SGI® UV2000 並列化プログラム利用の手引

2016/12/ 1

SGI Japan
HPC Technology Division / HPC Consulting
Professional Service Division

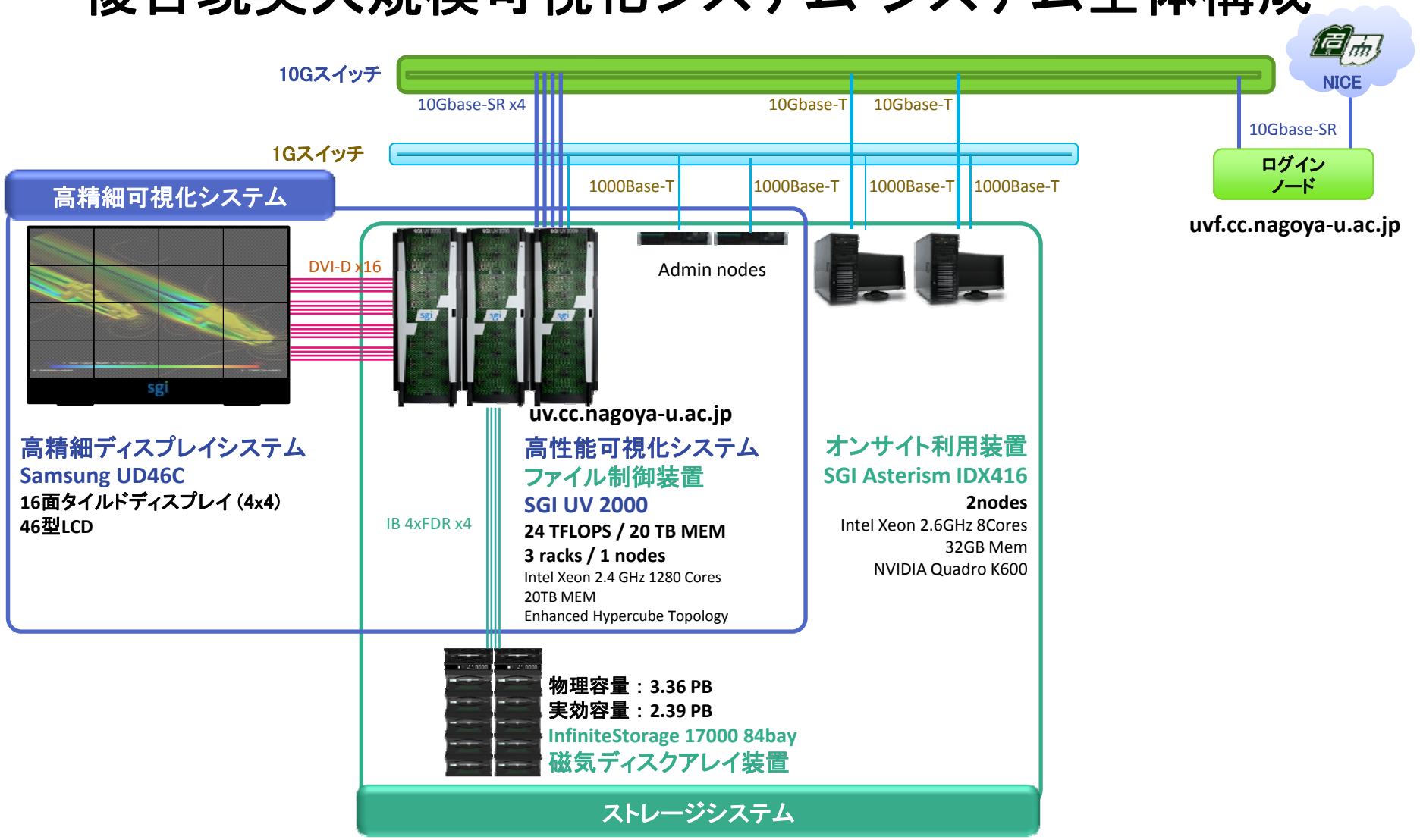
Agenda

Agenda

1. システム構成と利用方法
2. SGI UV2000アーキテクチャ
3. SGI UV2000におけるデータ配置と実行方法
4. 開発環境
5. コンパイルと実行
6. 最適化について
7. コンパイラオプション
8. 数値計算ライブラリ
9. デバッガ
10. 性能解析ツール
11. 並列化プログラミング
12. インテルコンパイラ自動並列化
13. OpenMPプログラム入門
14. MPIプログラム入門
15. ハイブリッドプログラミング

1. システム構成と利用方法

複合現実大規模可視化システム システム全体構成



高性能可視化システム HW/SW 構成

システム名称	SGI UV2000
CPU	Intel Xeon E5-4640
CPUクロック	2.40GHz
ソケット当りのコア数	8コア
キャッシュサイズ	20MB(8コアで共有)
全コア数/全ソケット数	1280コア/160ソケット
メモリ仕様 (ソケット当りのメモリバンド幅)	DDR3-1600 1600MHz x 4チャンネル/ソケット (51.2GBs/ソケット)
全主記憶容量 (ソケット当りの主記憶容量)	20TB (128GB/ソケット)
インターコネクト	NUMAlink6 (双方向6.7GB/s/チャンネル)
OS	SUSE Linux Enterprise Server 11 SP3 + SGI Performance Suite 1.9
コンパイラ	Intel Fortran/C++ Compiler 14.0 gcc/gfortran 4.3.4
数値計算ライブラリ	Intel Math Kernel Library(MKL) 11.1
MPIライブラリ	SGI MPT 2.11

ログイン方法

1. ログインノード (uvf) に sshでログインします。
   - ホスト名は uvf.cc.nagoya-u.ac.jp です。
   - 公開鍵認証
2. UV2000 (uv) にsshでログインします。
   - ホスト名は uv です。
   - パスワード認証

```
$ ssh username@uvf.cc.nagoya-u.ac.jp            ←  uvf に ssh でログイン

[username@uvf ~]$ ssh uv                                ←  uv に ssh でログイン
Password:                                               ←  パスワード認証 (passphraseでは無いことに注意)
Last login: Thu May  8 17:55:09 2014 from uvf.cc.nagoya-u.ac.jp


      Nagoya University Mixed Reality Large-scale Visualization System

                                    u v

            SGI UV 2000 160CPU/1280core 20TB Memory



[username@uv ~]$
```

ディスクの利用

- スパコンシステム共通
  - home
  - center
  - large
- UV専用領域
  - data領域

分類	ファイルシステム		総容量	備考
ユーザ割り当て	home	NFS		スパコンシステム共通 ホーム領域
	center	NFS		スパコンシステム共通 ソフトウェア、ユーティリティ
	large	NFS		スパコンシステム共通 データ領域
	data	xfs	2.4PB	UV専用データ領域 /data/usr/GROUP/USER を用意

PBSのキュー構成(暫定)と利用方法

- バッチジョブのキュー構成は下記の様に設定されています。

キュー名	キューへの割当		1ユーザ		並列数		メモリ		経過時間		備考
	コア数	メモリ	実行数	投入数	標準値	制限値	標準値	制限値	標準値	制限値	
uv-middle	512	7.2TB	1	4	64	128	900GB	1.8TB	24h	24h	デフォルト
uv-large	512	7.2TB	1	4	256	512	3.6TB	7.2TB	24h	24h	

- ジョブの投入　% qsub [*option*] <JOB_SCRIPT>

  -N　ジョブ名の指定
  -q　ジョブを投入するキューの指定
  -o　標準出力ファイルのPATHの指定
  -e　標準エラー出力ファイルのPATHの指定
  -l　ジョブ実行に必要なリソースの要求
  主なリソース　ncpus=(プロセッサ数の指定)
  mem=(最大物理メモリ容量)
  walltime=(ジョブを実行できる実際の経過時間)

PBSを通したジョブの実行

- ジョブスクリプト例：OpenMPプログラム（コンパイル済み）を実行する
  - qsubオプション部分 ：行の先頭に”#PBS”を記述します
    同じオプションをジョブ投入時に付加することをできます
  - それ以外の部分 ：シェルで実行されるコマンドとして扱われます

```
#!/bin/bash                          ← シェルを指定
#PBS -q uv-middle                        ← 投入するキューを指定
#PBS -o my-job.out                       ← 標準出力ファイルのPATHを指定
#PBS -e my-job.err                       ← 標準エラー出力ファイルのPATHを指定
#PBS -l select=1:ncpus=8             ← 必要なリソースの要求（8コア）
#PBS -N my-job                       ← 投入するジョブ名の指定

cd ${PBS_O_WORKDIR}                  ← 作業ディレクトリへ移動
export OMP_NUM_THREADS=8                 ← 並列度の設定
dplace -x2 -c0-7 ./a.out                 ← 実行（dplaceコマンドの説明は後述）
```

- ジョブの確認　% qstat
  ステータス　Q：実行待ち、R：実行中、E: 終了処理中、S：中断中
- ジョブの削除　% qdel <Job id>

PBSを通したインタラクティブジョブの実行

- インタラクティブジョブを実行する
  - qsub -I

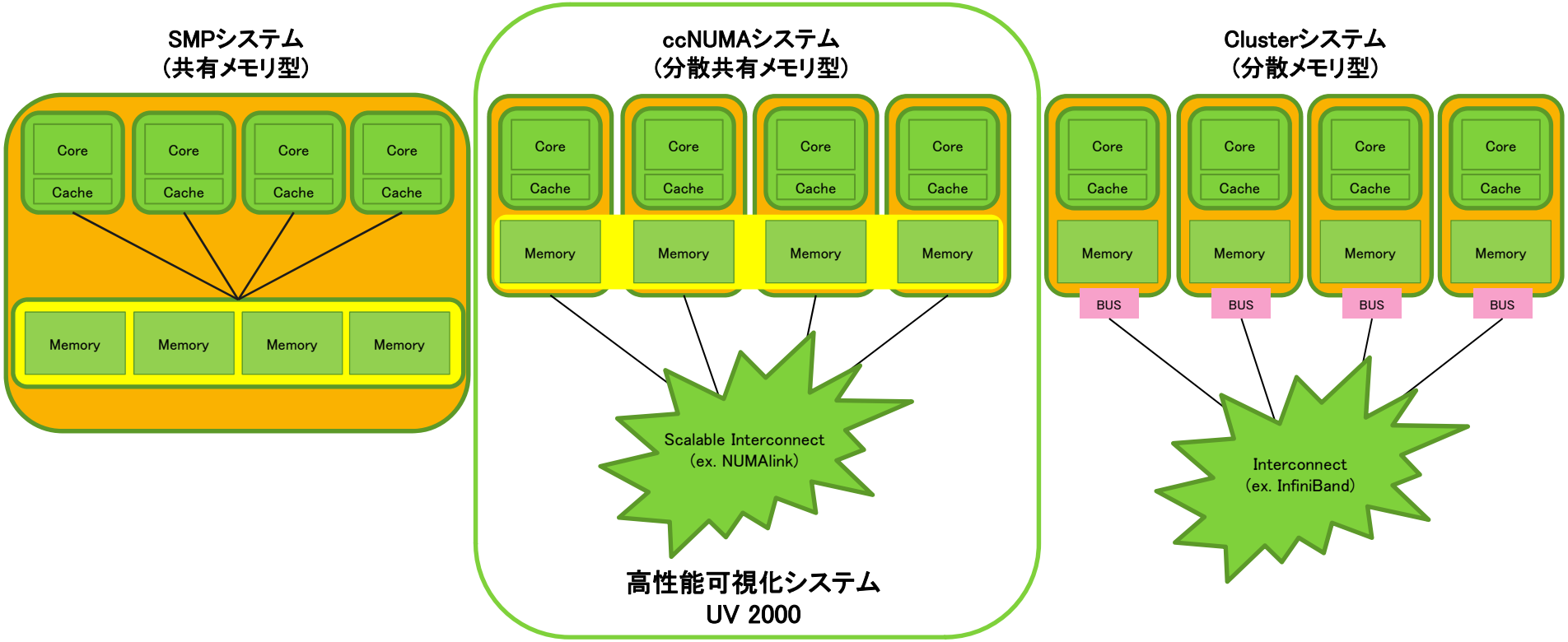
    qsubオプション部分 : -I: インタラクティブオプション

インタラクティブオプションをつけてキューに投入すると、ジョブの実行とともに入出力インタフェースがジョブ投入ウィンドウに返されます. ジョブを終了するには exit を入力します。

- qsub -I -q uv-middle -l select=1:ncpus=32 のように他のオプションを利用することも可能です。
- ジョブを終了するには exit を入力します。

2. SGI UV2000アーキテクチャ

一般的な並列計算機のアーキテクチャ



	SMP	ccNUMA	Cluster
特徴	• メモリを共有 • コンパイラによる自動並列化が可能	• ローカルにメモリを持つが論理的に共有可能 • 自動並列化が可能	• ローカルにメモリ • ノード間はデータ転送が必要
プログラミング	容易 （自動並列化、OpenMPが可能）	容易 （自動並列化、OpenMPが可能）	容易でない （自動並列化、OpenMPはノード内のみ）
H/W スケーラビリティ	中	高い	非常に高い

SGI UV2000構成要素

- 8 計算ブレード/IRU
- 16ソケット
- 128コア
- 2TB

- 4IRU/Rack
- 32計算ブレード
- 64ソケット
- 512コア
- 8TB

- 4IRU x 2Rack + 2IRU
- 80計算ブレード
- 160ソケット
- 1280コア
- 20TB

システム

ラック

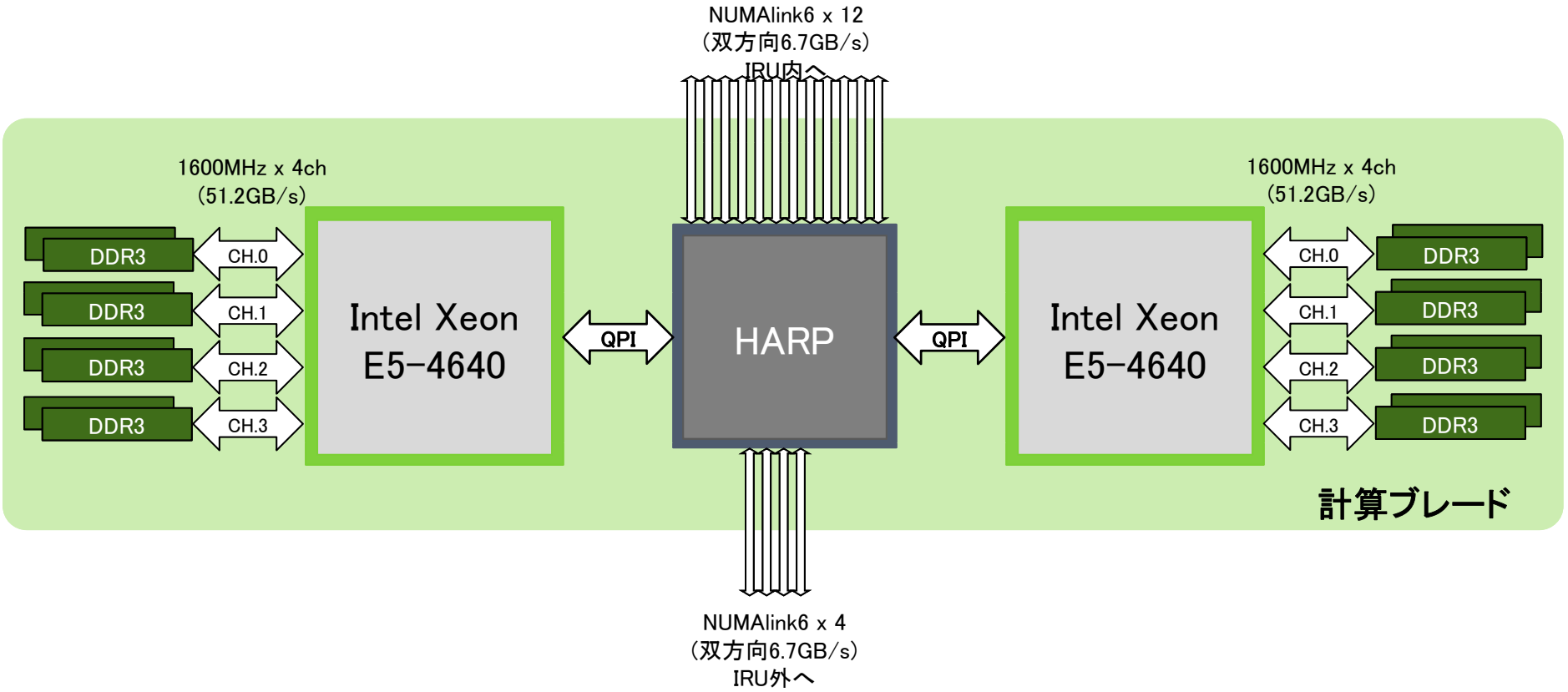
IRU

計算ブレード

- Intel Xeon E5-4640 x 2ソケット
- 16コア
- 256GB (128GB x 2)

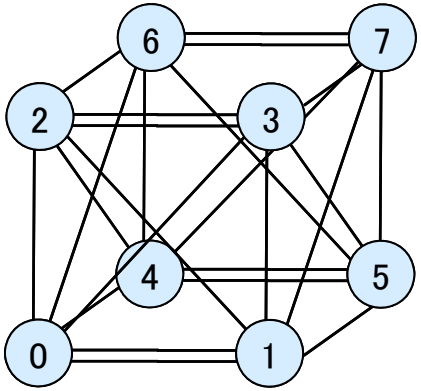
sgi

SGI UV2000の計算ブレード
ブロックダイアグラム



SGI UV2000のネットワークトポロジー

IRU内のトポロジー
(Enhanced Hyper-Cube Topology)

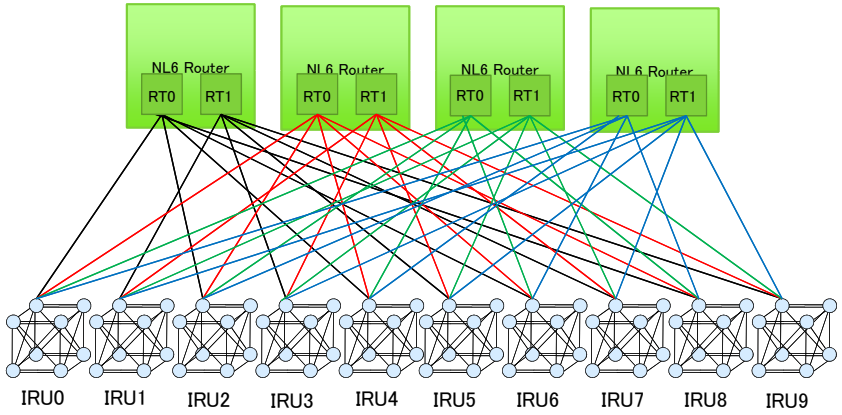


0　HARP ASICを表しています。

2本のNUMAlink6を示しています。

- 計算ブレードに一つHARP ASICがあり、2ソケットのCPUが接続されています。
- IRU内のHARPはDual Rail Enhanced Hyper-Cube Topologyで接続されています。

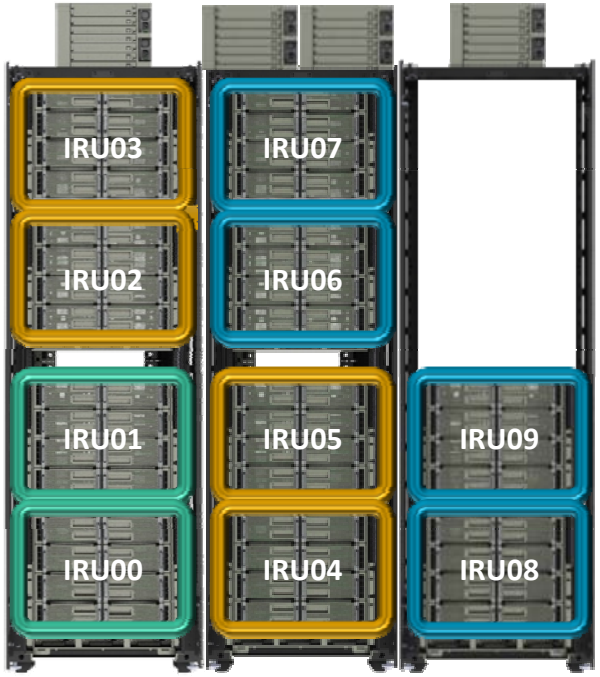
10IRUのトポロジー



- 各IRUのHyper-Cube上の同じ位置にあるHARP ASICについて図示しています。そのほかのHARP ASICについても同様に接続されています。
- 偶数番号のIRUにあるHARP ASICはそれぞれのNUMAlink6(NL6) RouterのRT0(Router0)に、奇数番号のIRUにあるHARPはRT1に接続されています。

SGI UV2000とPBSのキューの対応付け

- TSS会話型とPBSのキューは、UV2000システムの中で下記のように配置されます。

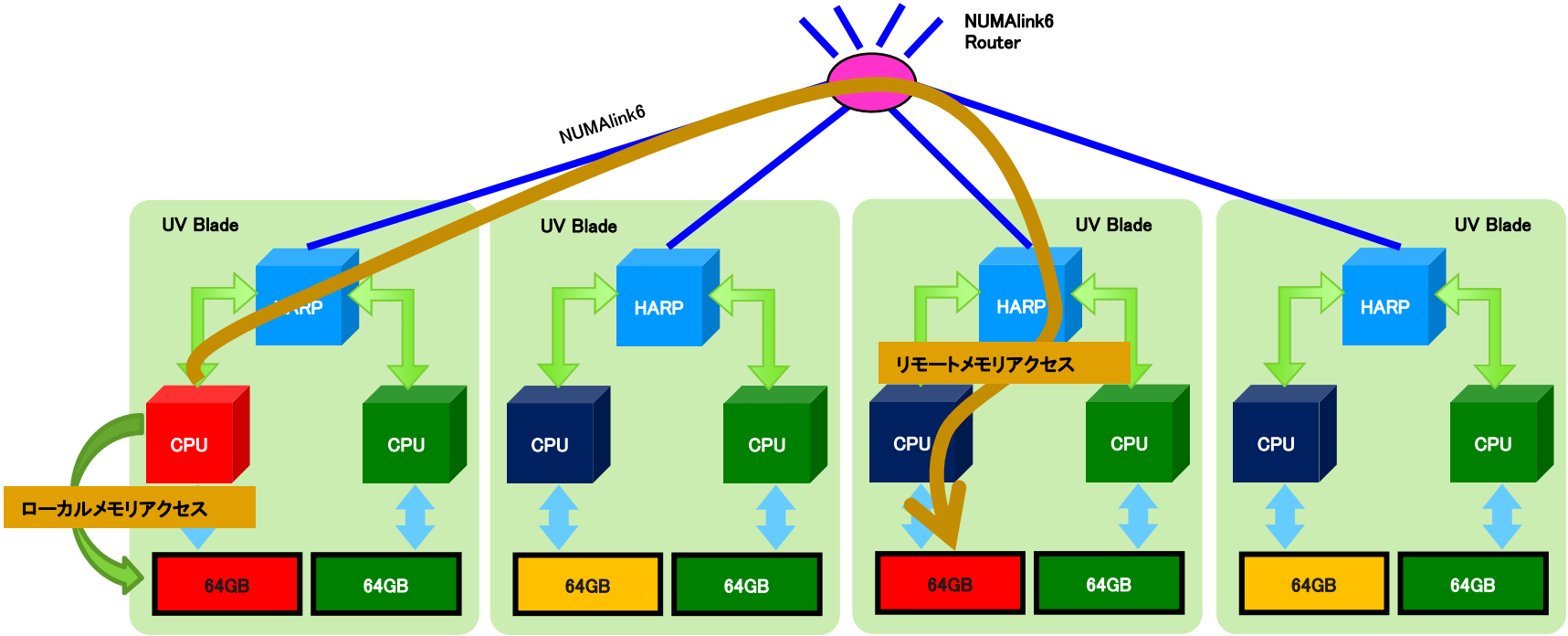


分類		キューへの割当		UV 2000 IRU番号
		コア数	メモリ	
TSS		256	4TB	IRU00～IRU01
PBS	uv-middle	512	7.2TB	IRU02～IRU05
	uv-large	512	7.2TB	IRU06～IRU09

3. SGI UV2000におけるデータ配置と実行方法

ファーストタッチ・ポリシー（SGI UV2000）

- 高性能可視化システム(SGI UV2000) は、ccNUMAアーキテクチャです。データはファーストタッチ・ポリシーでメモリに配置されます。
- ファーストタッチ・ポリシーとは、最初にデータに触れたコアのローカルメモリにデータが配置されます。
- NUMAアーキテクチャでは、特定のコアからみるとローカルメモリとリモートメモリがあります。
- データをできるだけローカルメモリに配置して計算することが高速化において必要です。
- プロセスをどこのコアに配置するかが重要になります。（dplaceまたはomplaceコマンド）



SGI UVにおける、ファーストタッチ・ポリシーの概念図

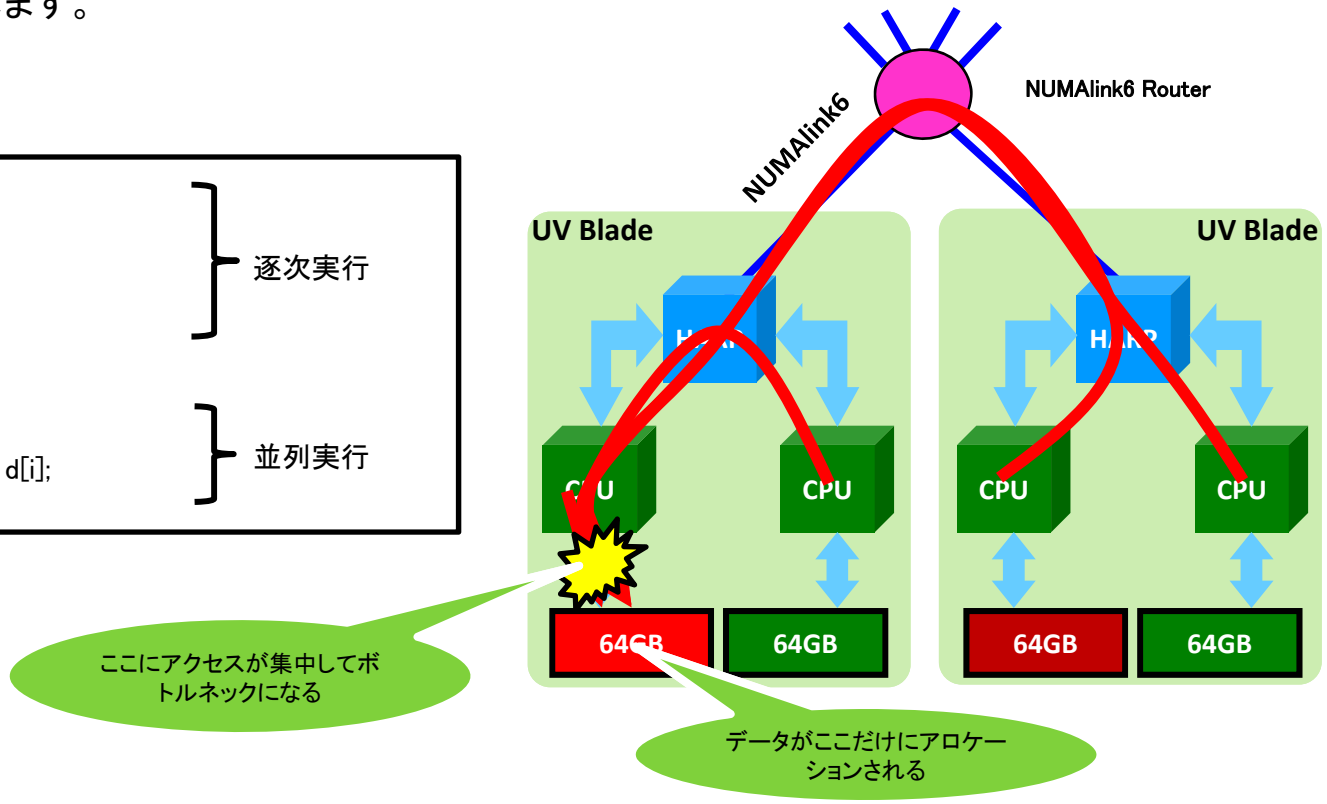
ccNUMAにおける並列化の留意点(1)

- 全てのデータは「ファーストタッチ」で(ページ単位で)メモリに配置されます。
- 初期化ループが逐次実行領域である場合、該当データは逐次実行したノードに配置されます。
- 並列実行領域で並列化されたループでは、全てのプロセッサから1ノードへのアクセスが集中してプログラムの性能が低下します。

```
        for( i=0; i<N; ++i){
                a[i]=0.0;
                b[i]=(double)i/2.0;
                c[i]=(double)i/3.0;
                d[i]=(double)i/7.0;
        }
#pragma omp parallel for
        for( i=0; i<N; ++i){
                a[i] = b[i] + c[i] + d[i];
        }
```

逐次実行

並列実行



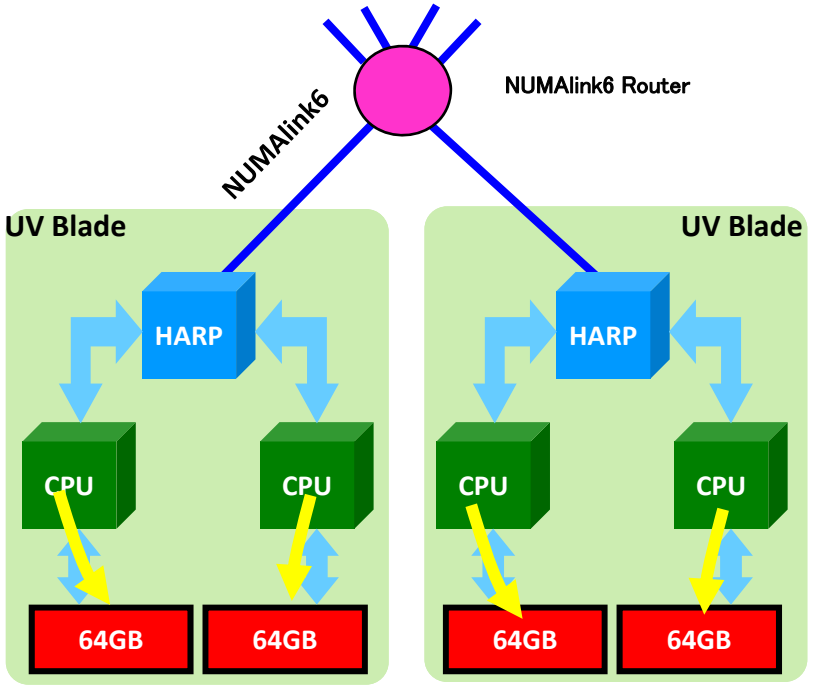
ccNUMAにおける並列化の留意点(2)

- 初期化ループを並列化します
  - 全てのデータは「ファーストタッチ」によりローカルなメモリに配置されます
- 並列実行領域では、各スレッドがローカルなメモリへアクセスすることになり、プログラムの性能が向上します

```
#pragma omp parallel for shared(a, b, ,c, d)
        for( i=0; i<N; ++i){
                a[i]=0.0;
                b[i]=(double)i/2.0;
                c[i]=(double)i/3.0;
                d[i]=(double)i/7.0;
        }
#pragma omp parallel for shared(a, b, c, d)
        for( i=0; i<N; ++i){
                a[i] = b[i] + c[i] + d[i];
        }
```

並列実行

並列実行

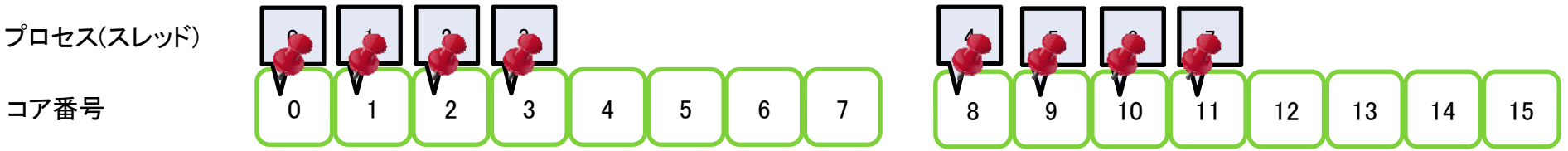


それぞれのローカルなメモリにデータがアロケーションされ、アクセスが分散される。

dplace/omplaceコマンド

- ファーストタッチで配置したデータを効率よく利用するために、プロセス/スレッドをコアに固定します。
- プロセスやスレッドをコアに固定するためには、dplaceまたはomplaceコマンドを使います。
  - プロセス(スレッド)が別のコアに移動してしまうことを防ぎます。
    - リモートメモリアクセスやキャッシュ利用の効率化
  - 並列化プログラムについてはオプションを用いて、管理プロセス(スレッド)の配置を抑止します。
    - 管理プロセス(スレッド)の配置を抑止することによって、計算プロセス(スレッド)を正しく配置します。
  - コアを独占できるわけではありません。
    - dplaceまたはomplaceコマンドで指定したコアには、さらに別のプロセス(スレッド)を重ねてしまうことができます。正しく指定することにより、性能劣化を防ぎます。

例. MPI 8並列を0-3, 8-11番のコアに配置する。
dplace -s1 -c0-3,8-11 ./a.out



dplace/omplaceコマンド利用方法

dplaceおよびomplaceコマンドでは、”-c”オプションでコア番号を指定してプロセス/スレッドを配置します。

- シリアルコード

  dplace -c7 ./a.out

  (“-c”オプションによりプロセスを配置するコア番号を指定します。ここでは、7番のコアにプロセスを配置します。)

- OpenMPコード

  dplace -x2 -c0-7 ./a.out

  (“-x2”オプションでOpenMPの管理スレッドを配置するスレッドから除外します。”-c”オプションにより0から7番のコアにスレッドを配置します。)

- MPIコード

  mpirun -np 8 dplace -s1 -c0-7 ./a.out

  (“-s1”オプションでMPIの管理プロセスを配置するプロセスから除外します。”-c”オプションにより0から7番のコアにプロセスを配置します。)

- Hybridコード

  mpirun -np 4 omplace -nt ${OMP_NUM_THREADS} -c 0-15 ./a.out

  (Hybridコードは、omplaceコマンドによりプロセスおよびスレッドを配置します。”-nt”オプションによりスレッド数を指定し、”-c”オプションで使用するコア番号を指定します。ここでは、0から15番のコアを指定しており、MPIプロセス数が4、OpenMPのスレッド数が4と設定された場合、MPIプロセスは0, 4, 8, 12番のコアに配置され、OpenMPスレッドはそれぞれのMPIプロセスと隣り合うコア(MPIランク0から生成されるOpenMPスレッドは0から3番のコア)に配置されます。)

dplace/omplaceコマンド利用方法

- 配置するコア番号の指定方法

dplace -c<*cpulist*>

cpulist	配置(コア番号)
0-3	0,1,2,3
0-7:2	0,2,4,6
0-1,4-5	0,1,4,5
0-3:2,8-9	0,2,8,9

omplace -nt ${OMP_NUM_THREADS} -c<*cpulist*>

cpulist	配置(コア番号)
0-N	0,1,2,3,..N(最後のコア番号)
1-:st=2	1,3,5,7,...(すべての奇数番号のコア)
0,1,1-4	0,1,1,2,3,4 (1番のコアに2つのプロセスを配置)
0-6:st=2,1-7:st=2	0,2,4,6,1,3,5,7
16-31:bs=2+st=4	16,17,20,21,24,25,28,29

バッチスクリプト例 – シリアルプログラム

```
#!/bin/bash
#PBS -N serial                          ← ジョブ名
#PBS -q uv-middle                       ← キュー名
#PBS -o stdout.log                      ← 標準出力ファイル
#PBS -e stderr.log                      ← 標準エラー出力ファイル
#PBS -l select=1:ncpus=8                ← リソースの確保(1ソケット/8コア)

cd ${PBS_O_WORKDIR}                     ← 作業ディレクトリへ移動

dplace -c7 ./a.out                      ← 実行
```

- dplace コマンドでプロセスを固定します。
- この例では、PBSによって1ソケット(8コア)を確保し、7番のコアにプロセスを配置します。
- シリアルプログラムでも、PBSで1ソケットを確保することで他のジョブの影響を少なくすることができます。
  - “#PBS -l select=1:ncpus=8”をして8コアを確保します。
- dplaceの”-c”オプションでコア番号を指定することができます。
  - PBSで1ソケット確保した場合、確保したリソースの中でコア番号が割り振られ、ジョブからは0から7番のコアが見えます。
  - dplaceコマンドのオプションで”-c7”と指定することで、7番のコアにプロセスを配置します。
- dplaceコマンドの”-c”オプションを指定しない場合、確保したコアの中で空いているコア(ほかのdplaceコマンドによってプロセスが配置されていないコア)の先頭からプロセスを配置します。

バッチスクリプト例 – OpenMPプログラム

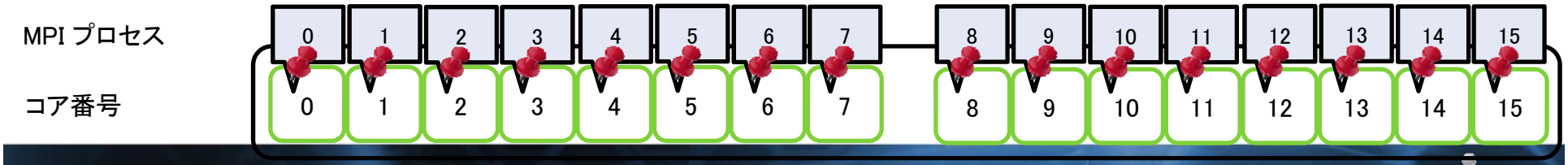
```
#!/bin/bash
#PBS -N openmp                          ← ジョブ名
#PBS -q uv-middle                       ← キュー名
#PBS -o stdout.log                      ← 標準出力ファイル
#PBS -e stderr.log                      ← 標準エラー出力ファイル
#PBS -l select=1:ncpus=16               ← リソースの確保(2ソケット/16コア)

cd ${PBS_O_WORKDIR}                     ← 作業ディレクトリへ移動

export KMP_AFFINITY=disabled            ← インテルのAffinityをdisabledにする
export OMP_NUM_THREADS=16               ← スレッド並列数の設定(16スレッド)

dplace -x2 -c0-15 ./a.out               ← 実行
```

- dplace コマンドでスレッドを固定します。
- KMP_AFFINITY=disabledに設定することでインテルコンパイラのスレッドバインド機能を無効します。
  - dplaceコマンドでスレッドを固定するため
  - dplaceコマンドを使わずにKMP_AFFINITYのみでスレッドをバインドすることも可能です。
- この例では、PBSによって2ソケット(16コア)を確保し、0-15番のコアにスレッドを配置します。dplaceの"-c"オプションでコア番号を指定することができます。
  - PBSで2ソケット確保した場合、確保したリソースの中でコア番号が割り振られ、ジョブからは0から15番のコアが見えます。
  - dplaceコマンドのオプションで"-c0-15"と指定することで、0-15番のコアにスレッドを配置します。
  - "-x2"オプションはインテルコンパイラでコンパイルしたOpenMPコードについて、管理スレッドをコアに配置しないためのオプションです。
    - 管理スレッドもコアに配置してしまうと、計算スレッドが適切に配置されずに性能が悪くなる場合があります。
- OpenMPの実行するスレッド数が8スレッド以下でも、PBSで1ソケットを確保することで他のジョブの影響を少なくすることができます。
  - "#PBS -l select=1:ncpus=8"をして8コアを確保します。
- dplaceコマンドの"-c"オプションを指定しない場合、確保したコアの中で空いているコア(ほかのdplaceコマンドによってプロセスおよびスレッドが配置されていないコア)の先頭からスレッドを配置します。
  - "-c"オプションを指定しない場合も、"-x2"オプションは必要です。

バッチスクリプト例 – MPIプログラム

```
#!/bin/bash
#PBS -N mpi                                  ← ジョブ名
#PBS -q uv-middle                            ← キュー名
#PBS -o stdout.log                           ← 標準出力ファイル
#PBS -e stderr.log                                 ← 標準エラー出力ファイル
#PBS -l select=1:ncpus=16:mpiprocs=16                    ← リソースの確保(2ソケット/16コア)
source /etc/profile.d/modules.sh
module load mpt
cd ${PBS_O_WORKDIR}                          ← 作業ディレクトリへ移動

mpiexec_mpt -np 16 dplace -s1 -c0-15 ./a.out     ← 16並列で実行
```

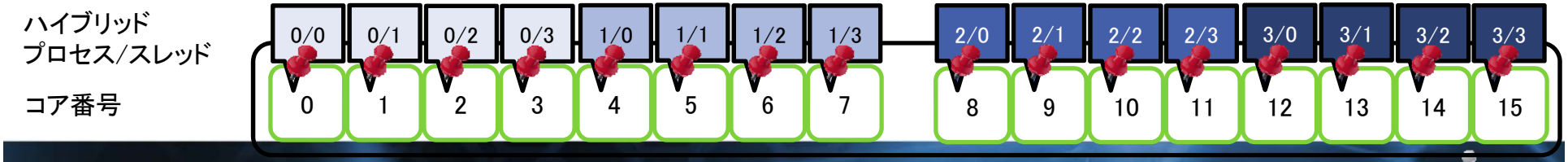
- dplace コマンドでプロセスを固定します。
- この例では、PBSによって2ソケット(16コア)を確保し、0-15番のコアにプロセスを配置します。dplaceの"-c"オプションでコア番号を指定することができます。
  - PBSで2ソケット確保した場合、確保したリソースの中でコア番号が割り振られ、ジョブからは0から15番のコアが見えます。
  - dplaceコマンドのオプションで"-c0-15"と指定することで、0-15番のコアにスレッドを配置します。
  - "-s1"オプションはMPIライブラリとしてSGI MPTを用いたMPIプログラムについて、管理プロセスをコアに配置しないためのオプションです。
    - 管理プロセスもコアに配置してしまうと、計算プロセスが適切に配置されずに性能が悪くなる場合があります。
- MPIプログラムの実行するプロセス数が8プロセス以下でも、PBSで1ソケットを確保することで他のジョブの影響を少なくすることができます。
  - "#PBS -l select=1:ncpus=8"をして8コアを確保します。
- dplaceコマンドの"-c"オプションを指定しない場合、確保したコアの中で空いているコア(ほかのdplaceコマンドによってプロセスが配置されていないコア)の先頭からプロセスを配置します。
  - "-c"オプションを指定しない場合も、"-s1"オプションは必要です。



バッチスクリプト例 – ハイブリッドプログラム

```
#!/bin/bash
#PBS -N hybrid                                          ← ジョブ名
#PBS -q uv-middle                                       ← キュー名
#PBS -o stdout.log                                      ← 標準出力ファイル
#PBS -e stderr.log                                      ← 標準エラー出力ファイル
#PBS -l select=1:ncpus=16:mpiprocs=4                    ← リソースの確保(2ソケット/16コア)

source /etc/profile.d/modules.sh
module load mpt
cd ${PBS_O_WORKDIR}                                     ← 作業ディレクトリへ移動
export KMP_AFFINITY=disabled                            ← インテルのAffinityをdisabledにする
export OMP_NUM_THREADS=4                                ← スレッド並列数の設定(4スレッド)
mpiexec_mpt -np 4 omplace -nt ${OMP_NUM_THREADS} -c 0-15 ./a.out    ← MPI=4プロセス x OpenMP=4スレッドで実行
```

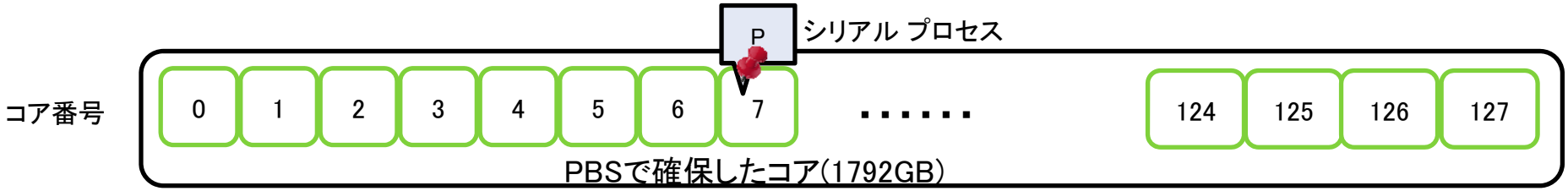
- omplace コマンドでプロセスおよびスレッドを適切に固定します。
- KMP_AFFINITY=disabledに設定することでインテルコンパイラのスレッドをバインド機能を無効します。
  - omplaceコマンドでプロセスおよびスレッドを固定するため
- この例では、PBSによって2ソケット(16コア)を確保し、0-15番のコアにプロセスおよびスレッドを配置します。omplaceの”-c”オプションでコア番号を指定することができます。
  - PBSで2ソケット確保した場合、確保したリソースの中でコア番号が割り振られ、ジョブからは0から15番のコアが見えます。
  - omplaceコマンドのオプションで”-c 0-15”と指定することで、0-15番のコアにプロセスおよびスレッドを適切に配置します。
  - 管理プロセスおよびスレッドはomplaceで自動的に排除され、計算プロセスおよびスレッドのみが配置されます。
- ハイブリッドプログラムの実行する並列数が8以下でも、PBSで1ソケットを確保することで他のジョブの影響を少なくすることができます。
  - “#PBS -l select=1:ncpus=8”をして8コアを確保します。
- omplaceコマンドの”-c”オプションを指定しない場合、確保したコアの中で空いているコア(ほかのdplaceまたはomplaceコマンドによってプロセスおよびスレッドが配置されていないコア)の先頭からプロセスおよびスレッドを配置します。



バッチスクリプト例
– シリアル大規模メモリプログラム

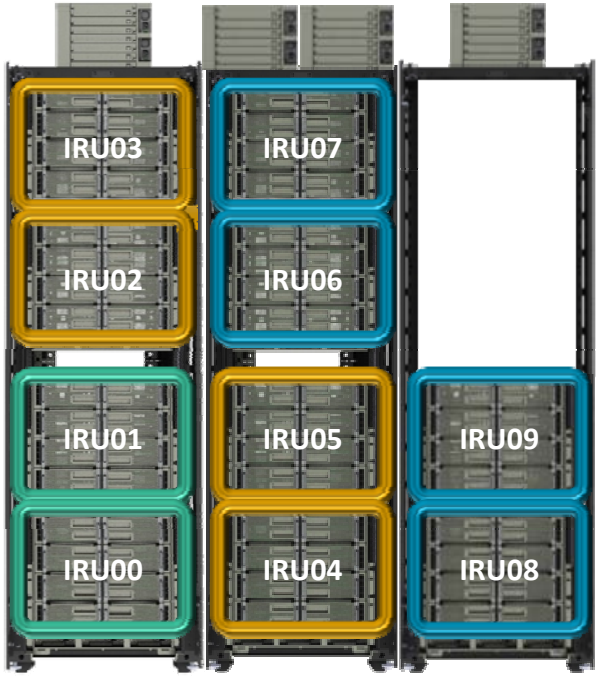
```
#!/bin/bash
#PBS -N serial                                  ← ジョブ名
#PBS -q uv-middle                               ← キュー名
#PBS -o stdout.log                              ← 標準出力ファイル
#PBS -e stderr.log                              ← 標準エラー出力ファイル
#PBS -l select=1:ncpus=8:mem=1800gb             ← リソースの確保(1ソケット/8コア, 1800GB)

cd ${PBS_O_WORKDIR}                             ← 作業ディレクトリへ移動

dplace -c7 ./a.out                              ← 実行
```

- dplace コマンドでプロセスを固定します。
- ここでは、8コア、1792GBのリソースを確保します。
  - “#PBS -l select=1:ncpus=8:mem=1800gb”としてリソースを確保します。
  - “ncpus=8”として8コアを要求していますが、PBSの設定により1CPUあたり112.5GB利用可能となっているため、”mem=1800gb”でメモリを1800GB確保するために、PBSは1800/112.5=16ソケット(128コア)を確保します。
- この例では7番目のコアにプロセスを固定します。
- プロセスが実行されるソケット以外のメモリへはリモートメモリアクセスになるため、ローカルメモリへのアクセスに比べて遅くなります。
- 可能であれば、プログラムの並列化を行いリモートメモリアクセスを減らすことをお勧めします。



UV2000上のジョブの配置確認方法

- ndstatコマンドでPBSのジョブがどこで流れているか確認できます。
  - ndstat は 計算ノードの使用状況を確認するコマンドです
  - UV2000におけるジョブの配置はジョブIDの下3桁を表示します




```
[sgise4@uv:~]$ ndstat
JOBID    USER     QUEUE    JOBNAME    NODE REQTIME  STAT  ELAPSE  START_TIME
-------- -------- -------- ---------- ---- -------  ----- -------  ----------
275      sgise4   uv-large mpi-genlte  512   08:00     R   00:02  May 18 13:35
279      w48684a  uv-middl mpi-genlte  128   08:00     R   00:05  May 18 13:32
280      w48684a  uv-middl mpi-genlte  128   08:00     Q      --  --
282      sgise4   uv-middl mpi-genlte  128   08:00     R   00:01  May 18 13:36
283      sgise4   uv-middl mpi-genlte  128   08:00     Q      --  --
284      sgise4   uv-middl mpi-genlte  128   08:00     Q      --  --
292      w48684a  uv-middl mpi-genlte  128   08:00     Q      --  --
293      w48684a  uv-middl mpi-genlte  128   08:00     Q      --  --

 IRU  |  0     1     2     3     4     5     6     7
------|-------------------------------------------------
 000  |  --    --    --    --    --    --    --    --
 000  |  --    --    --    --    --    --    --    --
 001  |  --    --    --    --    --    --    --    --
 001  |  --    --    --    --    --    --    --    --
 002  | *279  *279  *279  *279  *279  *279  *279  *279
 002  | *279  *279  *279  *279  *279  *279  *279  *279
 003  | *282  *282  *282  *282  *282  *282  *282  *282
 003  | *282  *282  *282  *282  *282  *282  *282  *282
 004  |  --    --    --    --    --    --    --    --
 004  |  --    --    --    --    --    --    --    --
 005  |  --    --    --    --    --    --    --    --
 005  |  --    --    --    --    --    --    --    --
 006  | *275  *275  *275  *275  *275  *275  *275  *275
 006  | *275  *275  *275  *275  *275  *275  *275  *275
 007  | *275  *275  *275  *275  *275  *275  *275  *275
 007  | *275  *275  *275  *275  *275  *275  *275  *275
 008  | *275  *275  *275  *275  *275  *275  *275  *275
 008  | *275  *275  *275  *275  *275  *275  *275  *275
 009  | *275  *275  *275  *275  *275  *275  *275  *275
 009  | *275  *275  *275  *275  *275  *275  *275  *275
```

TSS

uv-middle

uv-large

4. 開発環境

開発環境

システム名称	SGI UV2000
OS	SLES 11.3 & SGI Performance Suite 1.9
Fortran コンパイラ	Intel Fortran Compiler XE 14.0
C++/C コンパイラ	Intel C++ Compiler 14.0
数値計算ライブラリ	Intel Math Kernel Library 11.1
MPIライブラリ	SGI MPT 2.11
デバッガ	Intel Debugger, GNU Debugger
プロファイルツール	PerfSuite Perfcatcher, MPInside

5. コンパイルと実行

コンパイルコマンド

- インテルコンパイラのコンパイルコマンド
  - ifort　(Fortran77, 90, 95, 2003をサポート)
  - icc　(ISO標準 C をサポート)
  - icpc　(ISO標準 C++ サポート)
- オプション一覧を表示

- バージョンを表示

```
$ ifort -help
```

- コンパイル

```
$ ifort -v
ifort version 14.0.2
```

```
$ ifort  sample.f
```

※ C/C++の場合はコマンドを置き換えてください。

プログラムのコンパイルと実行

- シリアルプログラム

```
$ ifort prog.f                                  （コンパイル）
$ dplace ./a.out                                （実行）
```

- OpenMPプログラム

```
$ ifort -openmp prog.f                                  （コンパイル）
$ export KMP_AFFINITY=disabled    （インテルコンパイラのバインド機能の無効化）
$ export OMP_NUM_THREADS=4                              （スレッド数設定）
$ dplace -x2 ./a.out                                    （実行）
```

- MPIプログラム

```
$ ifort prog.f -lmpi                            （コンパイル）
$ mpirun -np 4 dplace -s1 ./a.out               （実行）
```

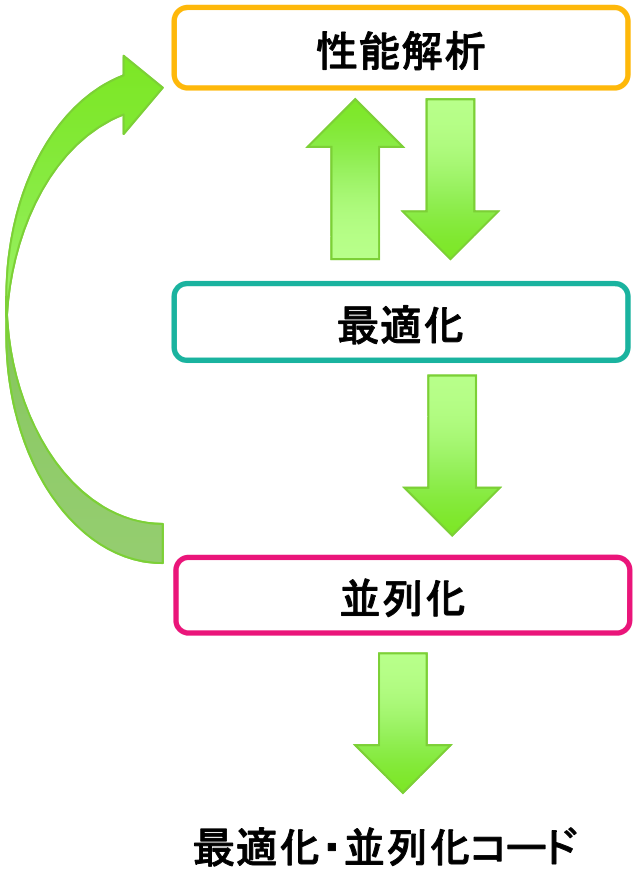
- MPI+OpenMP ハイブリッドプログラム

```
$ ifort -openmp prog.f -lmpi                            （コンパイル）
$ export KMP_AFFINITY=disabled    （インテルコンパイラのバインド機能の無効化）
$ export OMP_NUM_THREADS=4                              （スレッド数設定）
$ mpirun -np 4 omplace -nt ${OMP_NUM_THREADS} ./a.out   （実行）
```

6. 最適化について

最適化・並列化手順

- アプリケーションプログラムの高速化を検討する際は、一般に次のような手順で最適化・並列化を行います。
  - 性能解析ツールを使用して、プログラムのボトルネックになっている部分やその原因を特定する。
  - 1プロセッサでの高速化（最適化）を検討する。
  - 最適化したプログラムの並列化を検討する。
- 一般には、この手順を繰り返すことによって高い性能が得られます。



最適化・並列化手順

- プログラム最適化には様々の方法があります。
  - コンパイラオプション
  - 最適化されたライブラリ
  - コード修正による最適化

- 並列化にも様々な方法があります。
  - 自動並列化
  - OpenMP指示行
  - MPI
  - ハイブリッド (例. MPI + OpenMP)

7. コンパイラオプション

推奨するコンパイラオプション

- デフォルトで設定されている主なオプション

オプションの種類	オプション	オプションのレベル
最適化レベル	-O2	パフォーマンス向上のための最適化を行ないます。
特定のプロセッサ向けの最適化	-msse2	インテルプロセッサ向けにSSE2およびSSE命令を生成し、SSE2対応のインテルXeonプロセッサ向けの最適化をします。

- 推奨するオプション

オプションの種類	オプション	オプションのレベル
最適化レベル	-O3	-O2に加えプリフェッチ、スカラー置換、ループ変換、およびメモリアクセス変換などのより強力な最適を有効にします。
特定のプロセッサ向けの最適化	-xAVX	インテルAVXベクトル化命令および、SSE4.2、SSSE3, SSE3, SSE2, SSE命令を生成し、インテルXeon E5-2600番台および4600番台のプロセッサ向けに最適化をします。

最適化レベルオプション

オプション	内容
-O0	全ての最適化を無効とします。主にデバッグ時に利用。
-O1	・グローバルな最適化を有効化 ・組み込み関数の認識と組込み関数のインライン展開の無効 この最適化レベルでは、分岐が多く、実行時間の多くがループではないコードの性能向上が見込めます。
-O2	デフォルトの最適化レベル。最適化レベルを指定しない場合、この最適化レベルが適用されます。この最適化レベルでは次の最適化を行います。 ・インライン展開 ・定数伝播 ・コピー伝播 ・不要コードの削除 ・グローバルレジスタの割り当て ・グローバル命令スケジューリング ・スペキュレーション・コントロール ・ループのアンロール ・コード選択の最適化
-O3	-O2オプションに加えて、プリフェッチ、スカラー置換、キャッシュ・ブロッキング、ループ変換、メモリアクセス変換などの最適化を行います。 浮動小数点演算の多いループや大きなデータセットを処理するコードで性能向上が見込めます。 -axSSE4.2および-xSSE4.2オプションとの組み合わせでより詳細なデータ依存性解析をします。
-fast	-xHOST -O3 -ipo -no-prec-div -staticを有効にするマクロオプションです。 ※-fastオプションには-staticオプションが含まれるため、ダイナミック・ライブラリしか提供されていないライブラリを利用する場合、-Bdynamicオプションでそのライブラリを指定する必要があります。

最適化に関するオプション

オプション	内容
-x*プロセッサ*	*プロセッサ*で指定した特定のプロセッサ向けのバイナリを生成します。
-ax*プロセッサ*	*プロセッサ*で指定した特定のプロセッサ向けのバイナリと一般的なIA32アーキテクチャ向けのバイナリを一つのバイナリで生成します。
-vec	ベクトル化を有効/無効にします。デフォルトは有効。
-vec-report	ベクタライザーからのメッセージをコントロールします。 デフォルトではベクタライザーからのメッセージは出力されません。ベクタライザーからのメッセージを出力するためには、このオプションを有効にしてください。
-no-prec-div	IEEE準拠の除算よりも多少精度が低くなる場合がありますが、最適化を試みます。

特定のプロセッサ向けの最適化オプション

-ax*プロセッサ*
-x*プロセッサ*　：特定のプロセッサ向けの最適化を行います。

プロセッサ	特定のプロセッサ向けの最適化を行います。
HOST	コンパイルをしたプロセッサで利用可能な、最も高いレベルの命令を生成し、そのプロセッサ向けの最適化を行います。
AVX	SandyBridge(Intel Xeon E5-2600番台および4600番台)向けの最適化を行い、AVX命令を生成します。さらに、SSE4.2、SSE4、SSSE3、SSE3、SSE2、SSE命令を生成し、インテルAVX命令をサポートするプロセッサ向けの最適化を行います。
SSE4.2	Nehalem-EP(Intel Xeon 5500番台および5600番台)向けの最適化を行い、SSE4.2命令を生成します。さらに、SSE4のベクトル化コンパイル命令、メディア・アクセラレター、SSSE3, SSE3, SSE2, SSE命令を生成し、インテルCoreプロセッサ向け最適化を行います。
SSE4.1	SSE4のベクトル化コンパイル命令、メディア・アクセラレター、SSSE3, SSE3, SSE2, SSE命令を生成し、45nmプロセスルール世代のインテルCoreプロセッサ(Intel Xeon 5200番台、5400番台)向け最適化を行います。
SSSE3	SSSE3, SSE3, SSE2, SSE命令を生成し、インテルCore2 Duoプロセッサ(Intel Xeon 5100番台、5300番台)向け最適化を行います。
SSE3	SSE3, SSE2, SSE命令を生成し、インテルNetburstマイクロアーキテクチャ向け(Intel Xeon 5000番台)最適化を行います。

最適化に関するオプション(1)

- プロシージャ間解析の最適化

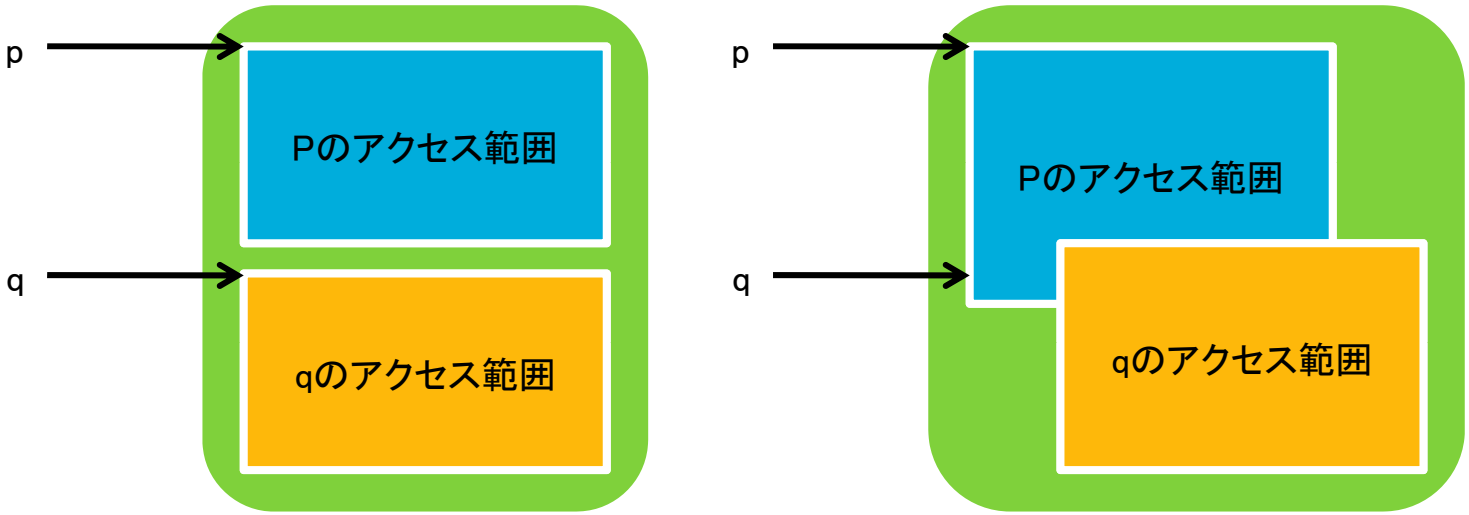
オプション	内容
-ip	1つのソースファイルにあるプロシージャ間の解析、最適化を行います。
-ipo	複数のソースファイルにあるプロシージャ間の解析、最適化を行います。リンク時にもオプションとして指定してください。

- 浮動小数点演算に関するオプション

オプション	内容
-ftz	アンダーフローが発生したときに値をゼロに置き換えます。 デフォルトでは、最適化レベルが-O1, -O2, -O3のとき、このオプションが有効になっています。最適化レベルが-O0のとき、このオプションは無効になっています。 このオプションが数値動作で好ましくない結果を出力した場合、-no-ftzオプションでアンダーフローが発生したときに値をゼロにフラッシュしなくなります。
-fltconsistency	浮動小数点の一貫性を向上させ、IEEE754規格に則った浮動小数点演算コードを生成します。

最適化に関するオプション(2)

オプション	内容
-falias -fno-alias -ffnalias -fno-fnalias	複数のポインタが同じメモリ領域を参照する（エイリアスがある）かどうかを、コンパイラに指示する。エイリアスがない場合、データ依存性問題の発生する可能性がないため、コンパイラは積極的な最適化を行うようになります。特に C/C++ コードの最適化に効果を発揮します。 ソースコードを書き換えてよいなら、ポインタに _restrict を使用することもできます(お勧め)。 エイリアスがある場合、このオプションを使うと正しい結果が得られません。 エイリアスがないことを利用者が認識している場合にのみ有効です。



最適化レポート

オプション	内容
-opt-report [n]	最適化レポートを標準エラー出力に表示 n=0 : disable optimization report output n=1 : minimum report output n=2 : medium output (DEFAULT) n=3 : maximum report output
-opt-report-file=name	最適化レポートをnameというファイルに出力
-opt-report-routine=name	nameで指定されたサブルーチンのレポートのみを出力
-opt-report-phase=name	nameで指定された最適化フェーズのレポートを出力
-opt-report-help	最適化レポート生成可能な最適化機構フェーズを表示

※最適化のフェーズについて

最適化フェーズ	最適化の内容	関連するオプション
ipo	Interprocedural Optimizer	-ipo, -ip
hlo	High-level Language Optimaizer	-O3 (Loop Unrolling)
ilo	Intermediate Language Scalar Optimizer	
hpo	High Performance Optimizer	
pgo	Profile Guided Optimizer	-prof_gen, -prof_use

リンクに関するオプション

オプション	内容
-static	スタティックライブラリをリンクします。スタティックライブラリが無い場合はエラーになります。
-Bstatic	スタティックライブラリを指定します。
-Bdynamic	ダイナミックライブラリを指定します。
-shared-intel	インテルのダイナミックライブラリをリンクします。
-static-intel	インテルのスタティックライブラリをリンクします。

Intel 64におけるメモリモデル

- Intel Compilerでは、および64ビットのバイナリは異なります。
- Intel64 メモリモデル
  - small(デフォルト)コードとデータは最初の2GBのメモリ空間に制限されます。
  - medium(-mcmodel=medium)コードは最初の2GBのメモリ空間に制限されますが、データは制限されません。
  - large(-mcmodel=large)コードもデータも制限されません。
- Intel64 アーキテクチャはの2GBの制限は、2GBを超える配列だけでなく、合計が2GBを超える共通ブロックとローカルデータにも適用されます。

データ変換 (FORTRANのみ)

- バイナリデータのエンディアン
  - Xeon, Opteron: Little Endian
  - Sparc, Power, SX　: Big Endian
- Big Endianバイナリファイルを Xeonのシステムで読み込むにはエンディアンの変換が必要です。
- コンパイルオプションによる変換

```
-convert big_endian
```

- 環境変数による変換
  - すべてのファイルに対してビックエンディアンに変換

```
$ export F_UFMTENDIAN=big
```

  - ユニット番号10, 20のみをビックエンディアンに変換

```
$ export F_UFMTENDIAN=big:10,20
```

  - ユニット番号10から20をビックエンディアンに変換

```
$ export F_UFMTENDIAN=10-20
```

  - 拡張子(.DAT)を指定してビックエンディアンに変換

```
$ export FORT_CONVERT.DAT=BIG_ENDIAN
```

8. 数値計算ライブラリ

Intel Math Kernel Library (MKL) 概要

- 特徴
  - 科学技術計算向け
  - インテルプロセッサ向けにチューニング
  - マルチスレッド対応
    - スレッド並列化
    - スレッドセーフ
  - 自動ランタイム・プロセッサ検出機能
  - CおよびFortranのインターフェイス

Intel Math Kernel Library (MKL) 内容

- Intel Math Kernel Library は、以下の機能が含まれます。
  - BLAS
  - BLACS
  - LAPACK
  - ScaLAPACK
  - PBLAS
  - Sparse Solver
  - Vector Math Library (VML)
  - Vector Statistical Library (VSL)
  - Conventional DFTs and Cluster DFTs
  - etc.

Intel Math Kernel Library (MKL)

- MKLをリンクする方法

シリアル版の場合:
$ ifort -o test test.f -lmkl_intel_lp64 -lmkl_sequential -lmkl_core

スレッド版の場合:
$ ifort -o test test.f -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5

・インテルコンパイラのオプション-mklでMKLをリンクすることもできます。

シリアル版の場合:
$ ifort -o test test.f -mkl=sequential

スレッド版の場合:
$ ifort -o test test.f -mkl=parallel

Intel Math Kernel Library (MKL)

- BLACSおよびScaLAPACKの利用方法

```
シリアル版の場合:
$ ifort -lmkl_scalapack_lp64 -lmkl_blacs_sgimpt_lp64 ¥
  -lmkl_intel_lp64 -lmkl_sequential -lmkl_core example1.f -lmpi

スレッド版の場合:
$ ifort -lmkl_scalapack_lp64 -lmkl_blacs_sgimpt_lp64 ¥
  -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 example1.f -lmpi
```

・インテルコンパイラのオプション-mklでMKLをリンクすることもできます。

```
シリアル版の場合:
$ ifort -lmkl_scalapack_lp64 -lmkl_blacs_sgimpt_lp64 ¥
  -mkl=sequential example1.f -lmpi

スレッド版の場合:
$ ifort -lmkl_scalapack_lp64 -lmkl_blacs_sgimpt_lp64 ¥
  -mkl=parallel example1.f -lmpi
```

Intel Math Kernel Library (MKL)

- スレッド並列版MKLを使う場合は注意が必要です

シリアルで実行	環境変数OMP_NUM_THREADSを1に設定します。または、シリアル版MKLをリンクします。
スレッド並列で実行	環境変数OMP_NUM_THREADSを並列実行数に設定します。OpenMPのプログラム中でMKLを使う場合、OMP_NUM_THREADSで設定されたスレッド数で実行されます。また、OpenMPのスレッド数とは違うスレッド数で実行したい場合はOMP_NUM_THREADS以外にMKL_NUM_THREADSを設定します。 ※OpenMPで並列化されたループ内でMKLのスレッド並列化された関数を用いる場合、デフォルトではOpenMPのネストが無効になっているため、MKLのスレッド並列は無効です。環境変数OMP_NESTEDを”yes”とすることにより、MKLのスレッド並列を有効にすることが可能です。
MPIで実行	MPIのみで並列実行する場合、MKLがスレッド並列で動作しないように環境変数OMP_NUM_THREADSを1に設定します。または、シリアル版MKLをリンクします。
ハイブリッドで実行	MPIとスレッド並列のハイブリッドでの実行をする場合、MKLのスレッド数をOMP_NUM_THREADSまたはMKL_NUM_THREADSで設定します。

9. デバッガ

デバッガ

- 以下のデバッガをご利用いただけます。

- gdb – GNU Debugger
  - Linux標準のデバッガ
  - マルチスレッド対応(OpenMP, pthread)

- idbc – Intel Debugger
  - Intel Compilerに付属のデバッガ
  - マルチスレッド対応(OpenMP, pthread)
  - インタフェイスを変更可(dbx風、gdb風)
  - GUI対応(idb)

(使用例)

- コアファイルの解析

```
% idbc ./a.out core
(idb)where
(idb)w
```

- idbからのプログラムの実行

```
% idbc ./a.out
(idb) run
```

- 実行中のプロセスへのアタッチ

```
% idbc -pid [process id] ./a.out
% gdb a.out [process id]
```

デバッグに関するオプション

オプション	内容
-g	オブジェクトファイルにデバッグ情報を生成します。最適化レベルオプション-Oが明示的に指定されていない場合、最適化レベルは-O0になります。
-traceback -g	デバッグのために必要な情報をオブジェクトファイルに埋め込みます。 Segmentation Faultなどのエラー終了時にエラーの発生箇所を表示します。
-check bounds -traceback -g	実行時に配列の領域外参照を検出します。2つのオプションと-gオプションを同時に指定してください
-fpe0 -traceback -g	浮動小数点演算の例外処理を検出します。2つのオプションと-gオプションを同時に指定してください。
-r8	real/compelx型で宣言された変数をreal*8/complex*16型の変数として取り扱います。
-i8	integer型で宣言された変数をinteger*8型の変数として取り扱います。
-save -zero	変数を静的に割り当て、ゼロで初期化します。

10. 性能解析ツール

性能解析ツール

- プログラムのホットスポットやボトルネックを検出するための性能解析ツールを用意しています。
  - シリアルプログラムだけでなく、OpenMPやMPIによる並列プログラムの性能解析も可能。
  - MPI通信の解析も可能。
  - 性能解析ツール
    - PerfSuite
  - MPI通信解析ツール
    - MPInside
    - Perfcatcher

PerfSuite

- PerfSuiteは、プログラムのホットスポットをルーチンレベル、ラインレベルで調査することができます。

- PerfSuiteの特徴
  - 再リンクを必要としない
    - （ラインレベルの解析は”-g”を付けて再ビルドの必要があります。）
  - MPIやOpenMPによる並列プログラムに対応
  - シンプルなコマンドライン・ツール
  - スレッド/プロセスごとにレポートを出力
  - ソースラインレベルで解析可能

PerfSuite 利用方法 (準備)

準備

- moduleコマンドでPerfsuiteを利用できるように設定します。

```
$ module load perfsuite
```

PerfSuite 利用方法 (実行コマンド)

- psrunコマンドを用いてプロファイルの取得をします。ラインレベルでの取得が必要な場合は"-g"オプションを付けてビルドします。
- PerfSuiteでプロファイル取得時の実行コマンドです。dplaceコマンドのオプションが変わりますのでご注意ください。

- シリアルプログラム (0番のコアで実行)

```
$ dplace -s1 -c0 psrun ./a.out
```

- OpenMPプログラム(4スレッドを0から3番のコアで実行)

```
$ dplace -x5 -c0-3 psrun -p ./a.out
```

- MPIプログラム(SGI MPTを用いて、4プロセスを0から3番のコアで実行)

```
$ mpirun -np 4 dplace -s2 -c0-3 psrun -f ./a.out
```

PerfSuite 利用方法 (実行例)

- OpenMPプログラム4スレッドの実行例
  - 実行後、スレッド/プロセス毎に以下の名前のファイルが生成されます。

”プロセス名.(スレッド番号.)PID.ホスト名.xml”

```
$ ls -l a.out*.xml
-rw------- 1 sgise4 12183  5月 20 15:22 a.out.0.987430.uv.xml   スレッド0
-rw------- 1 sgise4  5901  5月 20 15:22 a.out.1.987430.uv.xml   管理スレッド
-rw------- 1 sgise4 14027  5月 20 15:22 a.out.2.987430.uv.xml   スレッド1
-rw------- 1 sgise4 14241  5月 20 15:22 a.out.3.987430.uv.xml   スレッド2
-rw------- 1 sgise4 11960  5月 20 15:22 a.out.4.987430.uv.xml   スレッド3
```

PerfSuite 利用方法 (結果の表示例)

- プロファイル結果として出力されたファイルをpsprocessコマンドで成形してプロファイル結果を表示します。(ここではスレッド0のプロファイル結果を表示します)

```
$ psprocess a.out.0.987430.uv.xml
```

PerfSuite 利用方法（結果の表示例）

- OpenMPプログラムを4スレッドで実行したときのマスタースレッドの結果。

```
PerfSuite Hardware Performance Summary Report

Version                 : 1.0
Created                 : Wed May 20 15:32:48 JST 2015
Generator               : psprocess 0.5
XML Source              : a.out.0.987430.uv.xml

Execution Information
================================================================================
Collector               : libpshwpc
Date                    : Wed May 20 15:22:17 2015
Host                    : uv
Process ID              : 987430
Thread                  : 0
User                    : sgise4
Command                 : a.out

Processor and System Information
================================================================================
Node CPUs               : 1280
Vendor                  : Intel
Brand                   : Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz
CPUID                   : family: 6, model: 45, stepping: 7
CPU Revision            : 7
Clock (MHz)             : 2400.117
Memory (MB)             : 20035039.45
Pagesize (KB)           : 4

Cache Information
================================================================================
Cache levels            : 3
--------------------------------
...途中省略...

Profile Information
================================================================================
Class                   : itimer
Version                 : 1.0
Event                   : ITIMER_PROF (Process time in user and system mode)
Period                  : 40000
Samples                 : 583
Domain                  : all
Run Time                : 41.42 (seconds)
Min Self %              : (all)
```

モジュール毎のプロファイル結果

```
Module Summary
-------------------------------------------------------------------------------
Samples  Self %  Total %  Module

  580  99.49%  99.49%  /export/home/sgise4/gojuki/test_sample/himeno/a.out
    3   0.51%  100.00%  /opt/sgi/perfsuite/lib/libpshwpc_r.so.1.0.1
```

ファイル毎のプロファイル結果

```
File Summary
-------------------------------------------------------------------------------
Samples  Self %  Total %  File

  490  84.05%  84.05%  /export/home/sgise4/gojuki/test_sample/himeno/himenoBMTxp_omp.f90
   93  15.95%  100.00%  ??
```

関数毎のプロファイル結果

```
Function Summary
-------------------------------------------------------------------------------
Samples  Self %  Total %  Function

  484  83.02%  83.02%  L_jacobi__290__par_region0_2_128
   78  13.38%  96.40%  __intel_ssse3_rep_memcpy
   12   2.06%  98.46%  __intel_memset
    6   1.03%  99.49%  initmt
    3   0.51%  100.00%  xml_write_profileinfo
```

ラインレベルでのプロファイル結果

```
Function:File:Line Summary
-------------------------------------------------------------------------------
Samples  Self %  Total %  Function:File:Line

   80  13.72%  13.72%  L_jacobi__290__par_region0_2_128:/export/home/sgise4/gojuki/test_sample/himeno/himenoBMTxp_omp.f90:314
   78  13.38%  27.10%  __intel_ssse3_rep_memcpy:??:?
   59  10.12%  37.22%  L_jacobi__290__par_region0_2_128:/export/home/sgise4/gojuki/test_sample/himeno/himenoBMTxp_omp.f90:303
   53   9.09%  46.31%  L_jacobi__290__par_region0_2_128:/export/home/sgise4/gojuki/test_sample/himeno/himenoBMTxp_omp.f90:307
   50   8.58%  54.89%  L_jacobi__290__par_region0_2_128:/export/home/sgise4/gojuki/test_sample/himeno/himenoBMTxp_omp.f90:313
   36   6.17%  61.06%  L_jacobi__290__par_region0_2_128:/export/home/sgise4/gojuki/test_sample/himeno/himenoBMTxp_omp.f90:312
```

MPInside

- MPInsideはMPIプログラムにおいて、どのMPI関数で時間がかかっているのか、また通信するデータサイズなどのプロファイルを取得することができます。

- プロファイル結果によって、MPIプログラムのチューニングに有用な情報が得られます。

MPInside 利用方法(準備と実行)

- 準備
  - moduleコマンドでMPInsideを利用できるように設定します。

```
$ module load MPInside/3.6.5
```

- 実行例
  - 4プロセスを0から3番のコアで実行する場合を示します。

```
$ mpirun -np 4 dplace -s1 -c0-3 MPInside ./a.out
```

  - 実行結果はmpinside_statsファイルに保存されます。

MPInside 利用方法(実行結果)

- 4並列で実行したときの実行結果

```
MPInside 3.6.5 standard(Oct 16 2014 05:34:39) Input variables:

 >>> column meanings <<<<
MPI_Init:      MPI_Init
Waitall:       MPI_Waitall:    Bytes sent=0,Calls sending data+=count;Bytes received=0,Calls receiving data++
Isend:  MPI_Isend
Irecv:  MPI_Irecv
Barrier:       MPI_Barrier:    Calls sending data+=comm_sz;Calls receiving data++
Bcast:  MPI_Bcast:     Calls sending data+=comm_sz,Calls receiving data++;Root:Bytes sent++:Bytes
received+=count
Allreduce:     MPI_Allreduce:  Calls sending data+=comm_sz;Bytes received+=count,Calls receiving data++
MPI_Cart_create:       MPI_Cart_create
MPI_Cart_get:   MPI_Cart_get
MPI_Cart_shift: MPI_Cart_shift
mpinside_overhead:     mpinside_overhead:     Various MPInside overheads
```

```
>>>> Communication time totals (s) 0 1<<<<
CPU    Compute MPI_Init      Waitall Isend  Irecv  Barrier Bcast  Allreduce     MPI_Cart_create MPI_Cart_get
MPI_Cart_shift  mpinside_overhead
---    ------ General Point-to-point  Point-to-point  Point-to-point  Collective     Collective     Collective     General
General General None
0000   57.7816 0.0001 1.0282 1.4311 0.0111 0.0005 0.0001 0.0464 0.0001 0.0000 0.0000 0.0131
0001   57.7506 0.0001 1.0475 1.4389 0.0231 0.0024 0.0009 0.0353 0.0001 0.0000 0.0000 0.0105
0002   57.8491 0.0001 0.2999 1.9494 0.0141 0.0019 0.0008 0.1835 0.0001 0.0000 0.0000 0.0089
0003   57.7414 0.0001 0.4212 1.9357 0.0217 0.0021 0.0010 0.1759 0.0001 0.0000 0.0000 0.0144

>>>> Mbytes sent <<<<
CPU    Compute MPI_Init      Waitall Isend  Irecv  Barrier Bcast  Allreduce     MPI_Cart_create MPI_Cart_get
MPI_Cart_shift  mpinside_overhead
0000   ------      0    0  277    0    0    0    0    0    0    0    0
0001   ------      0    0  277    0    0    0    0    0    0    0    0
0002   ------      0    0  277    0    0    0    0    0    0    0    0
0003   ------      0    0  277    0    0    0    0    0    0    0    0

>>>> Calls sending data <<<<
CPU    Compute MPI_Init      Waitall Isend  Irecv  Barrier Bcast  Allreduce     MPI_Cart_create MPI_Cart_get
MPI_Cart_shift  mpinside_overhead
0000   ------      1  5896  1474    0    8    8  2956    0    0    0    0
0001   ------      1  5896  1474    0    8    8  2956    0    0    0    0
0002   ------      1  5896  1474    0    8    8  2956    0    0    0    0
0003   ------      1  5896  1474    0    8    8  2956    0    0    0    0

>>>> Mbytes received <<<<
CPU    Compute MPI_Init      Waitall Isend  Irecv  Barrier Bcast  Allreduce     MPI_Cart_create MPI_Cart_get
MPI_Cart_shift  mpinside_overhead
0000   ------      0    0    0  555    0    0    0    0    0    0    0
0001   ------      0    0    0  555    0    0    0    0    0    0    0
0002   ------      0    0    0  555    0    0    0    0    0    0    0
0003   ------      0    0    0  555    0    0    0    0    0    0    0

>>>> Calls receiving data <<<<
CPU    Compute MPI_Init      Waitall Isend  Irecv  Barrier Bcast  Allreduce     MPI_Cart_create MPI_Cart_get
MPI_Cart_shift  mpinside_overhead
0000   ------      0  1474    0  2948    2    2  739    1    1    2    0
0001   ------      0  1474    0  2948    2    2  739    1    1    2    0
0002   ------      0  1474    0  2948    2    2  739    1    1    2    0
0003   ------      0  1474    0  2948    2    2  739    1    1    2    0
```

Perfcatcher

- PerfcatcherはMPIプログラムやSHMEMプログラムの通信および同期のプロファイルを取得します。

- プロファイル結果によって、MPIプログラムのチューニングに有用な情報が得られます。

Perfcatcher 利用方法(準備と実行)

- 準備
  - moduleコマンドでPerfcatcherを利用できるように設定します。

```
$ module load perfcatcher
```

- 実行例
  - 4プロセスを0から3番のコアで実行する場合を示します。

```
$ mpirun -np 4 dplace -s3 -c0-3 perfcatch ./a.out
```

  - 実行結果はMPI_PROFILEING_STATSファイルに保存されます。

Perfcatcher 利用方法（実行結果）

```
============================================================
PERFCATCHER version 25
(C) Copyright SGI.  This library may only be used
on SGI hardware platforms. See LICENSE file for
details.
============================================================
MPI/SHMEM program profiling information
Job profile recorded:        Fri May 02 15:13:53 2014
Program command line:        ./a.out
Total MPI/SHMEM processes:   4

Total MPI/SHMEM job time, avg per rank              65.7406 sec
Profiled job time, avg per rank                     65.7406 sec
Percent job time profiled, avg per rank             100%

Total user time, avg per rank                       65.4 sec
Percent user time, avg per rank                     99.4819%
Total system time, avg per rank                     0.15 sec
Percent system time, avg per rank                   0.228169%

Time in all profiled MPI/SHMEM functions, avg per rank     0.198903 sec
Percent time in profiled MPI/SHMEM functions, avg per rank  0.302558%


Rank-by-Rank Summary Statistics
-------------------------------

Rank-by-Rank: Percent in Profiled MPI/SHMEM functions
      Rank:Percent
      0:0.273715%    1:0.255355%    2:0.292867%    3:0.388295%

 Least:  Rank 1     0.255355%
 Most:   Rank 3     0.388295%
 Load Imbalance:  0.066686%

Rank-by-Rank: User Time
      Rank:Percent
      0:99.4058%     1:99.4971%     2:99.5123%     3:99.5123%

 Least:  Rank 0     99.4058%
 Most:   Rank 3     99.5123%

Rank-by-Rank: System Time
      Rank:Percent
      0:0.289015%    1:0.212958%    2:0.197747%    3:0.212958%

 Least:  Rank 2     0.197747%
 Most:   Rank 0     0.289015%
```

```
Notes
-----
Wtime resolution is                 1e-06 sec

Rank-by-Rank MPI Profiling Results
----------------------------------

Activity on process rank 0

 MPI activity
 comm_rank          calls:     1  time:        0 s  0 s/call
 irecv              calls:   168  time:  0.0616438 s  datacnt 16601088
        Average data size 98816  (min 0, max 263168)   size:count(peer)
        263168:  42x(1)  132096:  42x(2)     0:  84x(-1)

        unique peers:    1    2   -1

 isend              calls:   168  time:  0.0758688 s  0.0004516 s/call
        Average data size 197632  (min 132096, max 263168)   size:count(peer)
        263168:  42x(-1)  263168:  42x(1)  132096:  42x(-1)  132096:  42x(2)
        unique peers:   -1    1    2

 waitall            calls:    84  time:  0.0198107 s  # of reqs 336  avg datacnt 49409
 barrier            calls:     2  time: 0.000370979 s  0.00018549 s/call
 allreduce          calls:    44  time:  0.0221236 s  0.000502809 s/call
        Average data size 4.18182  (min 4, max 8)   size:count(comm)
          8:   2x(1)     4:  42x(1)

        unique comms:    1

 bcast              calls:     2  time: 0.000123978 s  6.19888e-05 s/call
        Average data size 11  (min 10, max 12)   size:count(comm)
          12:   1x(1)    10:   1x(1)

        unique comms:    1



Activity on process rank 1

 MPI activity
 comm_rank          calls:     1  time:        0 s  0 s/call
 irecv              calls:   168  time:  0.0637088 s  datacnt 16601088
        Average data size 98816  (min 0, max 263168)   size:count(peer)
        263168:  42x(0)  132096:  42x(3)     0:  84x(-1)

        unique peers:    0    3   -1
```

11. 並列化プログラミング

並列化について

- プログラムを並列化することのメリットは、実行時間　（ターンアラウンドタイム）が短縮されることです。

- 「並列化によるスピードアップ s」とは、下式のように、スレッド数 1 で実行した場合の実行時間 T_1 と、スレッド数 N で実行した場合の実行時間 T_N の比であると定義します。

$$s = \frac{T_1}{T_N}$$

アムダールの法則 (1)

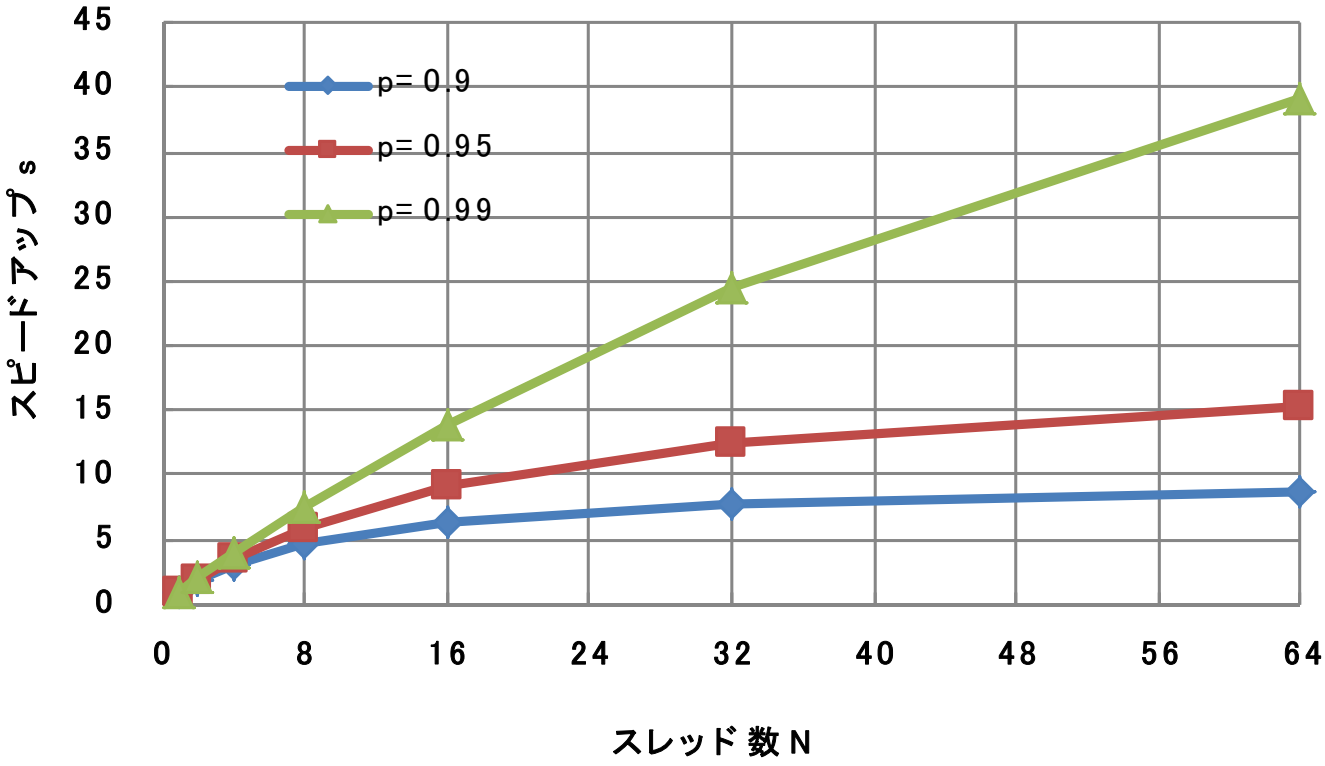
- あるプログラムを逐次実行した際の実行時間のうち、並列化できる部分の割合を p (0 ≦ p ≦ 1) とします。このとき、スレッド数 N で実行した場合のスピードアップ s は、並列化のオーバーヘッド等を無視できるとすると、以下の式に従うことが知られています。

$$s = \frac{1}{\frac{p}{N} + (1 - p)} \qquad (0 \le p \le 1)$$

- これを、アムダール の法則といいます。

アムダールの法則 (2)

- アムダール の法則によるスピードアップの理論値



アムダールの法則 (3)

- 多くのプロセッサを使用して高い並列性能を得るためには、実行時間中の並列処理されている時間の割合 p を少しでも高めることが重要です。
- 並列化のオーバーヘッドが増大することは、p が減少すること等価であると考えられます。
- したがって、並列性能を高めるためには、
  - 逐次実行領域を減らす
  - オーバーヘッドを減らす

  ことが重要です。

アムダールの法則 (4)

- 逐次実行領域を減らす
  - 並列実行領域を増やす。
  - OpenMPでは、master, critical, atomic, single 領域を減らす。
- オーバーヘッドを減らす
  - 小さい並列実行領域をたくさん定義するのではなく、
    大きな並列実行領域を定義するようにする (粗粒度)。
  - 十分な仕事量があるものだけ並列処理する。
  - 同期・待ち時間を避ける。
    - OpenMPでは、barrier を減らす、可能ならば nowait を指定する、...
    - ロードバランスを改善する。

12. インテルコンパイラ自動並列化

自動並列化

- インテルコンパイラによる自動並列化
  - マルチスレッドの並列化
  - コンパイラによる最適化と組み合わせた並列化
  - コンパイルオプションによる簡単な操作
  - 並列化診断メッセージによるレポート（ソースコードは出力されません）

インテルコンパイラによる自動並列化(1)

- インテルコンパイラで自動並列化を有効にするには-parallelオプションを指定します。

コンパイルとリンクを別々に行う場合

```
$ ifort -c -parallel myprog.f
$ ifort -parallel (または-openmp) myprog.o
```

コンパイルとリンクを一緒に行う場合

```
$ ifort -parallel myprog.f
```

- 実行時には、OpenMPによる並列化と同様に、次の環境変数でスレッド数やランタイム・スケジュールを設定します。

- 環境変数

OMP_NUM_THREADS	使用するスレッド数を指定します。 デフォルトは実行バイナリを作成したシステムの搭載されているコア数。
OMP_SCHEDULE	ランタイム・スケジューリングを指定します。 デフォルトはSTATIC。

インテルコンパイラによる自動並列化(2)

- 自動並列化では、2つの指示行を使うことができます。

Fortranの場合 !DEC$ PARALLEL Cの場合 #pragma parallel	ループに対して、想定される依存性を無視して自動並列化を行うことをコンパイラに指示します。ただし、依存性が証明されると並列化されません。
Fortranの場合 !DEC$ NOPARALLEL Cの場合 #pragma noparallel	ループに対して、自動並列化を無効にします。

例

```
!DEC$ NOPARALLEL
do I = 1, n
  x(i) = I
end do

!DEC$ PARALLEL
do I = 1, n
  a( x(i) ) = I
end do
```

自動並列化されません。

依存関係が想定されますが、
自動並列化されます。

インテルコンパイラによる自動並列化(3)

-parallel	自動並列化機能を有効にし、安全に並列化できるループのマルチスレッド・コード生成をコンパイラに指示します。このオプションは-O2または-O3オプションも指定する必要があります。
-par-threshold*n*	並列実行が効果的である可能性に基づいてループの自動並列化の閾値を設定します。 *n*=0: ループの計算量に関わらず、常に自動並列化します。 *n*=100: 性能向上が見込める場合のみ自動並列化します。 *n*=1～99は速度向上する可能性を表します。
-par-report*n*	自動並列化の診断情報を制御します。デフォルトでは自動並列化メッセージは出力されません。 *n*=0: 診断情報を出力しません。 *n*=1: 正常に自動並列化できたループに対して"LOOP AUTO-PARALLELIZED"のメッセージを出力します。 *n*=2: 正常に自動並列化したループとできなかったループに対してメッセージを出力します。 *n*=3: 2の出力に加えて自動並列化できなかった場合の判明した依存関係と想定される依存関係を出力します。

13. OpenMPプログラミング入門

- OpenMPの利用方法
- OpenMPとは
- ループの並列化
- OpenMP指示行と環境変数

OpenMPの利用方法 (1)

- インテル®コンパイラでは、OpenMP Fortran 3.0のAPIをサポートしています。インテル®コンパイラでOpenMPを使用するときは次の様に-openmpオプションを指定してコンパイルします。

コンパイルとリンクを別々に行う場合

```
$ ifort -c -openmp myprog.f
$ ifort -openmp myprog.o
```

コンパイルとリンクを一緒に行う場合

```
$ ifort -openmp myprog.f
```

- 実行するときはOpenMP環境変数OMP_NUM_THREADSで使用するスレッド数を指定します。

OpenMPの利用方法 (2)

-openmp	OpenMP指示行に基づきマルチスレッド・コードを生成します。
-openmp-reportn	OpenMPの診断情報を制御します。 デフォルトではOpenMPの診断メッセージは出力されません。 n=0: 診断メッセージを表示しません。 n=1: 正常に並列化された、領域、およびセクションを示す診断メッセージを表示します。 n=2: 1で表示されるメッセージに加えて、正常に処理されたMASTER、SINGLE、CRITICAL、ORDERED、ATOMICなどの診断メッセージを表示します。

OpenMPとは

- OpenMP指示行による並列化

```
!$OMP PARALLEL DO SHARED(A, B, C)
      do i = 1, 10000
        A(i) = B(i) + C(i-1) + C(i+1)
      end do
```

代表的なOpenMP指示行

・PARALLEL { ……}
・PARALLEL DO, PARALLEL DO REDUCTION(+: …)
・MASTER
・CRITICAL
・BARRIER

OpenMPの指示行

- OpenMP 指示行 = コンパイラに対する並列化命令
- OpenMP 機能が無効の場合には、単なる コメントとして扱われ無視されます。
- 大文字と小文字は区別されます。（Cの場合）
- 継続行は “&”アンパサンド(Cの場合は”/”バックスラッシュ)で記述します。 自由形式の場合は前の行の最後にも”&”が必要です。

```
!$OMP PARALLEL DO PRIVATE(変数p1, . . . ) SHARED(変数s1, . . . )
      do i = 1,N
            ...
      end do
```

並列実行領域

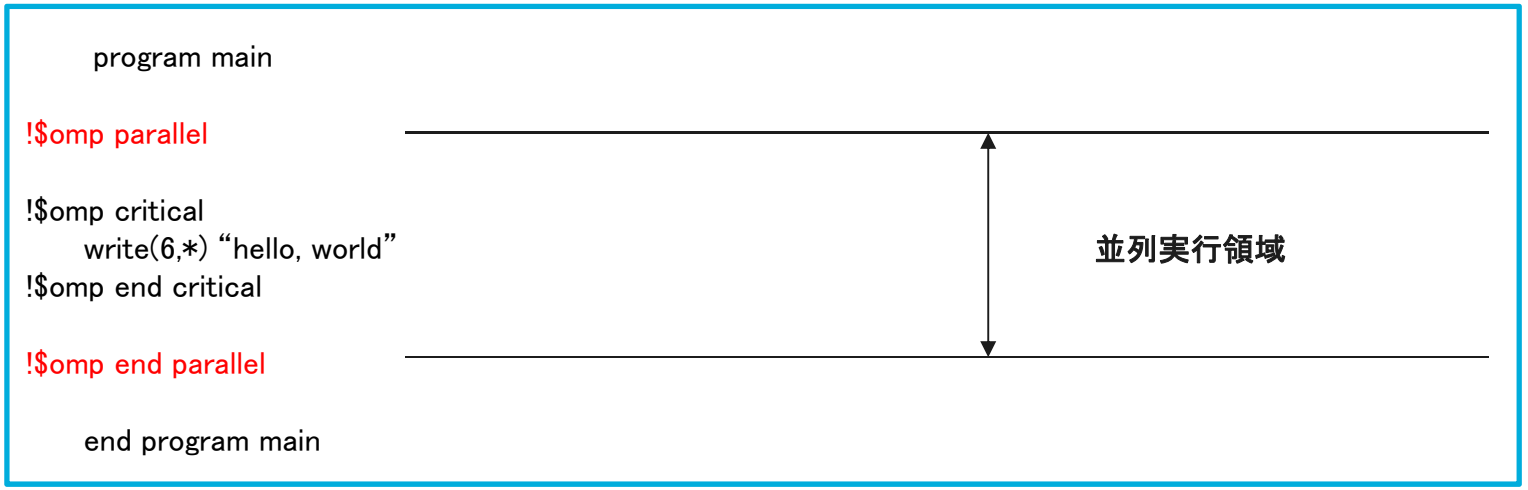
同じ意味

```
!$OMP PARALLEL DO PRIVATE(変数p1, . . . ) &
!$OMP&            SHARED (変数s1, . . . )
      do I = 1, N
            ...
      end do
```

並列実行領域

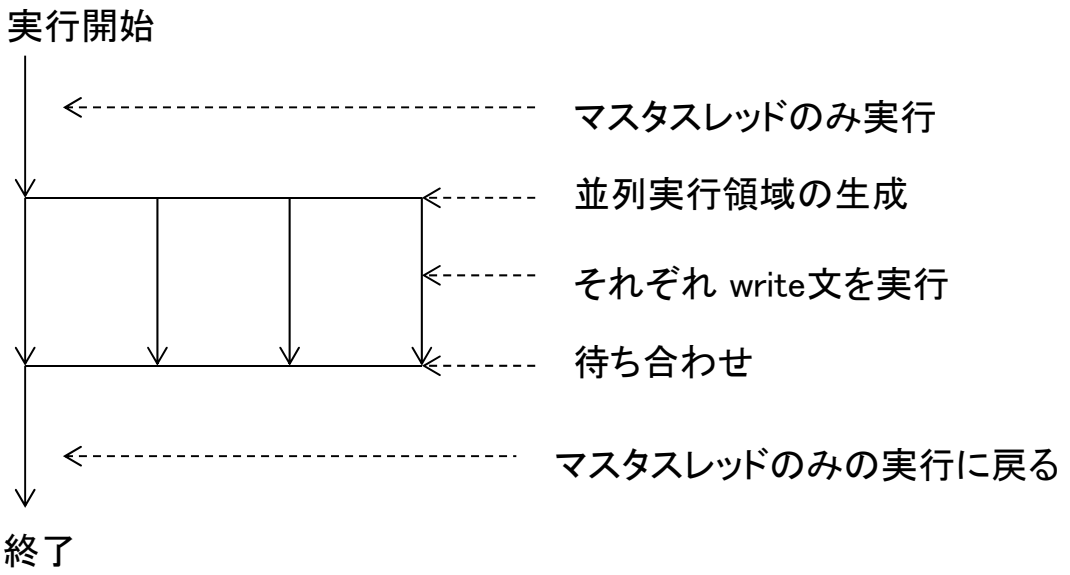
“hello, world”

- PARALLEL 指示行
  - !$OMP PARALLEL [オプション (節)]
    - 指示文に続く文を並列に実行します。



hello, world の実行例

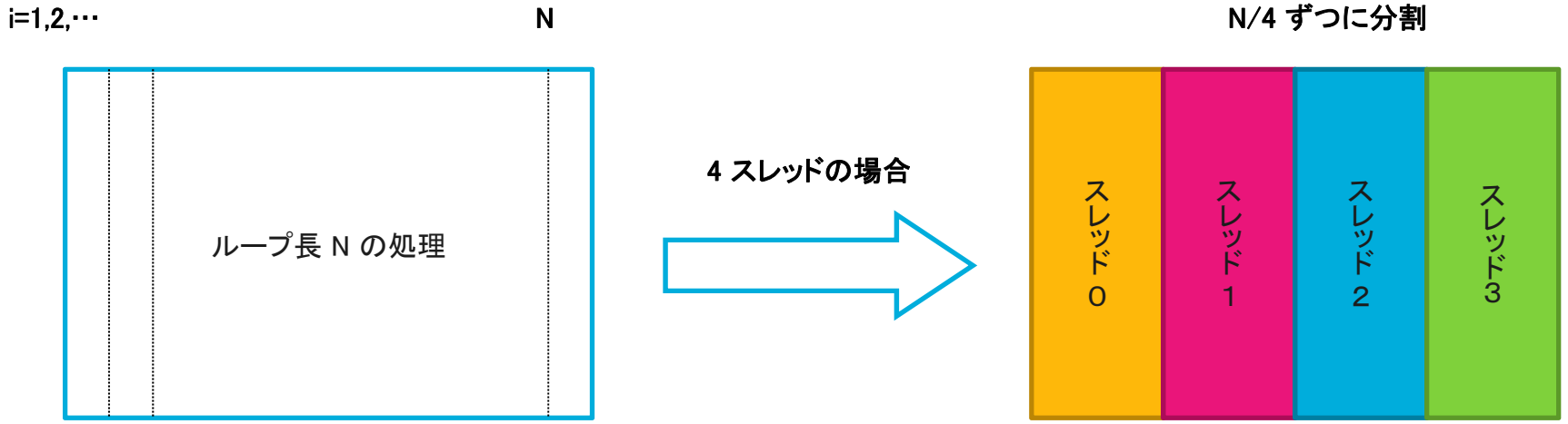
```
$ ifort -openmp -openmp-report1 hello.f
hello.f(3): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$
$ export OMP_NUM_THREADS=4
$ dplace -x2  ./a.out
hello, world
hello, world
hello, world
hello, world
$
```



doループのワークシェアリング

- DO指示行
  - !$OMP DO [オプション (節)]

  – 並列実行領域で使用し、後続する do ループを各スレッドで分担して実行します。
  – デフォルトでは、ループ長がスレッド数で均等に分割されます。



do ループのワークシェアリング

- PARALLEL DO 指示行
  - PARALLEL 指示行 + DO 指示行
  - 並列実行領域を作成し、後続の do ループを分割実行します。

```
      subroutine daxpy( n, c, x, y)

      integer :: n, i
      real(kind=8) :: c
      real(kind=8),dimension(n) :: x, y
!$omp parallel do private(i) shared(n, c, x, y)
      do I = 1, n
        y(i) = y(i) + c * x(i)
      end do

      return
      end subroutine daxpy
```

データスコープ属性

- 並列実行領域や分割実行されるループ中で参照される変数に関して、それらが、
  - 各スレッドごとに独立した変数とすべきか、
  - すべてのスレッドで共有される変数とすべきか、

  を宣言する必要があります。
  これらを「データスコープ属性」と言います。

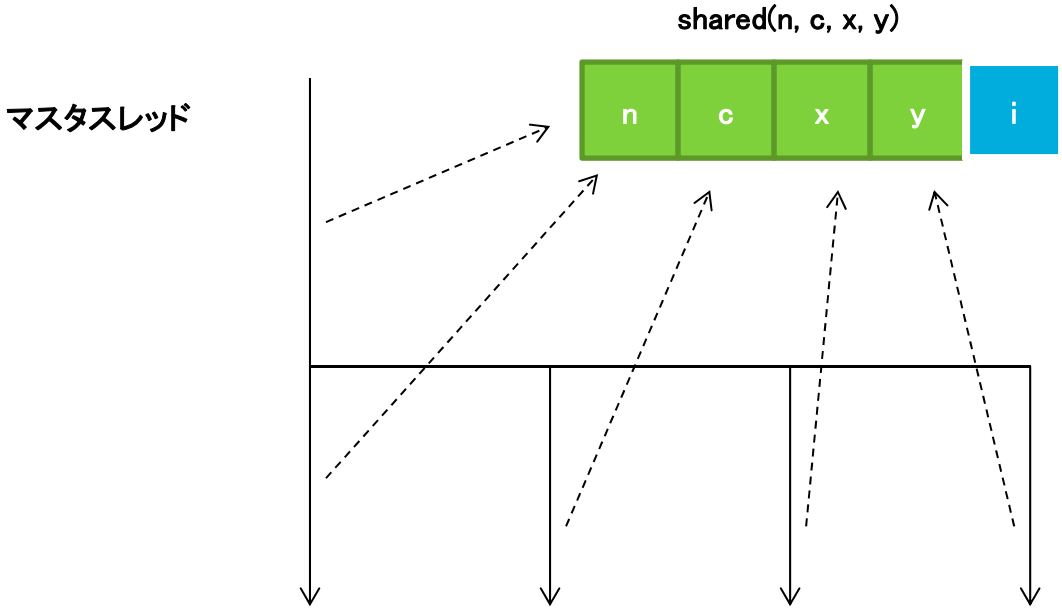
- データスコープ属性は、PARALLEL指示文や DO指示文の「オプション」として指定します。これらの「オプション」を、OpenMP では「節 (clause)」と呼びます。

```
!$omp parallel do private(i) shared(n, c, x, y)
```

private節　　shared節

shared 変数と private 変数

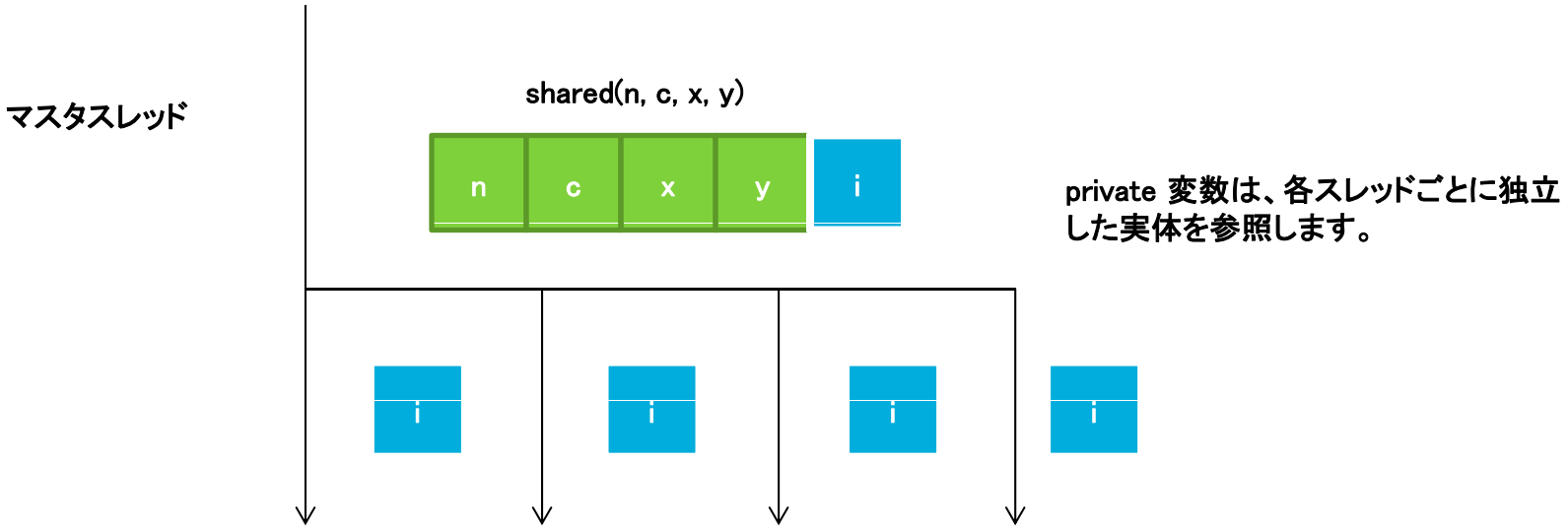
- shared 変数
  - shared 節に指定された変数に対しては、すべてのスレッドから同一のオブジェクトが参照されます。
  - オブジェクトの内容は、マスタスレッドが保持していたものと同一です。



shared 変数は、すべてのスレッドが同一の実体を参照します。

shared 変数と private 変数

- private 変数
  - private 節に指定された変数は、それぞれのスレッドに独立なオブジェクトが生成されます。
  - private 変数の内容は、元のマスタスレッドの変数の内容とは無関係です。



暗黙のデータ共有属性

- 暗黙のデータ共有属性
  - 並列実行領域の開始前に定義され、並列実行領域の開始時点で可視な変数は shared
  - ループのインデックス変数は private
  - 並列実行領域内で定義された変数は private
- デフォルトの変更
  - default(shared)
    データ共有属性が指定されない変数は shared とします。(デフォルト)
  - default(private)
    データ共有属性が指定されない変数は private とします。
  - default(none)
    すべての変数に対してデータ共有属性の明示的な指定を要求します。

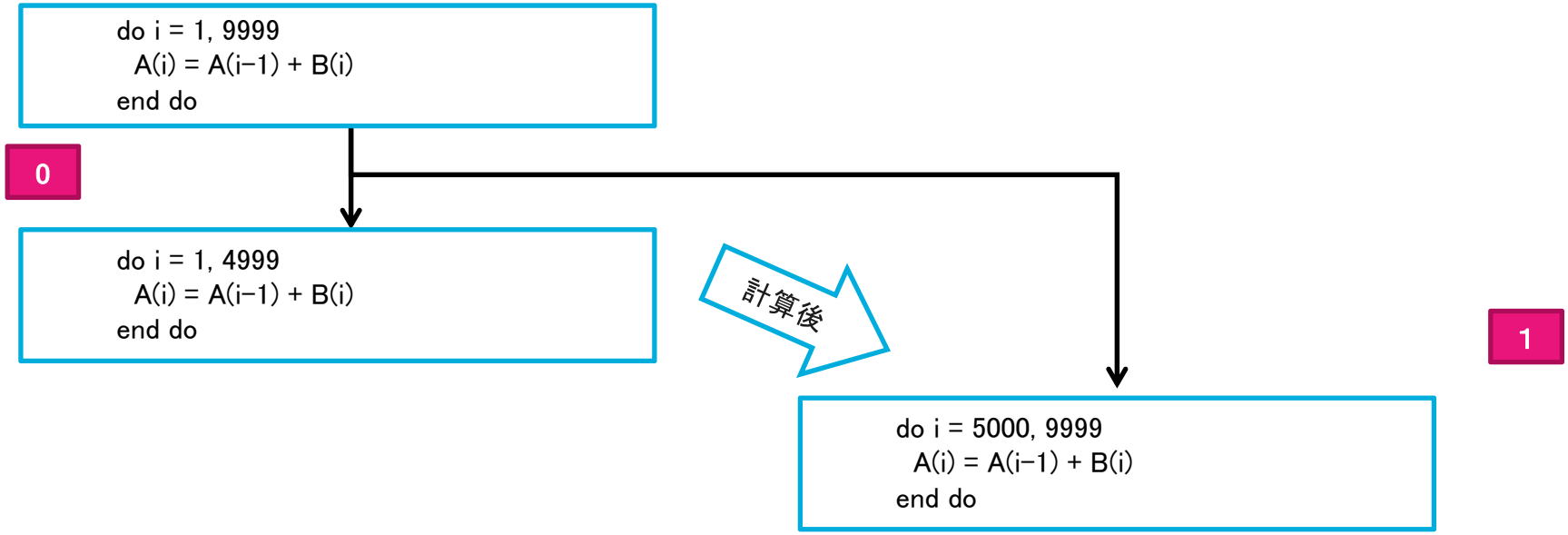
並列化可能なループ

- 並列化可能なループ
  - doループである
    - do whileなどのループは難しい（OpenMP3.0では対応）
  - ループ内に依存性がない
    → 次ページ以降参照
  - ループの途中でループを終了する命令がない
    → ループの前か後で終了するように回避する…
  - writeやread等のI/O命令を含まない

→ 手動による指示文挿入ならば可能

後方依存性のあるループ

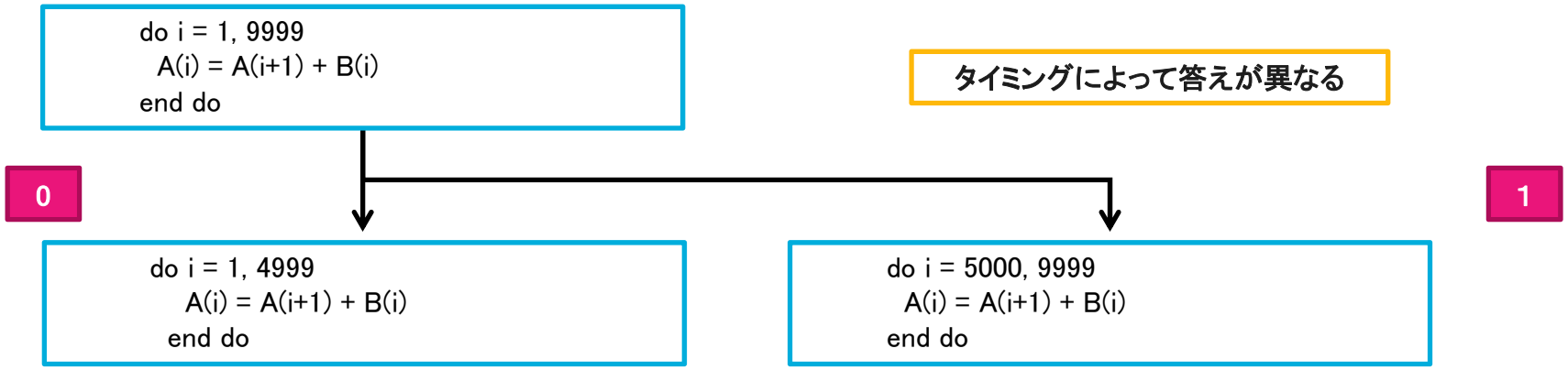
- 並列化できないループ～後方依存性のあるループ



(理由)スレッド1で i=5000の計算を行う時、A(4999)のデータを必要とするが、A(5000)はスレッド0によって計算済みでなければならないが、その保証をしようとすると逐次演算と同じになります。

前方依存性のあるループ

- 並列化できないループ～前方依存性のあるループ



(理由)スレッド0で i=4999の計算を行う時、A(5000)のデータを必要とし、A(5000)はスレッド1によって計算済みであってはならない。しかし、スレッド0と1が同時にこのdoループを開始することは保証されていないため、タイミングによって結果がおかしくなる可能性があります。(ただし、ループ分割などの方法により並列化は可能)

依存性のあるループ

- 並列化できないループ～前方・後方依存性のあるループ

  iとi-1, i+1が同じ行に書かれていなくても、以下のように同じループ内にあれば依存性が生じます。

```
do i = 1, imax-1
  A(I)   = A(I)   + ……
  A(I-1) = A(I-1) + ……
end do
```

```
do i = 1, imax-1
  A(I)   = A(I)   + ……
  A(I+1) = A(I+1) + ……
end do
```

間接参照のあるループ

- 並列化できないループ～間接参照のあるループ

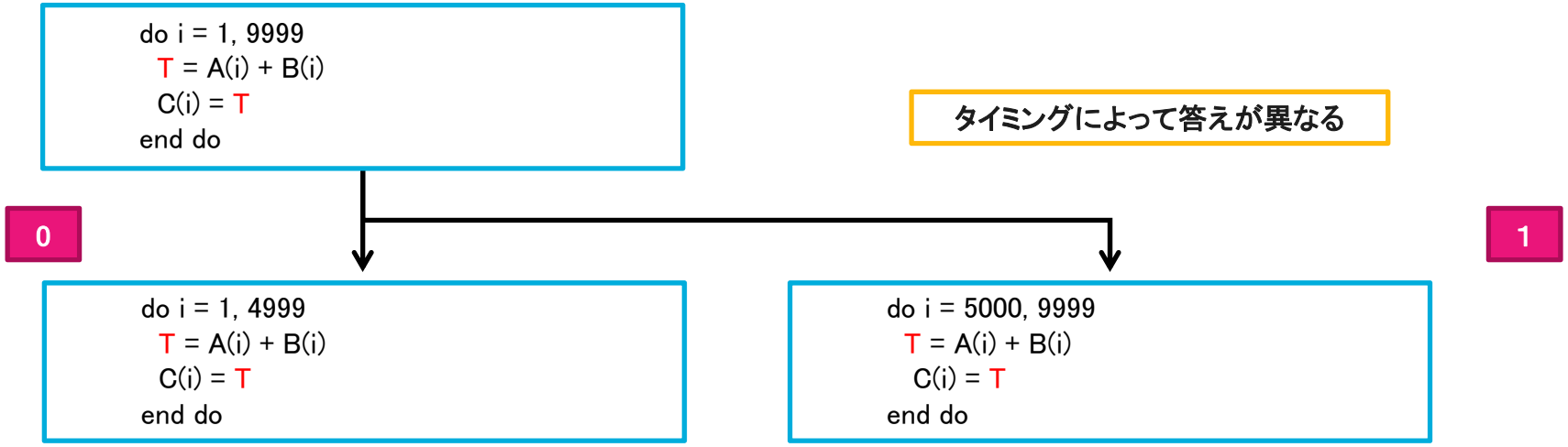
  iとi-1, i+1が同じ行に書かれていなくても、以下のように同じループ内にあれば依存性が生じます。

```
do i = 1, imax-1
  Index(i) = ……
end do

do i = 1, imax-1
  A( Index(i) )  = B(i) + C(i)
end do
```

  コンパイラには、配列Index( )の値がどうなっているかは分かりません。例えば、Index(1)とIndex(800)の値が同じ1だとすると、スレッド0と1は、同じ出力先に値を書き込むことになります。もし、ユーザがIndex( ) の値がすべて異なっていることが分かっているならば、自らの指示(責任)により並列化可能です。

一時変数を含むループ

- そのまま並列化するとまずいループ～一次変数を含む



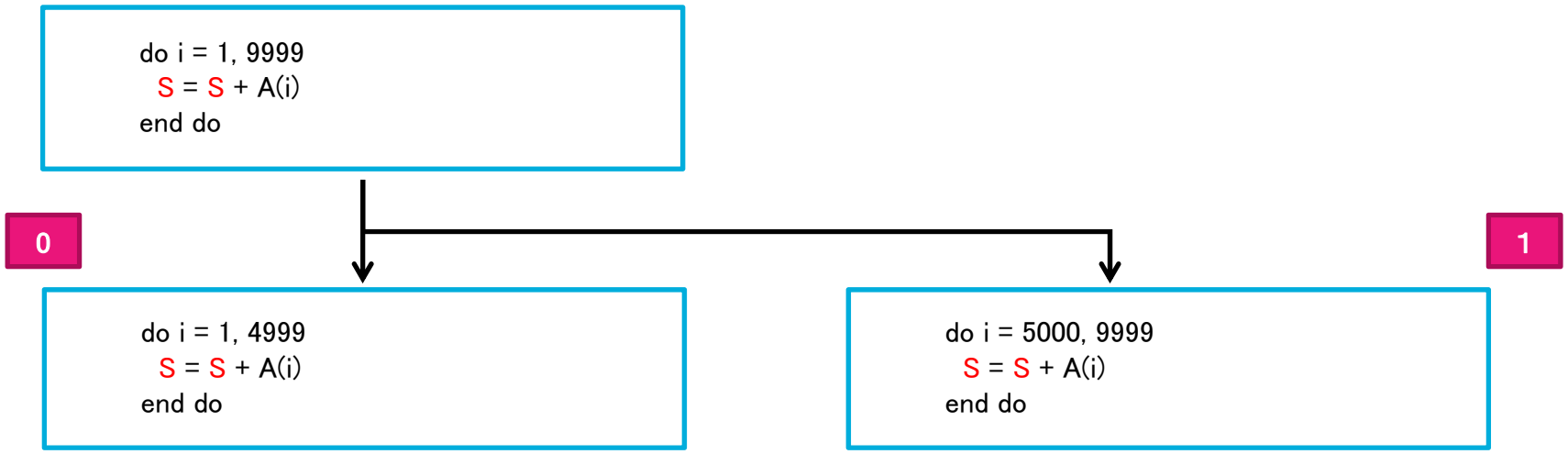
∵一次変数Tが、スレッド0と1の両方から同時にアクセスされてしまうと、タイミングによって答えが違ってくる

Tが各スレッドにローカルな変数ならば、並列化可能になります。具体的には変数Tを以下のようにprivate変数にします。

```
!$OMP PARALLEL DO PRIVATE(T)
```

縮約演算(reduction演算)

- そのまま並列化するとまずいループ～一次変数を含む

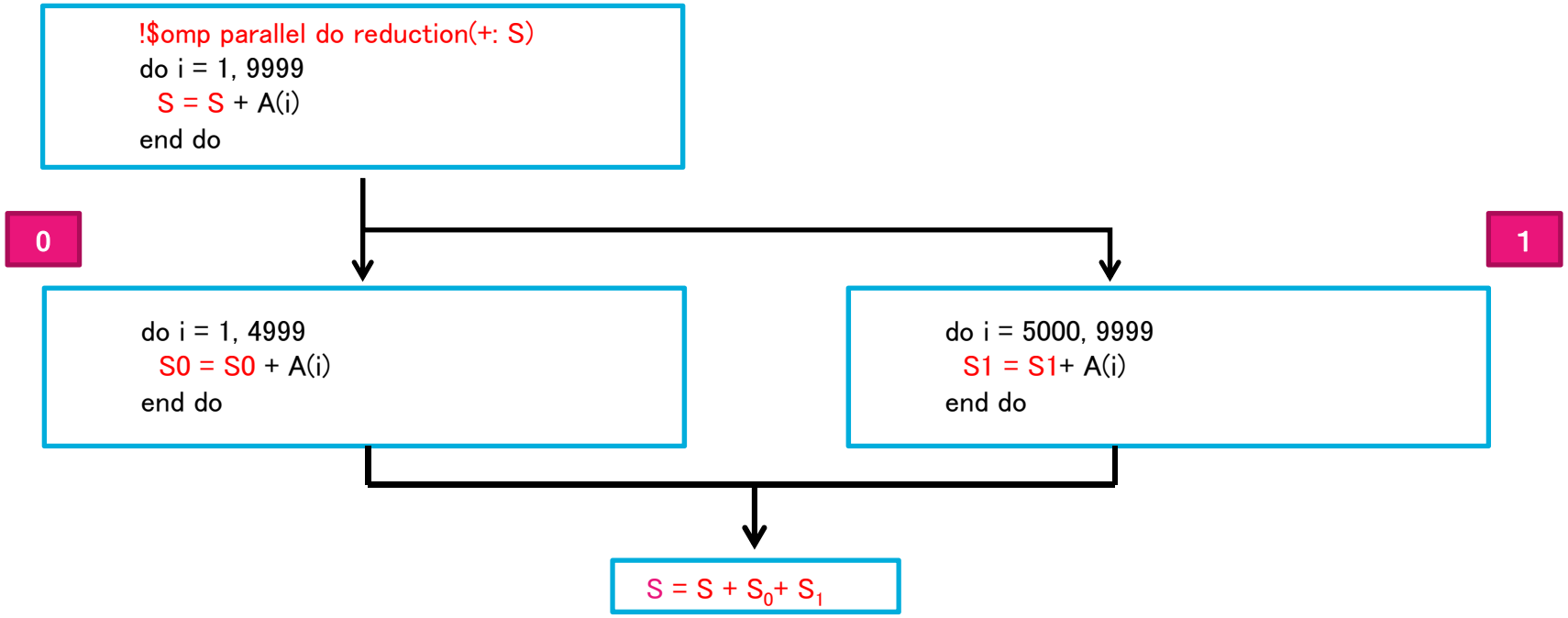


∵変数Sが、グローバルな属性ならば、スレッド0と1が次々と勝手にSの値を書き換えるため、不正な結果となる

Sを各スレッドにローカルな変数にすると部分和は求めることができるが、全体の和は？

縮約演算(reduction演算)

- そのまま並列化するとまずいループ～reduction演算



(注)reduction演算の結果は、逐次演算の結果と異なる場合があります。これは、演算の順序が異なり丸め誤差が生じる可能性があるためです。並列度数を変更しても結果が異なる場合があります。

縮約演算(reduction演算)

- reduction節
  - 配列を何らかの演算によってひとつのスカラー変数に縮約する操作を「reduction演算」、その変数を「reduction変数」と言います。
  - reduction節は次のような書式です。

    !$OMP DO REDUCTION (*op* : *var*)

    var はreduction変数 (のカンマ区切りリスト)
    op の演算子は、+, *, −, .AND. , .OR. , .EQV. , .NEQV. ,または、
    組み込み関数 MAX, MIN, IAND, IOR, IEORのいずれか。
  - reduction変数*var* は、ループ実行中に private 変数として扱われ、終了後に各スレッドの値を元の変数に縮約します。 *var* は、実行するreduction演算の種類に応じて次のように適切に初期化されます。
    - *op* = +、− の時 : 初期値 0
    - *op* = * の時 : 初期値 1
    - *op* = MAX の時: 初期値は与えられたデータ型で負の絶対値最大の値
    - *op* = MIN の時: 初期値は与えられたデータ型で正の絶対値最大の値

PARALLEL 指示行と DO 指示行の制限

- PARALLEL 指示行の制限
  - PARALLEL 指示行によって生成された並列実行領域から GOTO等で抜け出してはいけません。また、並列実行領域外から並列実行領域に入るような分岐を行ってはなりません。
  - 並列実行領域内で、同期（後述）を行わずに同一のファイル等に対して I/O 処理を行った場合の動作は未定義です。
- DO 指示行の制限
  - DO 指示行で分割されたループを GOTO や EXIT等で終了してはいけません。
  - DO 指示行で分割されるループのループ変数は、整数型でなければなりません。
  - OpenMP3.0では、C/C++の場合、符号付または符号なしの整数型変数、C++の場合、ランダムアクセスイテレータ型の変数、Cの場合はポインタ型の変数がサポートされます。

暗黙の同期とnowait

do指示文の終了時には、
暗黙の同期が行われます。

```
!$omp parallel
!$omp do
      do I = 1, n-1
        b(i) = a(i) + a(i-1)
      end do
!$omp end do
!$omp do
      do I = 1, n-1
        b(i) = a(i) + a(i-1)
      end do
!$omp end do
!$omp end parallel
```

各ループの終了時に、すべてのスレッドが終了するまで待ち合わせます。待ち合わせのためのオーバーヘッドがかかります。

nowait を指定すると、他のスレッドの
終了を待たずに次の処理に移ります。

```
!$omp parallel
!$omp do
      do I = 1, n-1
        b(i) = a(i) + a(i-1)
      end do
!$omp end do nowait
!$omp do
      do I = 1, n-1
        b(i) = a(i) + a(i-1)
      end do
!$omp end do nowait
!$omp end parallel
```

ここで同期

ループの終了時に待ち合わせず直ちに次の処理に移ります。これにより、待ち合わせのオーバーヘッドを減らすことができます。ただし、2つのループ間に依存性があってはいけません。

バリア同期

- BARRIER 指示行
  - すべてのスレッドの実行がプログラム上の同じ BARRIER 指示行に到達するまで、待ち合わせを行います。

```
!$OMP MASTER
      open(……)
      read(……)                    ! マスタスレッドがファイルをリード
      close(……)
!$OMP END MASTER

!$OMP BARRIER              ! 読込が完了するまで待つ

      ……
```

その他の同期のための指示行

- MASTER 指示行
  - マスタスレッドのみが実行する処理を指定します。
- CRITICAL 指示行
  - 同時にひとつのスレッドのみで実行される領域を定義します。共有されている領域への書き込みや、I/O を行う際の排他制御などに用います。
- ATOMIC 指示行
  - CRITICAL 指示文と同様に排他制御を行いますが、ハードウェアによる最適化を行うことができる特定の演算（インクリメント等）のみに限定したものです。
- ORDERED 指示行
  - ループ中で、逐次実行した場合と同じ順序で実行される領域を定義します。

環境変数

- OpenMP プログラムの実行を制御する環境変数
  - OMP_NUM_THREADS
    - 実行に使用するスレッド数を指定します。
  - OMP_SCHEDULE
    - schedule(runtime) 節を指定したfor 指示文のループ分割方法を指定します。
    - schedule節には、以下のようなものがあります。
      - static : 全体をスレッド数で分割します。(デフォルト)
      - static, chunk : chunk を単位として分割します。
      - dynamic : 実行時に(OpenMP ランタイムが) 決定します.
  - OMP_STACKSIZE
    - 各スレッド毎のスタックサイズの上限を指定します。デフォルト値は4m(4MB)
    - OpenMP を使用しない場合に正常に動作するプログラムが、OpenMP を有効にした場合、起動直後にsegmentation faultで異常終了する場合には、このOMP_STACKSIZEの問題である可能性が考えられます。
    - 巨大な配列をローカル変数としてを確保しているような場合には、スタックサイズの問題が発生する可能性があります。そのような配列はコモンブロックに含める等の対策を検討ください。
  - OMP_NESTED
    - 入れ子された並列化を有効または無効にします。デフォルトは無効

ランタイム関数の利用

- OMP_LIBモジュール(Fortranの場合)
  - USE OMP_LIB

- 代表的な実行環境取得関数
  - OMP_GET_NUM_THREADS()
    - 呼び出し時点で並列領域を実行中のスレッド数を返します。
  - OMP_GET_THREAD_NUM()
    - 呼び出したスレッドの番号 (0 ～ スレッド数 − 1) を返します。マスタースレッドは 0 番。
  - OMP_IN_PARALLEL()
    - 並列領域を実行中のとき 0以外の値を、そうでないときには0を返します。

14. MPIプログラミング入門

- MPIの利用方法
- MPIとは
- ループの並列化

MPIの利用方法

- MPIライブラリのリンク方法

  $ ifort mpi_program.f90 -lmpi

- 有用な環境変数

MPI_BUFS_THRESHOLD	バッファを用いた通信において、プロセスまたはホスト辺りのバッファを使うかの閾値を設定します。デフォルトでは64ホストより大きいホスト数でプロセス辺りのバッファを用います。64ホスト以下ではホスト辺りのバッファになります。
MPI_GROUP_MAX	1つのMPIプログラムで使用できるグループの最大数を設定します。デフォルトは32。
MPI_COMM_MAX	MPIプログラムが利用できるコミュニケータの最大値。デフォルトは256。
MPI_BUFFER_MAX	設定された値以上（単位はバイト）の辺りでノード内のシングルコピーを行なう。
MPI_DEFAULT_SINGLE_COPY_OFF	シングルコピーの最適化を行なわない。

MPIプログラムのデバッグ

- MPIプログラムでは、環境変数MPI_SLAVE_DEBUG_ATTACH を設定することで、設定したランクのプロセスが20秒間スリープする
  - スリープさせたいランク番号をセットして実行

```
$ setenv  MPI_SLAVE_DEBUG_ATTACH  0   （ランク0に設定）
$ mpirun -np 4 ./a.out
MPI rank 0 sleeping for 20 seconds while you attach the debugger.
You can use this debugger command:
    gdb /proc/26071/exe 26071
or
    idb -pid 26071 /proc/26071/exe
```

  - 別のシェルからデバッガでアタッチ

```
$ gdb /proc/26071/exe 26071
(gdb) cont
```

MPI とは

- MPI (Message Passing Interface) とは、「メッセージパッシング方式」により通信を行いながら並列計算を行うための API のひとつです。
  - http://www-unix.mcs.anl.gov/mpi/
  - http://www.mpi-forum.org/
- SGI® UV2000システムでは、Message Passing Toolkit (MPT) によって 高性能なMPI を提供しています。
- C、C++、Fortranのどれからでも使うことができます。
- たくさんの関数がありますが、その中の10個程度の関数を知っていれば、基本的なメッセージ通信を行なうことができます。

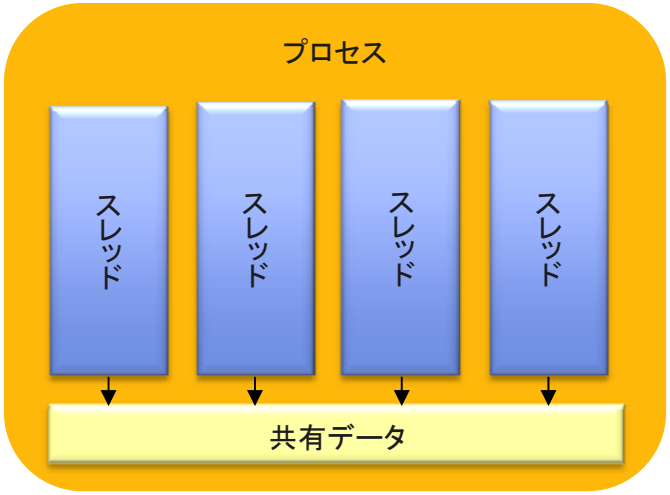
OpenMP と MPI

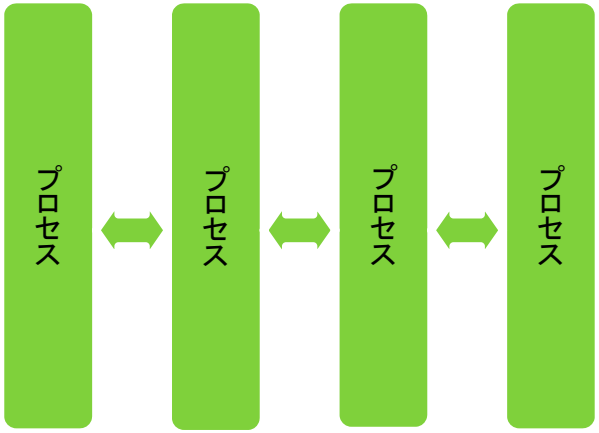
OpenMP

共有メモリモデル
主にループレベルの並列化
段階的な適用が可能で導入が容易

高度なスケーラビリティを得るためには、粒度を大きくするエ夫が必要

MPI
分散メモリモデル
領域分割等による並列化
最初からプログラム全体の並列化必要であり、導入の敷居が高い
粗粒度の並列化
スケーラビリティを得るためには高速なネットワークインターコネクトや遅延隠蔽が必要





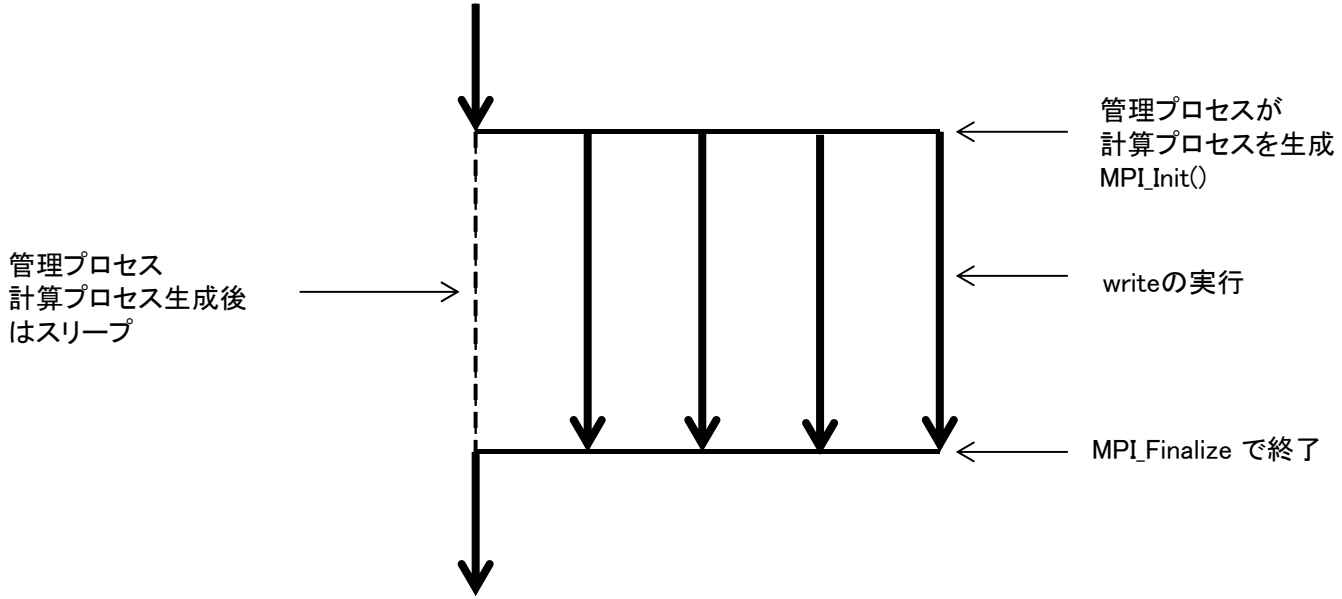
hello, world の並列化

- ヘッダファイル
  - mpif.h をインクルードします。
- MPI_Init と MPI_Finalize
  - MPI プログラムは、必ず MPI_Init( ) で開始し、MPI_Finalize( ) で終了します。

```
program main

    include “mpif.h”

    integer :: ierr, iam, nsize

    call MPI_Init(ierr)
    call MPI_Comm_rank(mpi_comm_world, iam, ierr)
    call MPI_Comm_size(mpi_comm_world, nsize, ierr)
    write(6,*) “hello, world : I am “,iam, “/”,nsize
    call MPI_Finalize(ierr)

    end program main
```

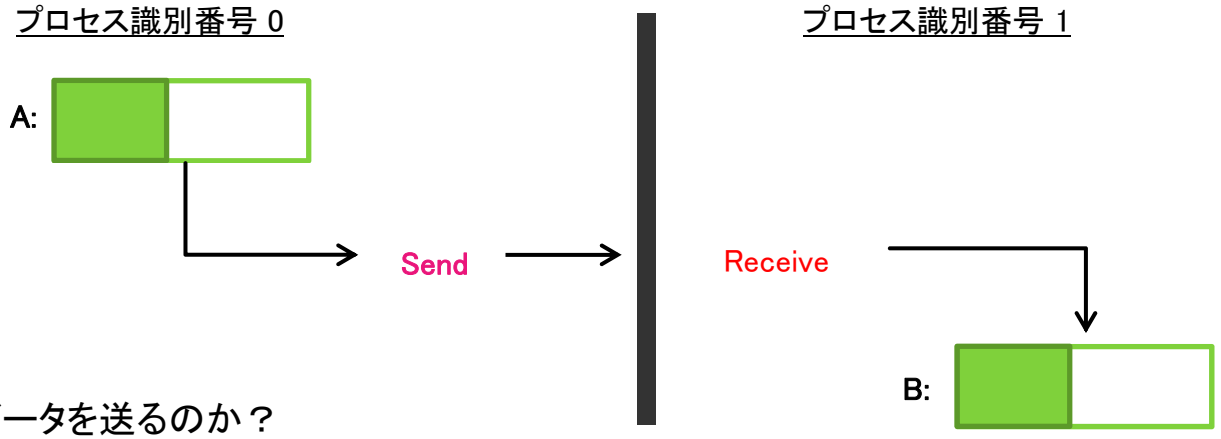
hello, world の実行例

```
$ ifort hello.f  -lmpi
$ mpirun -np 4 ./a.out
 hello, world : I am        1 /        4
 hello, world : I am        2 /        4
 hello, world : I am        3 /        4
 hello, world : I am        0 /        4
$
```



メッセージ・パッシングのプロセス

・基本的なメッセージパッシングのプロセス



– 必要な情報

- どこのデータを送るのか？
- 誰へデータを送るのか？
- 送ろうとするデータの型は？
- どの程度の量のデータを送るのか？
- 受け取る側はどのようにそのデータを識別するのか？

※どこから誰へ送るという記述を行うために、
ランク（rank :プロセス識別番号）という情報を使用します。

メッセージ・パッシングのプログラム例

```
#include <stdio.h>
#include <mpi.h>
int main(int *argc, char ***argv)
{
    int   ierr, iam, nsize, isrc, rval ;
    int   dest=0, tag=0;
    MPI_Status  istat;
    char  message[ ]="Greeting from process";

    ierr = MPI_Init(argc, argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &iam) ;
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &nsize) ;
    dest = 0;
    if(iam != 0) {
        ierr=MPI_Send(&iam, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
    } else {
        for(isrc = 1 ; isrc < nsize ; isrc++) {
            ierr=MPI_Recv(&rval, 1, MPI_INT, isrc, tag, MPI_COMM_WORLD, &istat);
            printf("%s  %d¥n", message, rval);
        }
    }
    ierr = MPI_Finalize();
}
```

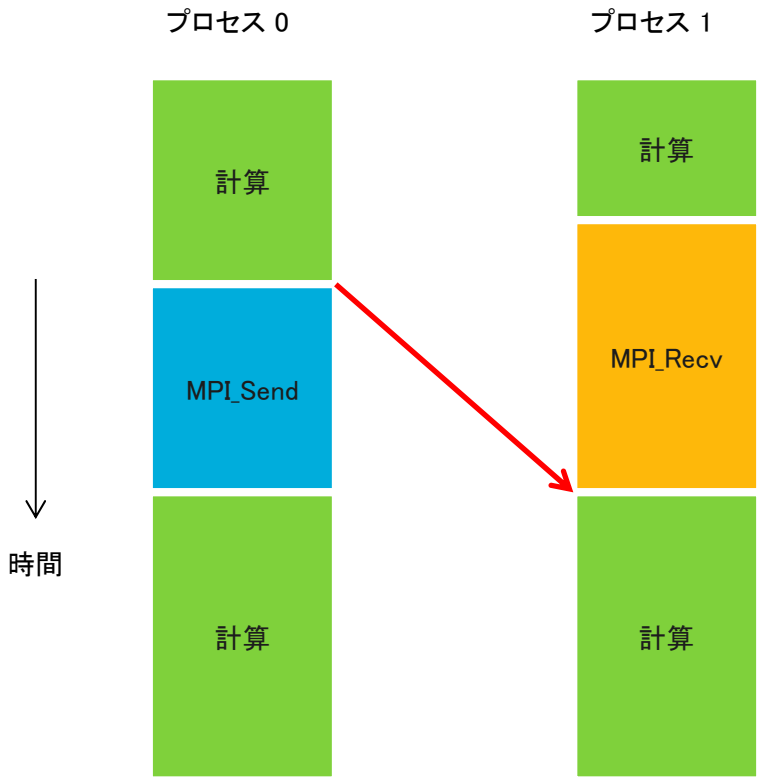
```
$ icc mpi_sample.c -lmpi
$ mpirun -np 4 ./a.out
Greeting from process   1
Greeting from process   2
Greeting from process   3
```

0以外のランクが、ランク0にメッセージを送信する

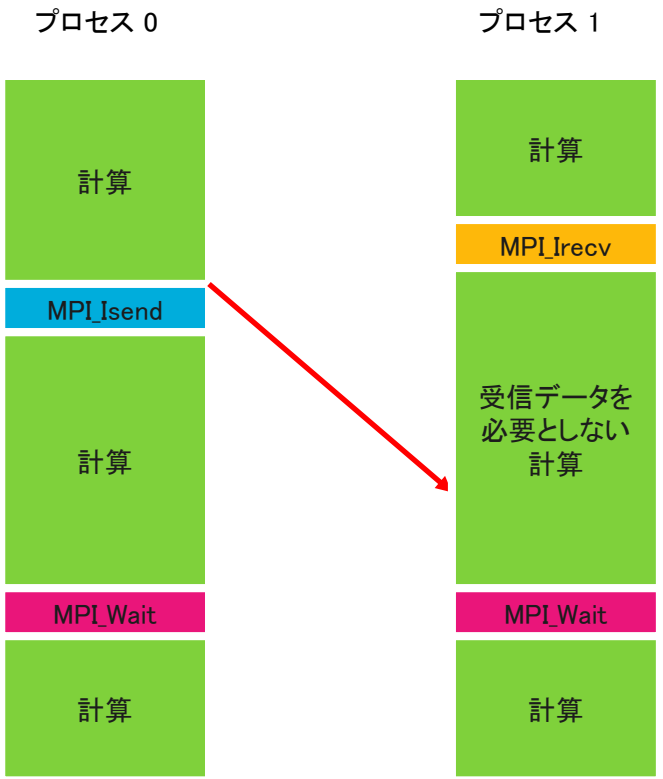
ランク0が、他のランクからのメッセージを受信する

1対1通信

- 同期通信（ブロッキング通信）では、通信が完了するまで、プロセスはブロックされます。



- 非同期通信（ノンブロッキング通信）では、通信はバックグラウンドで行われ、プロセスはブロックされません。

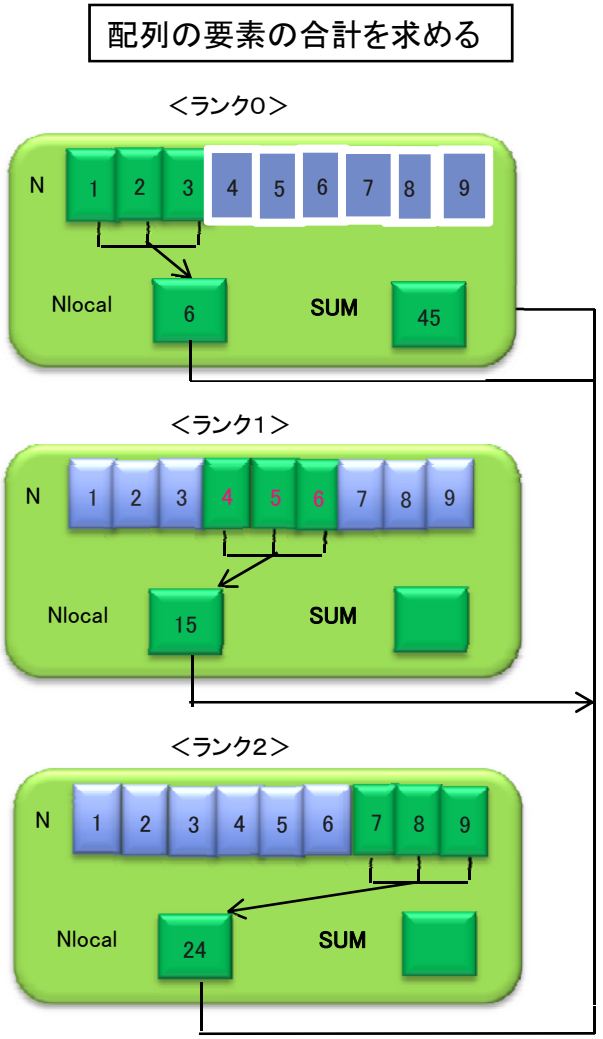


reduction演算：MPI_Reduce

– 通信しながら指定された演算を行い、その結果を1つのプロセスのバッファへ格納します。

– 指定可能な演算は次の通りです。

演算	内容
MPI_MAX	最大値
MPI_MIN	最小値
MPI_SUM	合計
MPI_PROD	積
MPI_LAND	論理AND
MPI_BAND	ビットAND
MPI_LOR	論理OR
MPI_BOR	ビットOR
MPI_LXOR	論理XOR
MPI_BXOR	ビットXOR
MPI_MAXLOC	最大と位置
MPI_MINLOC	最小と位置



15. ハイブリッドプログラミング

- ハイブリッドの利用法
- ハイブリッドプログラムの実行イメージ

ハイブリッドの利用方法

•MPIとOpenMP（自動並列も可）を組み合わせハイブリッドにプログラムを実行することが可能です。

•次の方法でコンパイル & リンクします。

```
$ ifort $(FFLAGS) -openmp -o a.out test.f90 -lmpi
```

hello, worldサンプル

```
program hello_hyb

    use mpi

    implicit none

    integer :: ierr
    integer :: myrank
    integer :: mythread

    integer :: omp_get_thread_num

    call mpi_init(ierr)
    call mpi_comm_rank(mpi_comm_world, myrank, ierr)

!$omp parallel private(mythread)
    mythread = omp_get_thread_num()
!$omp critical
    write(6,*) "hello world. I am MPI=",myrank,"Thread=",mythread
!$omp end critical
!$omp end parallel

    call mpi_finalize(ierr)

    end program
```

MPI_INIT

OpenMP並列実行領域

MPI_FINALIZE

ハイブリッドプログラム実行イメージ

- MPIプロセスが2つ、各MPIプロセスから4スレッドで実行

```
$ export OMP_NUM_THREADS=4
$ mpirun -np 2 omplace -nt ${OMP_NUM_THREADS} ./a.out
```