(a.out)を実行するジョブスクリプト (Hydra を利用する場合)

```
...
export I_MPI_PERHOST=1
export I_MPI_HYDRA_BOOTSTRAP=rsh                    1.a.
export I_MPI_HYDRA_BOOTSTRAP_EXEC=/bin/pjrsh        1.b.
export I_MPI_HYDRA_HOST_FILE="${PJM_O_NODEINF}"     1.c.
mpiexec.hydra -n 4 a.out                            2.
...
```

注意

Intel MPIのMPIプログラムは、Multipurpose Daemon (MPD)ではジョブの仮想ノード上で実行できません。

参考

ジョブ運用ソフトウェアは、Intel MPIに対してDevelopment StudioのMPIと同様な実行ビューを提供する、mpiexec.hydraコマンドのラッパーコマンドmpiexec.tcs_intelを用意しています。
ラッパーコマンドmpiexec.tcs_intelを利用したジョブスクリプトの例については、"C.4 ラッパーコマンドmpiexec.tcs_intelを使用したMPIプログラム実行 [PG]" を参照してください。

- MPICH 3.4.2 [PG]

以下の一連の操作をするジョブスクリプトを投入することで、MPICHのMPIプログラムをジョブの仮想ノード上で実行できます。

[Scalable Process Management System (Hydra) を利用する場合]

1. 環境変数の設定

   Hydra の環境変数を以下のように設定します。

   a. HYDRA_BOOTSTRAP=rsh

   b. HYDRA_BOOTSTRAP_EXEC=/bin/pjrsh

   c. HYDRA_HOST_FILE="${PJM_O_NODEINF}"

   これにより、rsh、ssh コマンドの代わりにpjrshコマンドを使用するようになります。

2. MPI プログラム実行

   MPICHで作成されたMPIプログラムを実行します。プログラム実行にはmpiexec.hydraコマンドを利用します。

[例] プロセス並列のMPIプログラム(a.out)を実行するジョブスクリプト (Hydra を利用する場合)

```
...
export HYDRA_BOOTSTRAP=rsh                          1.a.
export HYDRA_BOOTSTRAP_EXEC=/bin/pjrsh              1.b.
export HYDRA_HOST_FILE="${PJM_O_NODEINF}"           1.c.
```

```
mpiexec.hydra -n 4 a.out                                    2.
...
```

- OpenMPI 4.1.5

  以下のようにして、OpenMPIのMPIプログラムをノードまたは仮想ノード上で実行できます。

注意

OpenMPIのダウンロードサイトから入手したopenmpi-4.0.0のソースファイルをビルドし、インストールするとき、ログインノードとすべての計算ノードで同じパスで参照できる共有ファイルシステム上にインストールしてください。

1. OpenMPIインストール先のetc/openmpi-mca-params.confファイルに以下を追加します。

```
plm_rsh_agent=/bin/pjrsh
```

2. MPIプログラム実行

   a. FXサーバで実行する場合
   mpiexecコマンドの-machinefileオプションで、pjshowipコマンドの出力から作成したノードリストファイルを指定します。
   [例] プロセス並列のMPIプログラム(a.out)を実行するジョブスクリプト

```
xospastop
pjshowip > ./iplist_"${PJM_JOBID}"
mpiexec -n 4 -machinefile ./iplist_"${PJM_JOBID}" --map-by node:SPAN a.out
rm -f ./iplist_"${PJM_JOBID}"
```

   b. PRIMERGYサーバで実行する場合
   mpiexecコマンドの-machinefileオプションで、ノードリストファイル${PJM_O_NODEINF}を指定します。
   [例] プロセス並列のMPIプログラム(a.out)を実行するジョブスクリプト

```
mpiexec -n 4 -machinefile "${PJM_O_NODEINF}" --map-by node:SPAN a.out
```

参考

警告メッセージについて

以下の警告メッセージが出力された場合、mpiexecコマンドのオプションとして--mca btl_openib_allow_ib 1を指定すると警告出力を抑止できることもあります。詳細はOpenMPIプロジェクトが公開している情報を参照してください。

```
You can override this policy by setting the btl_openib_allow_ib MCA parameter to true.
```

- Platform MPI 9.1.4 [PG]

  以下の一連の操作をするジョブスクリプトを投入することで、Platform MPIのMPIをジョブの仮想ノード上で実行できます。

1. Platform MPIの環境変数MPI_REMSHに/bin/pjrshを設定します。これにより、rsh、sshコマンドの代わりにpjrshコマンドを使用するようになります。

2. MPI プログラム実行

   Platform MPIで作成されたMPIプログラムを実行します。ここで、-hostfileオプションでノードリストファイル${PJM_O_NODEINF}を指定します。

[例] プロセス並列のMPIプログラム(a.out)を実行するジョブスクリプト

```
...
export MPI_REMSH=/bin/pjrsh                                 1.
mpirun -np 4 -hostfile "${PJM_O_NODEINF}" a.out             2.
...
```

C.2 プロセスへの CPU 資源のバインド [PG]

PRIMERGYサーバで、ジョブ運用ソフトウェアを使用してDevelopment Studio以外のMPI処理系のMPIプログラムを実行する場合は、pjpbindコマンドを利用することで、Development Studioと同様のCPU資源(CPU コアまたは論理CPU)のバインドが実現できます。

pjpbindコマンドは自身のプロセスにCPU資源をバインドしたあと、引数で指定されたプログラムを実行します。このとき、ジョブに割り当てられたノード内のCPU資源のうち、まだプロセスがバインドされていないCPU資源を優先的に選択し、プロセスにバインドします。
pjpbindコマンドの詳細は、manマニュアルを参照してください。

[例] ハイブリッド並列のMPIプログラム(a.out)を実行するジョブスクリプト

```
#PJM -L "vnode=2"
#PJM -L "vnode-core=8"
export OMP_NUM_THREADS=8
mpiexec -n 2 pjpbind a.out
```

この例のようにマルチスレッドのMPIプログラムをpjpbindコマンドで実行する場合は、スレッドへのCPU資源のバインドができます。
その際、スレッド数は環境変数PARALLELまたはOMP_NUM_THREADSで指定してください。これらの環境変数が同時に指定された場合は、環境変数PARALLELが優先されます。GNU gccの環境変数GOMP_CPU_AFFINITYを解釈可能なランタイムライブラリを使用するプログラムでは、pjpbindコマンドによって、1ノード内の複数のプロセス間で重複しないようにCPU資源をスレッドへバインドできます。

なお、pjpbindコマンドに--disable-threadオプションを指定した場合、実行されるプログラムで、プロセスにCPU資源を自動的にバインドしますが、スレッドへは CPU 資源を自動的にはバインドしません。

pjpbindコマンドを利用するときに以下の環境変数を設定することで、CPU資源のバインドのパターンを変更できます。

表C.1 CPU コアのバインドに関する環境変数

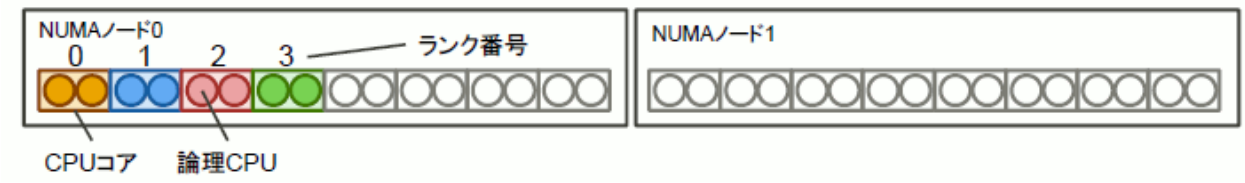
環境変数名	説明
PLE_MPI_PIN_DOMAIN	"omp" または整数 *n* を指定します。 ・ "omp" を指定した場合 環境変数 PARALLEL または OMP_NUM_THREADS で指定された値に等しい個数の CPU 資源をプロセスにバインドします。この 2つの環境変数が同時に指定された場合は、環境変数 PARALLEL が優先されます。どちらの環境変数も省略された場合は、整数値 1 を指定したときと同じ動作になります。 ・ 整数 *n* を指定した場合 *n* 個の CPU 資源をプロセスにバインドします。 CPU コアと論理 CPU のどちらをバインドするかは、環境変数 PLE_MPI_PIN_CELL の設定に従います。"omp" または整数値 *n* 以外の値を指定するか、この環境変数が未設定の場合、"omp" を指定したときと同じ動作になります。
PLE_MPI_PIN_CELL	"core" または "unit" を指定します。 ・ "core" を指定した場合 CPU コアごとにプロセスへ CPU 資源をバインドします。 ・ "unit" を指定した場合 論理 CPU ごとにプロセスへ CPU 資源をバインドします。 "core" または "unit" 以外の値を指定するか、この環境変数が未設定の場合は、以下の動作になります。 ・ "unit" を指定したときと同じ動作になる ― PLE_MPI_PIN_DOMAIN 環境変数が設定されている。 ― PLE_MPI_PIN_DOMAIN 環境変数が設定されていない、かつ、pjpbind コマンドが起動されたノードで、1つの CPU コアに複数のジョブプロセスがバインドされる。 ・ "core" を指定したときと同じ動作になる ― PLE_MPI_PIN_DOMAIN 環境変数が設定されていない、かつ、pjpbind コマンドが起動されたノードで、1つの CPU コアに 1つのジョブプロセスだけがバインドされる。 ― インテル®ハイパースレッディング・テクノロジーが有効ではない環境。

環境変数名	説明
PLE_MPI_PIN_ORDER	"range"、"compact"、または "scatter" を指定します。 ・ "range" を指定した場合 1つ前のプロセスにバインドされた CPU 資源から見て、CPU ID が昇順となるように、プロセスに CPU 資源をバインドしていきます。 ・ "compact" を指定した場合 1つ前のプロセスにバインドされた CPU 資源から距離が近くなるように CPU 資源をバインドします。 ・ "scatter" を指定した場合 1つ前のプロセスにバインドされた CPU 資源から距離が遠くなるように CPU 資源をバインドします。 どの場合も、CPU ID が最も小さい CPU 資源が、ジョブの中で最初の起点となり、上記順序でサイクリックに CPU 資源をバインドしていきます。 CPU コアと論理 CPU のどちらをバインドするかは、環境変数 PLE_MPI_PIN_CELL の設定に従います。"range"、"compact" または "scatter" 以外の値を指定するか、この環境変数が未設定の場合、"scatter" を指定したときと同じ動作になります。

これらの環境変数を使用した CPU 資源のバインドのイメージを以下に示します。

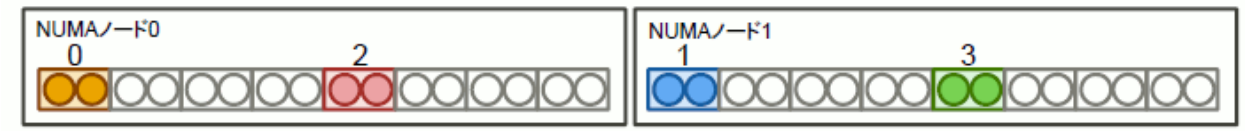
[例1]

```
...
export PLE_MPI_PIN_CELL=core
export PLE_MPI_PIN_ORDER=compact
mpiexec -n 4 pjpbind a.out
```



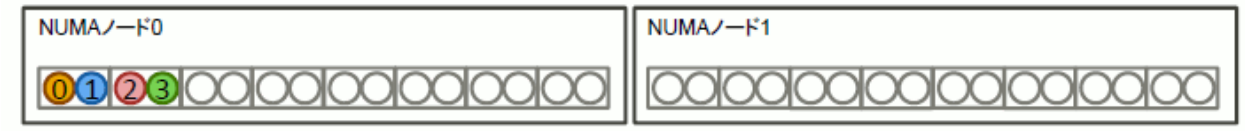
[例2]

```
...
export PLE_MPI_PIN_CELL=core
export PLE_MPI_PIN_ORDER=scatter
mpiexec -n 4 pjpbind a.out
```



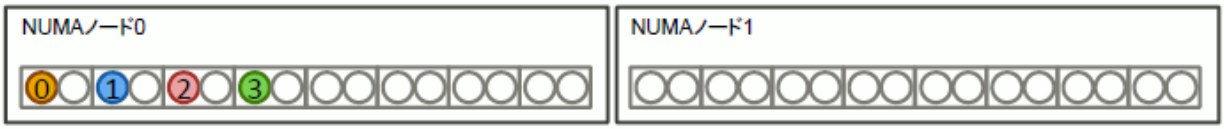
[例3]

```
...
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=compact
mpiexec -n 4 pjpbind a.out
```



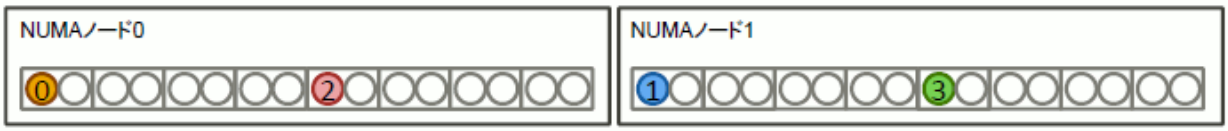
[例4]

```
...
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=range
mpiexec -n 4 pjpbind a.out
```



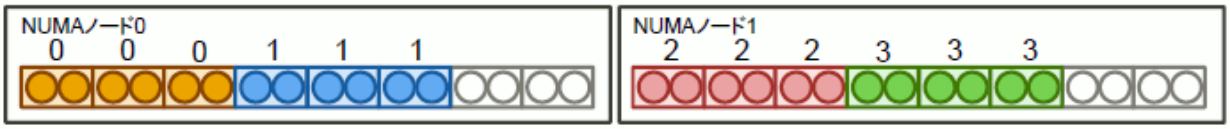
[例5]

```
...
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=scatter
mpiexec -n 4 pjpbind a.out
```



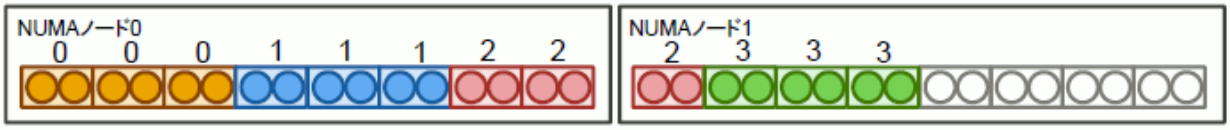
[例6]

```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=omp
export PLE_MPI_PIN_CELL=core
export PLE_MPI_PIN_ORDER=compact
mpiexec -n 4 pjpbind a.out
```



[例7]

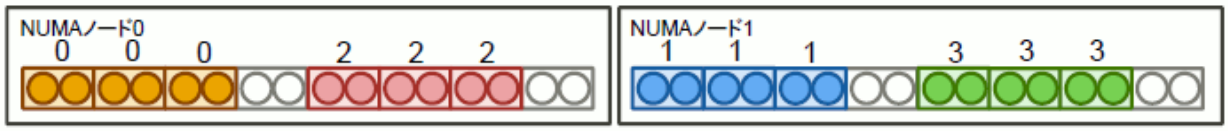
```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=omp
export PLE_MPI_PIN_CELL=core
export PLE_MPI_PIN_ORDER=range
mpiexec -n 4 pjpbind a.out
```



[例8]

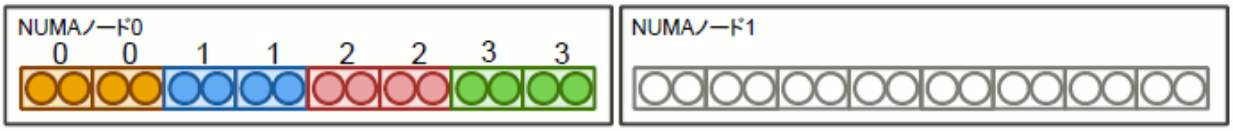
```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=omp
export PLE_MPI_PIN_CELL=core
```

```
export PLE_MPI_PIN_ORDER=scatter
mpiexec -n 4 pjpbind a.out
```



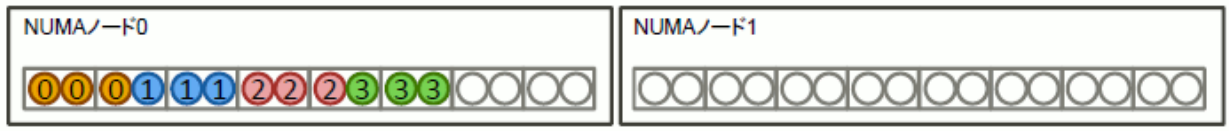
[例9]

```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=2
export PLE_MPI_PIN_CELL=core
export PLE_MPI_PIN_ORDER=compact
mpiexec -n 4 pjpbind a.out
```



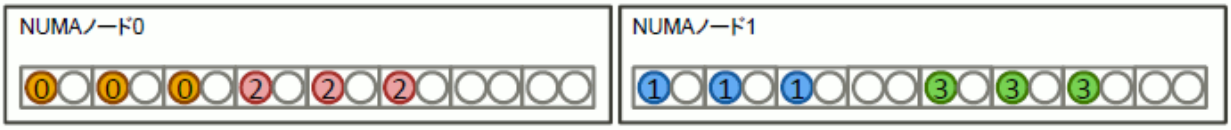
[例10]

```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=omp
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=compact
mpiexec -n 4 pjpbind a.out
```



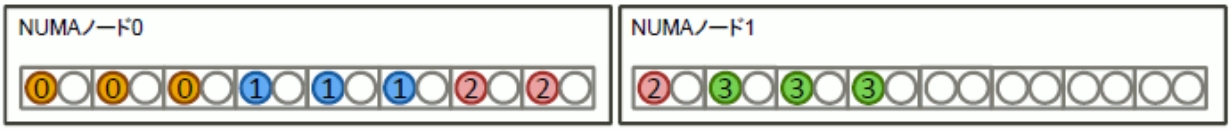
[例11]

```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=omp
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=scatter
mpiexec -n 4 pjpbind a.out
```



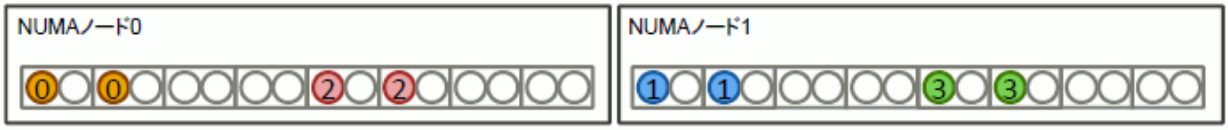
[例12]

```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=omp
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=range
mpiexec -n 4 pjpbind a.out
```



[例13]

```
...
export OMP_NUM_THREADS=3
export PLE_MPI_PIN_DOMAIN=2
export PLE_MPI_PIN_CELL=unit
export PLE_MPI_PIN_ORDER=scatter
mpiexec -n 4 pjpbind a.out
```



C.3 NUMA メモリの割り当てポリシーの設定 [PG]

Development Studio以外の MPI 処理系の MPI プログラムを実行する場合は、環境変数 PLE_MEMORY_ALLOCATION_POLICY を使用することで、NUMA メモリの割り当てポリシーが設定できます。

注意

- 環境変数 PLE_MEMORY_ALLOCATION_POLICY は、pjpbind コマンドを実行する場合だけ有効になります。
- NUMAメモリの割り当てポリシーを設定するためには、計算ノードにnumactlパッケージがインストールされている必要があります。numactlパッケージのインストールについては管理者へ相談してください。

NUMA アーキテクチャーの計算ノードでは、NUMA メモリのアクセス速度に起因したジョブの実行性能のゆらぎや劣化が発生する可能性があります。この影響を低減するためのメモリの割り当て方が、NUMA メモリの割り当てポリシーです。

環境変数 PLE_MEMORY_ALLOCATION_POLICY に設定できる値は、Development StudioのMPIにおけるMCAパラメーターplm_ple_memory_allocation_policyと同じです。MCA パラメーター plm_ple_memory_allocation_policy に設定できる値と NUMA メモリの割り当てポリシーの詳細については、Development Studioのマニュアル「MPI使用手引書」の "NUMAメモリ割り当てポリシーの設定値" を参照してください。

C.4 ラッパーコマンドmpiexec.tcs_intelを使用したMPIプログラム実行 [PG]

Intel MPI の mpiexec.hydra コマンドのラッパーコマンド mpiexec.tcs_intel を利用する際のジョブスクリプトの実行例を以下に示します。

注意

ラッパーコマンド mpiexec.tcs_intel を利用する場合は、ジョブ内の PATH 環境変数に、Intel MPI の mpiexec.hydra コマンドのインストールディレクトリが含まれている必要があります。

[例1] プロセス並列の MPI プログラム (a.out) を実行するジョブスクリプト

```
#!/bin/sh
#PJM -L "node=4"
mpiexec.tcs_intel a.out
```

[例2] フラット並列の MPI プログラム (a.out) を実行するジョブスクリプト

```
#!/bin/sh
#PJM -L "node=4"
#PJM --mpi "proc=64"
mpiexec.tcs_intel a.out
```

[例3] ハイブリッド並列の MPI プログラム (a.out) を実行するジョブスクリプト

```
#!/bin/sh
#PJM -L "node=4"
#PJM --mpi "proc=16"
export OMP_NUM_THREADS=4
mpiexec.tcs_intel a.out
```

ラッパーコマンド mpiexec.tcs_intel を利用してMPIプログラムを実行した場合、"C.2 プロセスへの CPU 資源のバインド [PG]" で説明した pjpbind コマンドを内部で呼び出し、プロセスへ CPU 資源をバインドします。

pjpbind コマンドのオプションを環境変数 PLE_I_MPI_PJPBIND_OPT で指定できます。
ラッパーコマンド mpiexec.tcs_intel 内で pjpbind コマンドによる CPU 資源のバインドを自動で行わないようにするには、環境変数 PLE_I_MPI_PJPBIND に "disable" を指定してください。

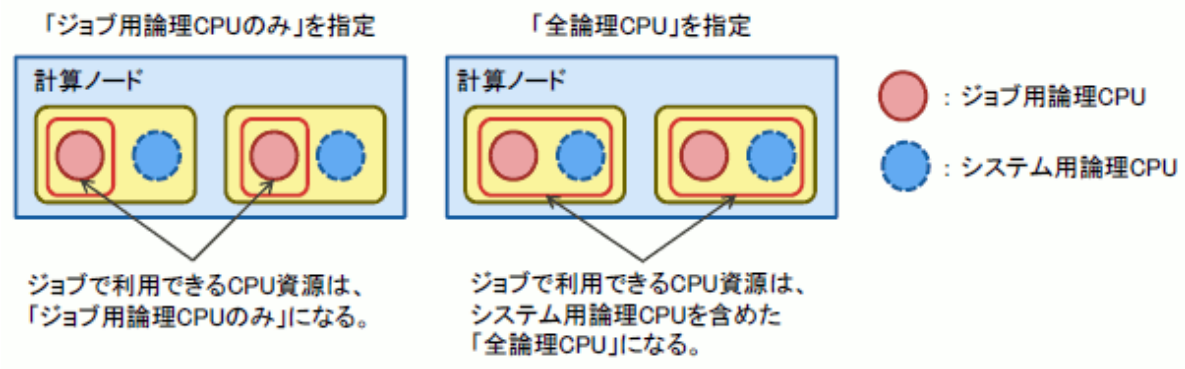
表C.2 ラッパーコマンド mpiexec.tcs_intel の環境変数

環境変数名	説明
PLE_I_MPI_PJPBIND_OPT	pjpbind コマンドのオプションを文字列で指定します。 例: export PLE_I_MPI_PJPBIND_OPT="--disable-thread" この環境変数は、環境変数 PLE_I_MPI_PJPBIND が設定されていない、または環境変数 PLE_I_MPI_PJPBIND に "enable" を指定した場合のみ、有効になります。
PLE_I_MPI_PJPBIND	"enable" または "disable" を指定します。 ・ enable pjpbind コマンドによる CPU 資源のバインドが行われます。 ・ disable pjpbind コマンドによる CPU 資源のバインドは行われません。 この環境変数を省略した場合、"enable" を指定したときと同じ動作になります。

C.5 ジョブで利用できる CPU 資源の範囲の設定 [PG]

Development Studio以外のMPI処理系のMPI プログラムで、インテル®ハイパースレッディング・テクノロジーが有効な環境の場合、ジョブ ACL 機能によってジョブで利用できる計算ノードの CPU 資源の範囲が設定されます。設定される CPU 資源の範囲は「ジョブ用論理 CPU のみ」または「全論理 CPU」です。

図C.1 ジョブが利用できる CPU 資源の範囲



ジョブで利用できる計算ノードの CPU 資源の範囲は、ジョブ ACL 機能で設定されますが、ジョブスクリプトでも設定できます。ジョブスクリプトで CPU 資源の範囲を設定する場合は、"C.2 プロセスへの CPU 資源のバインド [PG]" で説明した pjpbind コマンドを利用します。ジョブスクリプトには、設定する CPU 資源の範囲に従って、pjpbind コマンドが参照する以下の環境変数名とその設定値を指定してください。

表C.3 CPU 資源の範囲設定時に pjpbind コマンドが参照する環境変数名と設定値

設定する CPU 資源の範囲	環境変数名および設定値
ジョブ用論理 CPU のみ	PLE_ASSIGN_LOGICAL_CPU=job
全論理 CPU	PLE_ASSIGN_LOGICAL_CPU=all

なお、環境変数 PLE_ASSIGN_LOGICAL_CPU が設定されていない、または、PLE_ASSIGN_LOGICAL_CPU に "job" または "all" 以外の値が指定された場合、ジョブで利用できる CPU 資源の範囲は、ジョブ ACL 機能の設定に従います。

以下は、ジョブで利用できる CPU 資源の範囲を設定したジョブスクリプトの例です。

[例1] MPI プログラム (a.out) を「ジョブ用論理 CPU のみ」で実行するジョブスクリプト

```
#!/bin/sh
#PJM -L "node=4"
#PJM --mpi "proc=16"
...
export PLE_ASSIGN_LOGICAL_CPU=job
mpiexec.hydra pjpbind a.out
```

[例2] MPI プログラム (a.out) を「ジョブ用論理 CPU のみ」、MPI プログラム (b.out) を「全論理 CPU」で実行するジョブスクリプト

```
#!/bin/sh
#PJM -L "node=4"
#PJM --mpi "proc=16"
...
export PLE_ASSIGN_LOGICAL_CPU=job
mpiexec.hydra pjpbind a.out

export PLE_ASSIGN_LOGICAL_CPU=all
mpiexec.hydra pjpbind b.out
```

参考

論理 CPU が 3つ以上存在する場合、全論理 CPU のうち 1つをシステム用論理 CPU とし、残りすべてをジョブ用論理 CPU とします。

付録D　ジョブに対する操作について

ここでは、ジョブに対する操作が、ジョブの状態やジョブの種類によって可能かどうかを示します。

表内の記号の意味は以下のとおりです。

○: ジョブに対する操作は可能
×: ジョブに対する操作は不可
－: ジョブがその状態になることはない

バルクジョブやステップジョブに関して、2段になっているセルは以下の意味になります。

上段: バルクジョブやステップジョブに対する操作（操作対象としてジョブIDを指定する）
下段: バルクジョブやステップジョブのサブジョブに対する操作（操作対象としてサブジョブIDを指定する）

2段になっていないセルは、バルクジョブやステップジョブとそのサブジョブのどちらにも共通であることを示します。

表D.1 ジョブの削除 (pjdel)

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
ACCEPT	○	○	○	○
REJECT	×	×	×	×
QUEUED	○	○	○	○
RUNNING-A	○	－ ○	－ ○	○
RUNNING-P	×(*)	－ ×(*)	×(*)	×(*)
RUNNING	○	○	○	○
RUNNING-E	×(*)	－ ×(*)	×(*)	×(*)
RUNOUT	×	○ ×	－ ×	○
CANCEL	×	×	×	×
ERROR	○	○	○	－
EXIT	×	×	×	×
HOLD	○	○	○	－
SUSPEND	○	○	○	－
SUSPENDED	○	○	○	－
RESUME	○	○	○	－

(*) --enforceオプションを指定すれば、操作が可能です。ただし、--enforceオプションを指定する権限が必要です。

表D.2 ジョブの固定 (pjhold)

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
ACCEPT	×	×	×	×
REJECT	×	×	×	×
QUEUED	○	○	○	×
RUNNING-A	○(*1)	－	－	×

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
		○(*1)	○(*1)	
RUNNING-P	×(*2)	―	×(*2)	×
		×(*2)		
RUNNING	○(*1)	○(*3)	○(*1)	×
		○(*1)		
RUNNING-E	×(*2)	―	×(*2)	×
		×(*2)		
RUNOUT	×	×	―	×
			×	
CANCEL	×	×	×	×
ERROR	×	×	×	―
EXIT	×	×	×	×
HOLD	×	×	×	―
SUSPEND	○	○	○	―
SUSPENDED	○	○	○	―
RESUME	○	○	○	―

(*1) ジョブやサブジョブの投入時にpjsubコマンドの--norestartオプションを指定している場合は、操作できません。
(*2) --enforceオプションを指定すれば、操作が可能です。ただし、--enforceオプションを指定する権限が必要です。
(*3) バルクジョブ内の固定可能な状態のサブジョブが対象になります。固定できない状態のサブジョブは無視します。

表D.3 ジョブの固定解除 (pjrls)

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
ACCEPT	×	×	×	×
REJECT	×	×	×	×
QUEUED	×	×	×	×
RUNNING-A	×	―	―	×
		×	×	
RUNNING-P	×	―	×	×
		×		
RUNNING	×	○(*)	×	×
		×		
RUNNING-E	×	―	×	×
		×		
RUNOUT	×	○(*)	×	×
		×		
CANCEL	×	×	×	×
ERROR	×	×	×	―
EXIT	×	×	×	×
HOLD	○	○	○	―

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
SUSPEND	×	×	×	−
SUSPENDED	×	×	×	−
RESUME	×	×	×	−

(*)バルクジョブ内の固定解除可能な状態のサブジョブが対象になります。固定解除できない状態のサブジョブは無視します。

表D.4 ジョブの待ち合わせ (pjwait)

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
ACCEPT	○	○	○	○(*)
REJECT	○	○	○	○(*)
QUEUED	○	○	○	○(*)
RUNNING-A	○	−	−	○(*)
		○	○	
RUNNING-P	○	−	○	○(*)
		○		
RUNNING	○	○	○	○(*)
RUNNING-E	○	−	○	○(*)
		○		
RUNOUT	○	○	−	○(*)
			○	
CANCEL	○	○	○	○(*)
ERROR	○	○	○	−
EXIT	○	○	○	○(*)
HOLD	○	○	○	−
SUSPEND	○	○	○	−
SUSPENDED	○	○	○	−
RESUME	○	○	○	−

(*) pjwaitコマンドは、会話型ジョブの終了は待ち合わせず、また情報を何も表示せずに正常終了します。

表D.5 ジョブに対するシグナル送信 (pjsig)

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
ACCEPT	×	×	×	×
REJECT	×	×	×	×
QUEUED	×	×	×	×
RUNNING-A	×	−	−	×
		×	×	
RUNNING-P	×	−	×	×
		×		
RUNNING	○	○	○	○

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
RUNNING-E	×	― ×	×	×
RUNOUT	×	×	― ×	×
CANCEL	×	×	×	×
ERROR	×	×	×	―
EXIT	×	×	×	×
HOLD	×	×	×	―
SUSPEND	×	×	×	―
SUSPENDED	×	×	×	―
RESUME	×	×	×	―

表D.6 ジョブのパラメーター変更 (pjalter)

ジョブの状態	通常ジョブ マスタ・ワーカ型ジョブ	バルクジョブ	ステップジョブ	会話型ジョブ
ACCEPT	×	×	×	×
REJECT	×	×	×	×
QUEUED	○	○ ×	○(*1)	×
RUNNING-A	×	― ×	― ×	×
RUNNING-P	×	― ×	×	×
RUNNING	○ (*2)	○ (*2)	○ (*2)	×
RUNNING-E	×	― ×	×	×
RUNOUT	×	×	― ×	×
CANCEL	×	×	×	×
ERROR	○	○ ×	○(*1)	―
EXIT	×	×	×	×
HOLD	○	○ ×	○(*1)	―
SUSPEND	×	×	×	―
SUSPENDED	×	×	×	―
RESUME	×	×	×	―

(*1) サブジョブに対する操作の場合、リソースユニット名の変更はできません。
(*2) FXサーバ上で実行中のジョブに対する経過時間制限値の変更だけができます(管理者がジョブ運用管理機能の設定で許可している場合)。ただし、エンドユーザの権限では、経過時間制限値の変更は短縮だけができます。

付録E　ジョブ実行環境の利用について

E.1　ジョブ実行環境のイメージファイルの作成

ジョブ実行環境を自分で用意するには、イメージファイルの作成が必要です。

以下は、各ジョブ実行環境の手順です。

E.1.1　Docker モードの場合

"2.3.8 ジョブの実行環境の指定"に示した注意事項を満たすイメージであれば作成方法は任意です。

構築したコンテナイメージをジョブ実行環境として使用したい場合は、利用可能な実行方法(SDI指定、UDI指定)を管理者に確認してください。SDI指定の場合は、システムへのコンテナイメージの登録を管理者に依頼してください。

参考

FXサーバのコンテナ内でジョブ運用ソフトウェアの機能を動作させる場合には、下記に示す、FXサーバ固有の機能を利用するためのパッケージと設定が必要です。

[パッケージ一覧]

コンテナには、下記に示すジョブ運用ソフトウェアの同梱パッケージ("表E.1 ジョブ運用ソフトウェアの機能を動作させるのに必要なパッケージ一覧"参照)と、それらのパッケージが必要とするOSのパッケージ("表E.2 ジョブ運用ソフトウェアが動作するのに必要なOSのパッケージ一覧"参照)をコンテナイメージにインストールしてください。ジョブ運用ソフトウェアの機能を動作させる場合、ベースとするOSは、動作実績のある、計算ノードと同じ版数のOSを使用することを推奨します。ジョブ運用ソフトウェアのパッケージの入手については管理者にお問い合わせください。

なお、ここでは、コンテナイメージの大きさが大きくなりすぎないように、OSはRed Hat Enterprise Linux 8のCoreグループパッケージだけのインストールを基本とし、最低限必要な追加パッケージだけを記しています。コンテナを利用する上で不足する機能がある場合には、必要に応じてパッケージを追加してください。

表E.1 ジョブ運用ソフトウェアの機能を動作させるのに必要なパッケージ一覧

FJSVpxkrm
FJSVpxkrm-uti
FJSVpxpsm
FJSVpxpwrm_api
FJSVpxtof
FJSVxosfhehpc
FJSVxoshpcpwr
FJSVxoshwb
FJSVxoslibmpg
FJSVxoslibmpg-module
FJSVxosmemutils
FJSVxossec
docker-ce
libpfm
libpfm-devel
papi
papi-devel
papi-libs

xpmem
xpmem-kmod

表E.2 ジョブ運用ソフトウェアが動作するのに必要なOSのパッケージ一覧

coreutils
elfutils-devel
elfutils-libelf-devel
gcc
gcc-c++
libstdc++-devel
libatomic
libevent-2
openssl-devel
zlib-devel

[必要な設定]

コンテナの起動時にジョブ運用ソフトウェアの動作に必要なホストOS環境のディレクトリ(マウントポイント)を、管理者が起動設定ファイルにあらかじめ設定する必要があります。この設定については、管理者に依頼してください。

E.1.2 Mckernel モードの場合

公開サイト(https://github.com/ihkmckernel)からソースコードを入手し、ソースツリー内に配置されたREADMEに従ってMcKernelイメージファイル(mckernel.img)を構築してください。

構築したイメージをジョブ実行環境として使用したい場合は、利用可能な実行方法(SDI指定、UDI指定)を管理者に確認してください。SDI指定の場合は、システムへのイメージの登録を管理者に依頼してください。

E.1.3 KVM モードの場合

仮想マシンイメージは、利用者が作業用ワークステーションで、virt-installユーティリティ、virshコマンドラインインターフェースを使用して作成します。

作成したイメージをジョブ実行環境として使用したい場合は、利用可能な実行方法(SDI指定、UDI指定)を管理者に確認してください。SDI指定の場合は、システムへのイメージの登録を管理者に依頼してください。

[仮想マシンの要件]

virt-installユーティリティやvirshコマンドラインインターフェースを使って作成する仮想マシンイメージファイルについて記します。

仮想マシンイメージファイルの作成には、virt-installユーティリティを使用します。virt-installで指定するドメイン名(--domain)や、仮想マシンイメージのパス(--disk)は任意です。仮想マシンイメージファイルの作成と同時に作られるドメインXMLファイル(*<ドメイン名>*.xml)は、virshコマンドを使った仮想マシンの起動に必要なものですが、ジョブ運用ソフトウェアでは使用しません。
仮想マシンイメージファイルのフォーマットはQEMUがサポートする形式と同じです。
仮想マシンは、virtiofsをサポートするLinuxカーネルがインストールされていることが必要です(Red Hat Enterprise Linux 8.2以降を推奨)。

参照

仮想マシンの詳細は、Red Hat社のドキュメント「Red Hat Enterprise Linux 8 System Design Guide」の"PART IX. VIRTUALIZATION ON ARM 64 SYSTEMS"を参照してください。このドキュメントの入手方法については、管理者にお問い合わせください。

そのほか、ジョブ運用ソフトウェアでKVMモードを使う場合には、下記の要件も満たすようにしてください。

- SELinuxが無効に設定されていること。

- sshdパッケージがインストールされていること。
- ネットワーク設定はDHCPによるIPアドレスの付与ができるように設定されていること。
- デバイス名に紐づかないコネクションが存在すること。

  本要件を満たすために、仮想マシン内で下記を行ってください。

  ー デバイス名に紐づかないコネクションの作成(コネクション名を"kvm"とした場合の例)

  ```
  # nmcli connection add type ethernet ifname '*' con-name kvm
  Connection 'kvm' (cd871c4d-01ce-4335-8dbb-53d064d9d9ce) successfully added.
  ```

  ー コネクション定義の確認

  コネクションの作成が行われるとifcfg-*<コネクション名>*(本例では'ifcfg-kvm')というファイルが/etc/sysconfig/network-scripts/に作成されます。

  確認すべきポイントは下記です。

  - 'BOOTPROTO=dhcp'であること。
  - 'ONBOOT=yes'であること。
  - 'DEVICE='の行が存在しない(デバイス名に紐づかない)こと。

  ```
  # cat /etc/sysconfig/network-scripts/ifcfg-kvm
  TYPE=Ethernet
  PROXY_METHOD=none
  BROWSER_ONLY=no
  BOOTPROTO=dhcp
  DEFROUTE=yes
  IPV4_FAILURE_FATAL=no
  IPV6INIT=yes
  IPV6_AUTOCONF=yes
  IPV6_DEFROUTE=yes
  IPV6_FAILURE_FATAL=no
  IPV6_ADDR_GEN_MODE=stable-privacy
  NAME=kvm
  UUID=cd871c4d-01ce-4335-8dbb-53d064d9d9ce
  ONBOOT=yes
  ```

  ー (参考)コネクションの削除方法

  ```
  # nmcli connection del kvm
  Connection 'kvm' (cd871c4d-01ce-4335-8dbb-53d064d9d9ce) successfully deleted.
  ```

- ジョブ運用ソフトウェアの資源管理エージェントパッケージがインストールされていること。

注意

ジョブ運用ソフトウェアの資源管理エージェントパッケージは、systemdのinitシステムのサービスで動作するように作成されています。資源管理エージェントをsystemd以外のinitシステムで独自に動作させる場合には、資源管理エージェントのソースアーカイブを使って、利用する仮想マシンの環境に合わせてビルドし、資源管理エージェントを配置してください。ビルド手順はソースアーカイブに同梱されているREADMEをご覧ください。

資源管理エージェントパッケージ(FJSVpxkrm-libvirt-agent)および資源管理エージェントソースアーカイブ(FJSVpxkrm-libvirt-agent-*<version>*.src.tar.gz)はTechnical Computing SuiteのDVDに同梱されています。これらの入手については管理者にお問い合わせください。

参考

virshコマンドラインインタフェースを使って、仮想マシンの起動・終了ができます。事前に仮想マシンイメージの動作を確認する場合、仮想マシン内で資源管理エージェント(pxkrm-libvirt-agent.service)も正常に起動することを確認してください。

以下に動作確認のオペレーション例を記します。

```
# virsh define 〈ドメインXMLファイル(ドメイン名.xml)へのパス〉(例：/etc/libvirt/qemu/test-domain.xml)
# virsh start --console 〈ドメイン名〉(例：test-domain)
(ここから仮想マシン内)
login:
password:
...
# systemctl status pxkrm-libvirt-agent.service    (systemctlコマンドによる確認)
● pxkrm-libvirt-agent.service - Technical Computing Suite - Kernel Resource Manager: libvirt agent
   Loaded: loaded (/usr/lib/systemd/system/pxkrm-libvirt-agent.service; enabled; vendor preset: disabled)
   Active: active (exited) since Wed 2019-12-04 14:26:53 JST; 6min ago
  Process: 1490 ExecStart=/usr/sbin/pxkrm-libvirt-agent_ctl start (code=exited, status=0/SUCCESS)
 Main PID: 1490 (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/pxkrm-libvirt-agent.service
           mq1497 /usr/libexec/FJSVtcs/krm/libvirtagent

Dec 04 14:26:54 tcs-base-vm.fujitsu.com libvirtagent[1497]: [INFO] [krm] 9999...
Dec 04 14:26:54 tcs-base-vm.fujitsu.com libvirtagent[1497]: [INFO] [krm] 9999...
...

または、
# ps -ef | grep krm | grep libvirt        (psコマンドによる確認)
root      3178     1  0 Sep24 ?        00:33:05 /usr/libexec/FJSVtcs/krm/libvirtagent
root      3178     1  0 Sep24 ?        00:33:05 /usr/libexec/FJSVtcs/krm/libvirtexec
...
# shutdown -h now
...
(ここまで仮想マシン内)
# virsh undefine 〈ドメイン名〉
```

E.2 トラブルシューティング

ここでは、ジョブ実行環境を指定して投入したジョブが正常に終了しなかった場合の対処方法について説明します。現象が解消しない場合は管理者にお問い合わせください。

E.2.1 ジョブがPJMコード28で終了した

ジョブのPJMコード(終了コード)が28の場合、原因は以下が考えられます。

- ジョブ実行環境の指定の間違い。

  pjsubコマンドの-L jobenvオプションで指定したジョブ実行環境の名前を間違えている可能性があります。

  ```
  $ pjsub -L jobenv=container job.sh
  ※ ジョブ実行環境名containerが間違っている。
  ```

  ジョブ投入時に指定したジョブ実行環境名を間違えている場合は、正しいジョブ実行環境名を指定してください。

指定したジョブ実行環境名が正しい場合は、以下に示す問題がないかを管理者に問い合わせてください。

- コンテナイメージや仮想マシンイメージの設定の問題。(Dockerモード(SDI)、KVMモード(SDI))
  Dockerモードでのコンテナイメージや KVMモードでの仮想マシンイメージの設定などが誤っている可能性があります。
- コンテナイメージや仮想マシンイメージそのものの問題。(Dockerモード(SDI)、KVMモード(SDI))
  Dockerモードでのコンテナイメージや KVMモードでの仮想マシンイメージが不正である可能性があります。
- ジョブに割り当てたメモリの枯渇、またはMcKernelイメージの問題。(McKernelモード(SDI))

E.2.2 ジョブがPJMコード29で終了した

ジョブのPJMコード(終了コード)が29の場合、原因は以下が考えられます。

・ コンテナイメージの問題 (Dockerモード(UDI))

pjsubコマンドの-xオプションで環境変数PJM_JOBENV_DOCKER_IMAGEに指定したコンテナイメージのパスが間違っている、または指定したコンテナイメージが不正なファイル形式でないか確認してください。

```
$ pjsub -L jobenv=custom-docker -x PJM_JOBENV_DOCKER_IMAGE=/directory/my-docker.tar job.sh
※コンテナイメージのパス/directory/my-docker.tarが間違っている、または不正な形式である。
```

ファイル形式に問題がある場合は、docker exportコマンドで、指定したコンテナイメージをtar形式でアーカイブしてください。

コンテナイメージのパスおよびファイル形式に問題がない場合は、コンテナイメージによるコンテナの起動に失敗している可能性があります。指定したコンテナイメージを適切に修正してください。

・ McKernelイメージの問題 (McKernelモード(UDI))

pjsub コマンドの-xオプションで環境変数PJM_JOBENV_MCKERNEL_IMAGEを指定していない、または環境変数PJM_JOBENV_MCKERNEL_IMAGEで指定したMcKernelイメージファイルのパスが正しいかどうか確認してください。

```
$ pjsub -L jobenv=custom-mckernel job.sh
※環境変数PJM_JOBENV_MCKERNEL_IMAGEを指定していない。

$ pjsub -L jobenv=custom-mckernel -x PJM_JOBENV_MCKERNEL_IMAGE=/directory/my-mck.img job.sh
※環境変数PJM_JOBENV_MCKERNEL_IMAGEで指定したMcKernelイメージのパスが間違っている。
```

McKernelイメージの指定方法に問題がない場合は、McKernelイメージが不正で起動に失敗している可能性があります。McKernelイメージをMcKernelの公開サイトの情報に基づいて適切に修正してください。

・ 仮想マシンイメージの問題 (KVMモード(UDI))

pjsubコマンドの-xオプションで環境変数PJM_JOBENV_KVM_IMAGEを指定していない、または環境変数PJM_JOBENV_KVM_IMAGEで指定した仮想マシンイメージのパスが正しいかどうか確認してください。

```
$ pjsub -L jobenv=custom-kvm job.sh
※環境変数PJM_JOBENV_KVM_IMAGEを指定していない。

$ pjsub -L jobenv=custom-kvm -x PJM_JOBENV_KVM_IMAGE=/directory/my-kvm.img job.sh
※環境変数PJM_JOBENV_KVM_IMAGEで指定した仮想マシンイメージのパスが間違っている。
```

仮想マシンイメージの指定方法に問題がない場合は、仮想マシンイメージが不正で、libvirtを経由したQEMUによる仮想マシンの起動に失敗している可能性があります。仮想マシンイメージを要件に従って適切に修正してください。仮想マシンが起動するかどうか、事前に動作確認することを推奨します。

E.2.3 ジョブがPJMコード140で終了した

ジョブのPJMコード(終了コード)が140の場合、原因は以下が考えられます。

・ 同じノードに複数のKVMジョブが割り当てられました。(KVMモード)

ジョブが1ノードのノード専有ジョブになっているかどうか確認してください。

```
$ pjsub -L vnode=1,jobenv=custom-kvm -x PJM_JOBENV_KVM_IMAGE=/directory/my-kvm.img job.sh
※ 仮想ノードを割り当てるノード共有ジョブ(vnode=1)として投入している。
```

ノード専有ジョブとして実行するためには、物理ノードを割り当てるようにしてください(-L node=1)。PRIMERGYサーバの場合は、仮想ノードをSIMPLEXモード(-L vnode=1 -P exec-policy=simplex)で割り当てることでもノード専有ジョブになります。