Hewlett Packard
Enterprise

並列化プログラミング

日本ヒューレット・パッカード合同会社
2024 年 11 月 14 日

Agenda

- TSUBAME4.0 概要
- 並列化の基礎
- Intel コンパイラと最適化
- OpenMP
- MPI
- Linaro Forge
- Hands On

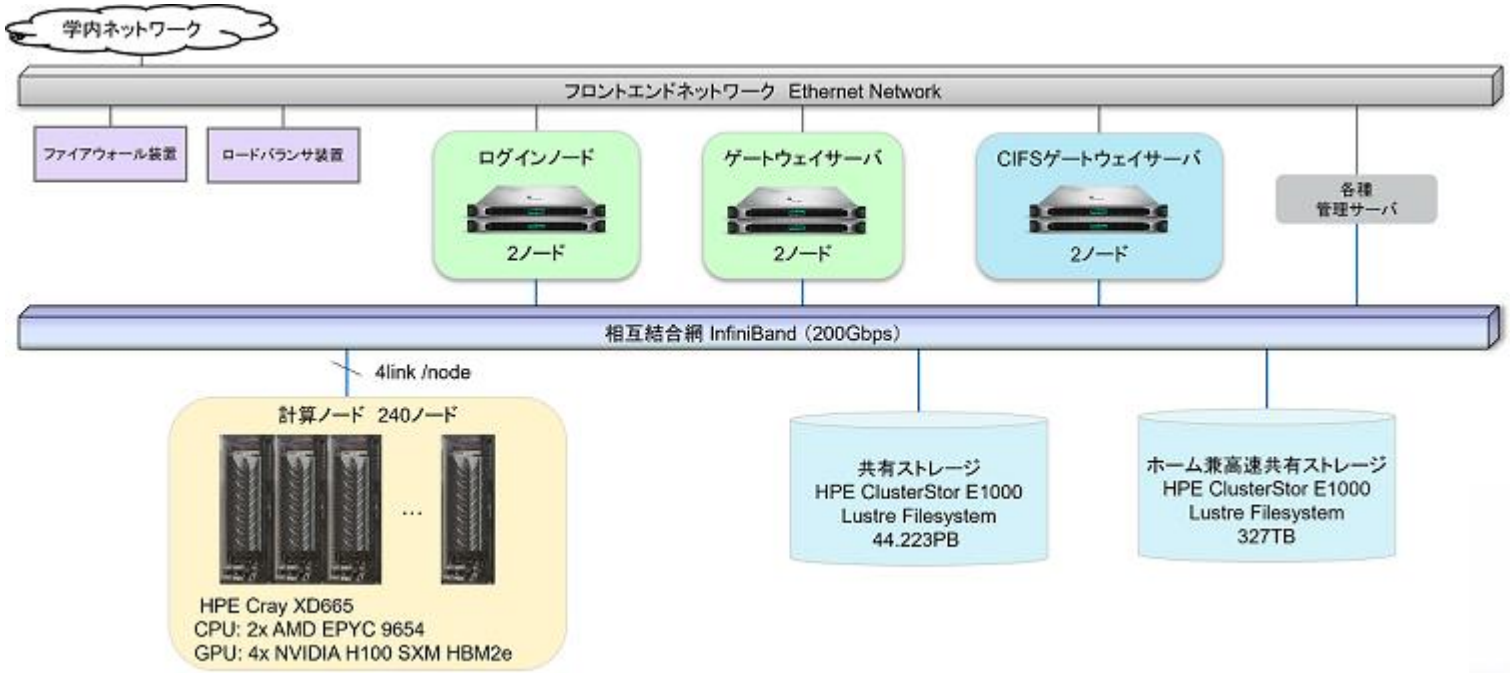
TSUBAME4.0 概要

TSUBAME4.0

- 諸元

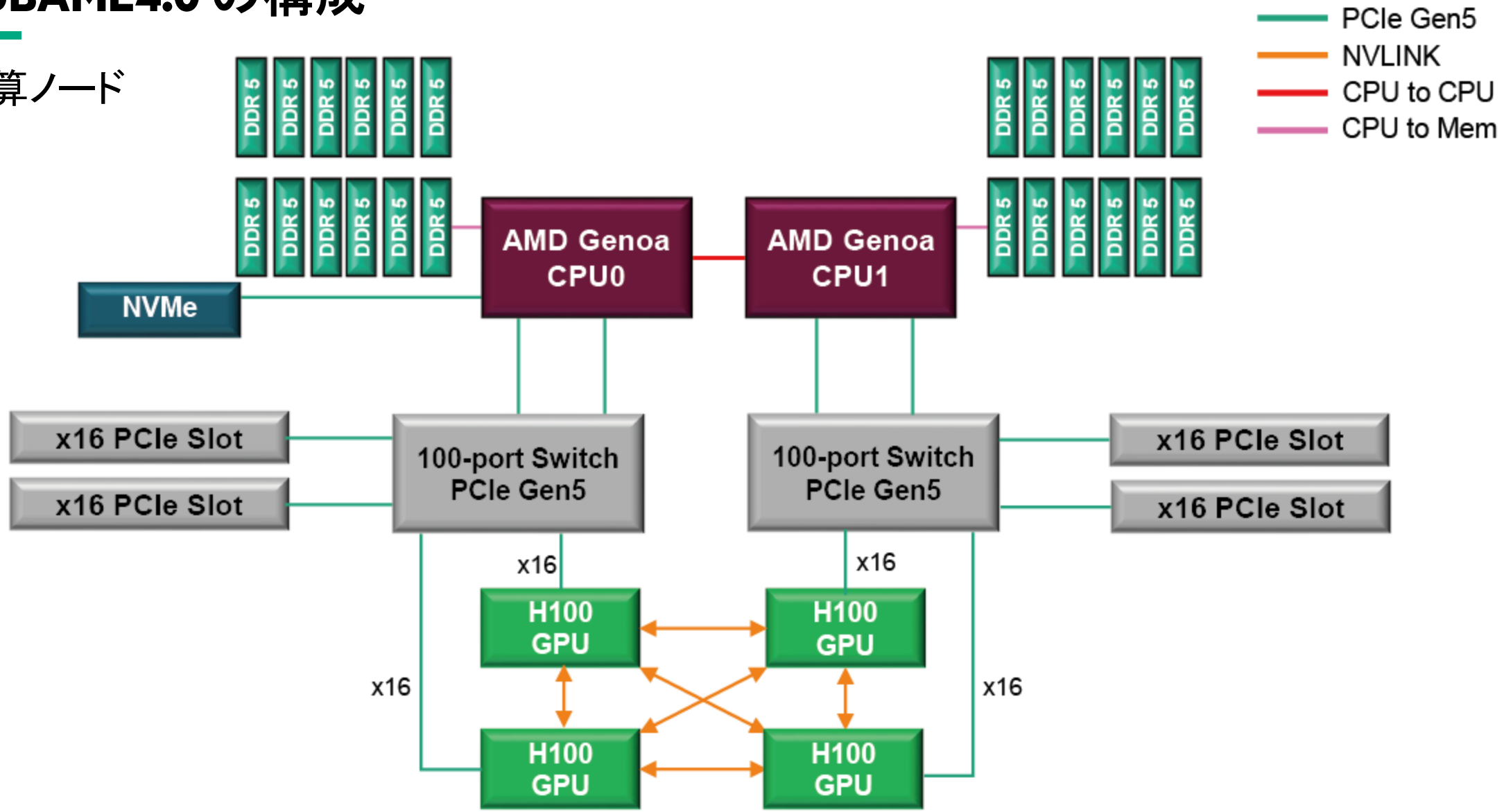
計算ノード	CPU	AMD EPYC 9654 2.4GHz * 2 socket
	コア数/スレッド数	96コア/192 スレッド * 2 socket
	メモリ	768GiB (DDR5-4800)
	GPU	NVIDIA H100 SXM5 94GB HBM2e *4
	インターコネクト	InfiniBand NDR200 200Gbs * 4
システム	ノード数	240
	総コア数	46,080
	理論演算性能(倍精度)	66.8PFLOPS
	理論演算性能(半精度)	952PFLOPS
	ネットワーク	InfiniBand NDR200 200Gbs, ファットツリー

TSUBAME4.0 の構成



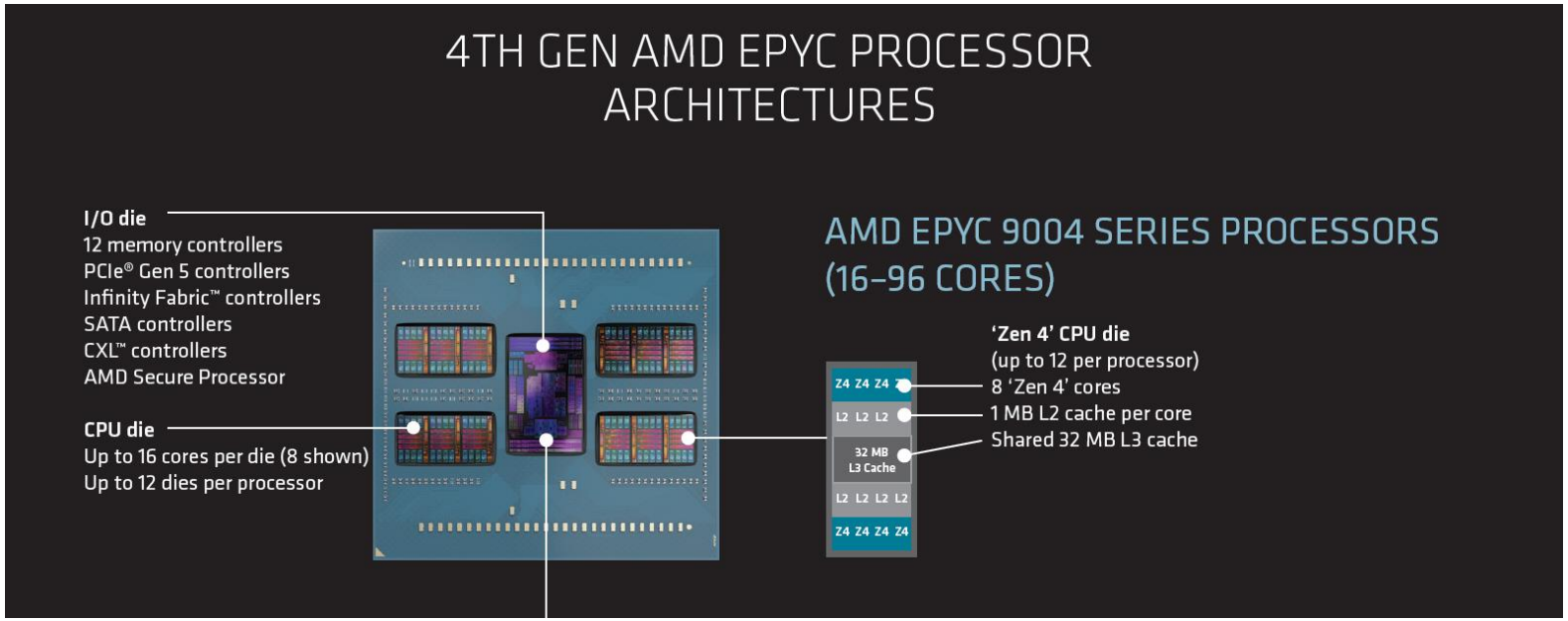
TSUBAME4.0 の構成

- 計算ノード



TSUBAME4.0 の構成

- CPU: AMD EPYC 9654 (コードネーム: Genoa)
  - 96 個のコア
  - クロック
    - ベース: 2.4GHz
    - 最大ブースト: 3.7GHz
  - キャッシュ
    - L1: 32 KB (命令、データとも)
    - L2: 1MB
    - L3: 32MB (shared)
  - テクノロジ
    - AVX-512



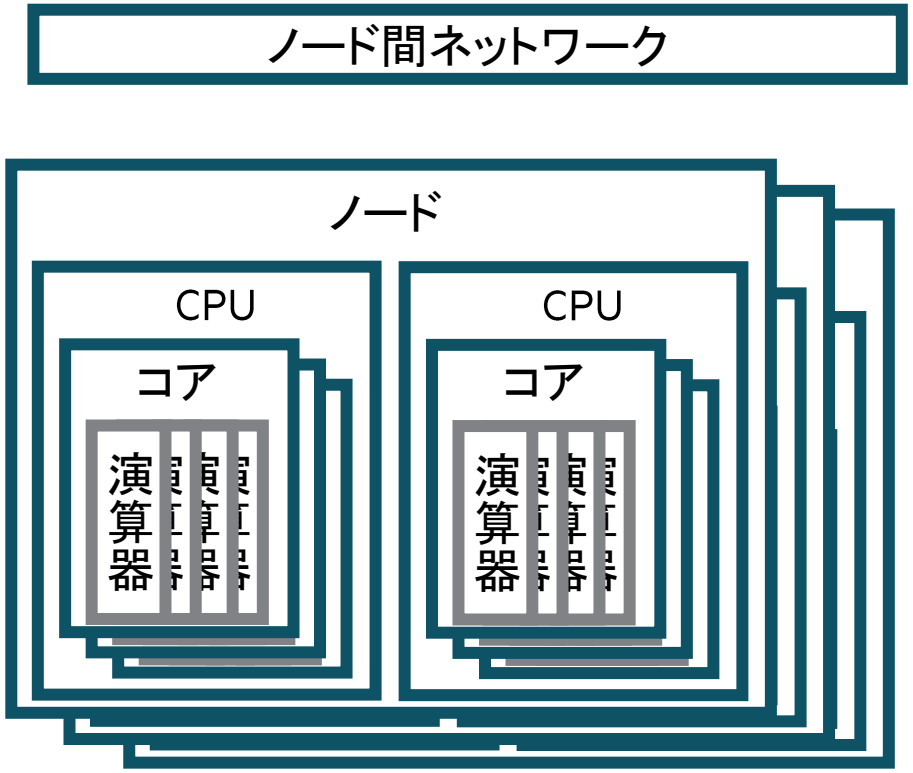
(4TH GEN AMD EPYC™ PROCESSOR ARCHITECTURE https://www.amd.com/en/products/processors/server/epyc/4th-generation-architecture.html より)

並列化の基礎

TSUBAME4.0 における並列性

- 並列計算とは？
  - 処理を独立化した小さな処理に細分化して、複数の処理装置で同時に実行させること。

- TSUBAME4.0 における並列性
  - コア内に複数の演算器
    - キーワード:
      SIMD, ベクトル化、AVX512, ...
  - ノード内の複数の CPU,
    CPU 内の複数のコア
    - キーワード:
      共有メモリ型並列化、OpenMP, NUMA, ...
  - 多数のノード
    - キーワード:
      分散メモリ型並列化, MPI, ...



並列化

- メモリ構成による分類（後述）
  - 共有メモリ型
    - すべての計算要素がメモリを共有している
  - 分散メモリ型
    - 各計算要素が独自のメモリを持ち、他の計算要素のメモリを直接参照できない。

- 並列プログラミングモデル
  - SPMD (Single Program, Multi Data) モデル
    - 一つの並列プログラムしか存在せず、各処理装置に並列プログラムがコピーされ、各処理装置で同じプログラムが実行される
  - Master/Worker モデル
    - Master プログラムが Worker の生成と消去を管理

性能評価指標

- 台数効果
  - 逐次版のプログラムを実行したときの計算時間を T_s, p 台使って並列計算したときの計算時間を T_p とすると、台数効果 S_p は次のようにあらわされます。

$$S_p = T_s / T_p$$

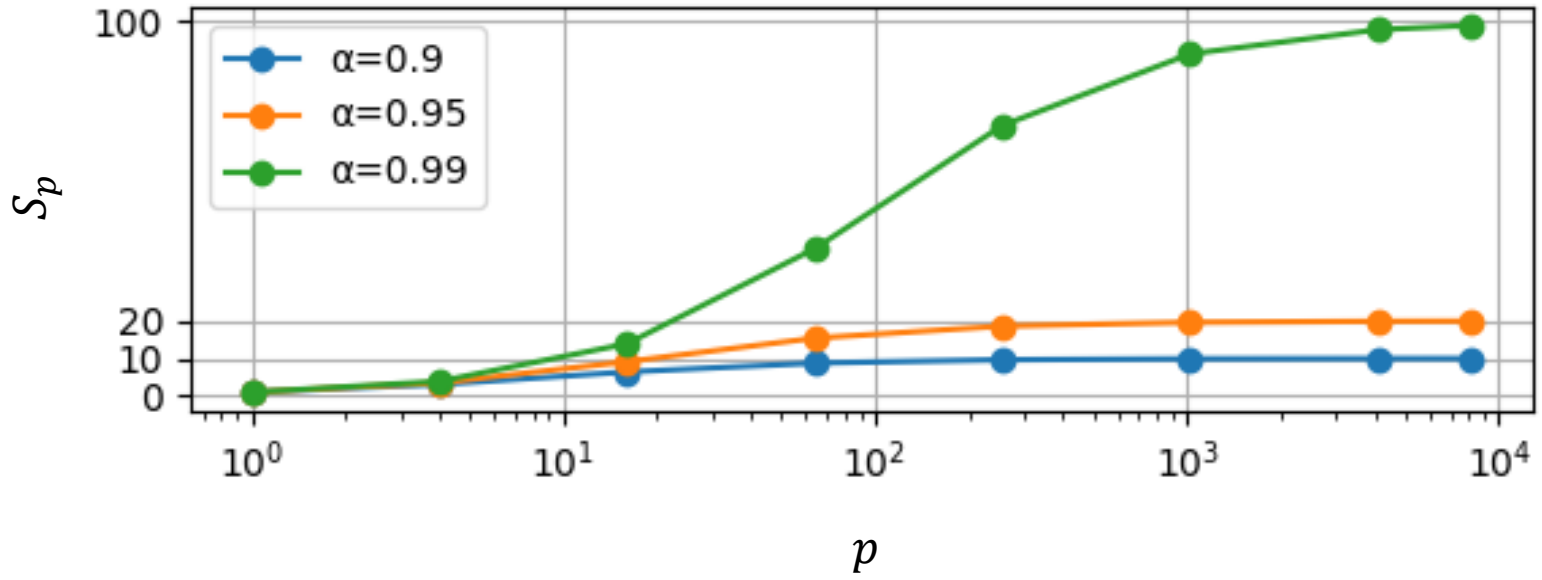
- 並列化効率
  - p 台使って計算した際の並列化効率 E_p [%] は、台数効果 S_p を使って、次のようにあらわされます。

$$E_p = S_p / p \times 100$$

アムダールの法則

- あるプログラムを逐次実行した際の実行時間のうち、並列処理可能な部分の割合をα $(0 \leq \alpha \leq 1)$ とします。このプログラムを p 並列で実行した場合の台数効果 S_p は、並列化のオーバーヘッド等を無視できるとすると、以下の式に従います。

$$S_p = \frac{1}{\frac{\alpha}{p} + (1 - \alpha)}$$



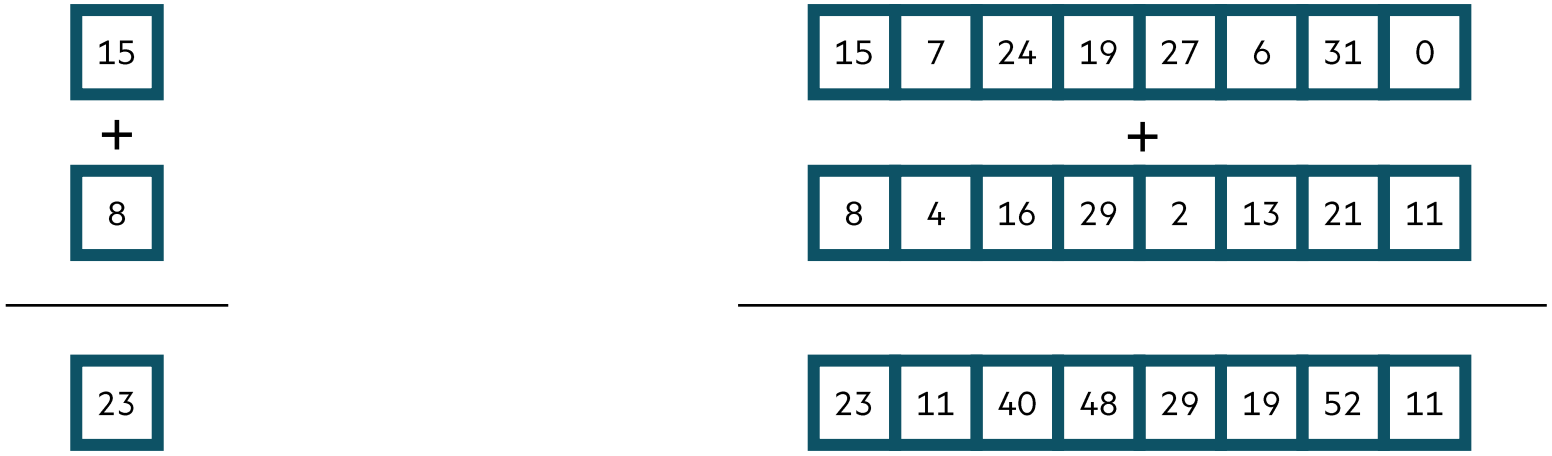
アムダールの法則

- たとえば全体の 90 % が並列化できるプログラム ($\alpha = 0.9$) においては、台数効果 S_p の最大値は 10 になります。

- アムダールの法則の式より、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部分の割合 α を大きくする必要があります。
  - 逐次実行部分を減らす
  - 並列化のオーバーヘッドを減らす

ベクトル化

- SIMD: Single Instruction Multiple Data
  - 一つの命令を同時に複数のデータに適用する並列化。

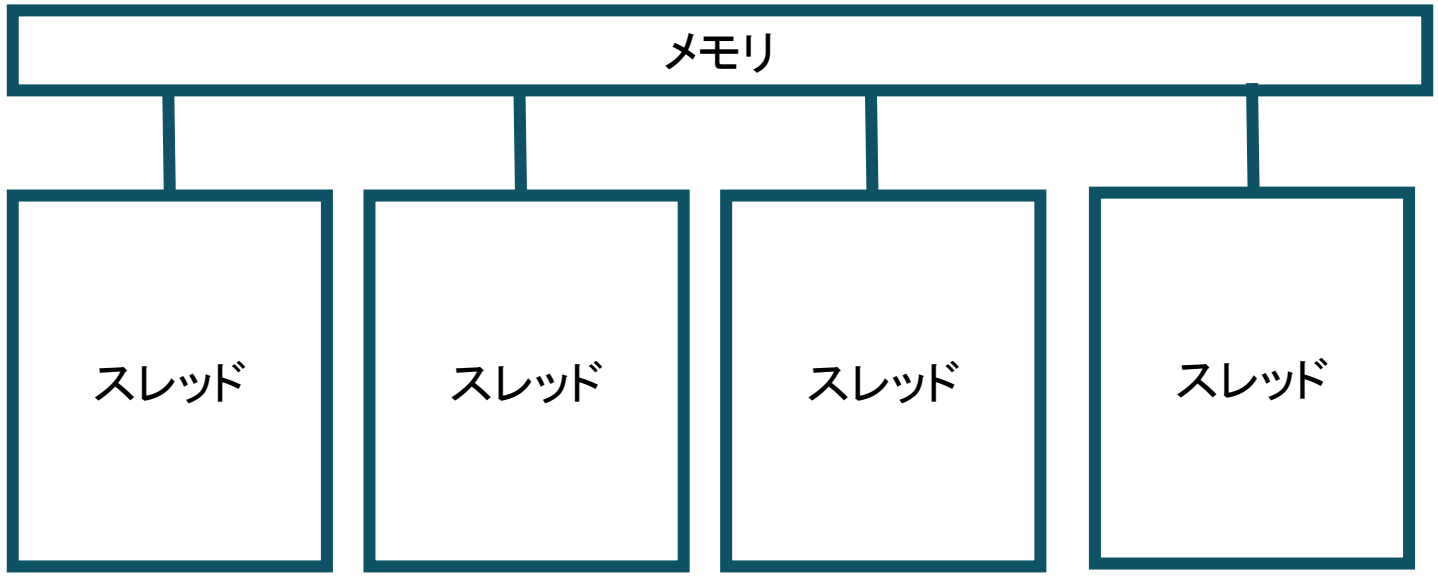


- AVX-512
  - 512 bit のレジスタを利用し、複数のデータ (倍精度ならレジスタ 1 本あたり 8 個のデータを格納可能) に対して一括して同じ演算を行う SIMD 処理方式。

共有メモリ型並列化

- スレッドが基本的な要素
- すべてのスレッドがメモリを共有:
  どのスレッドも等しくメモリにアクセスできる

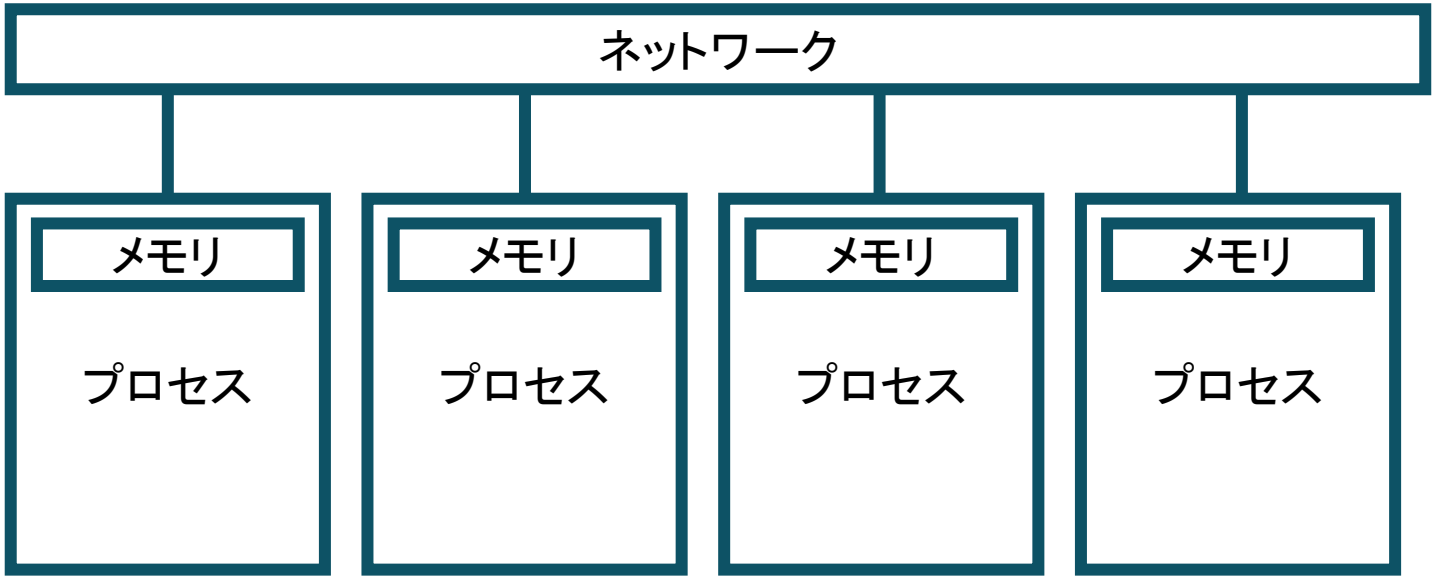
スレッド:
プロセスの中で実行されるひとまとまりの命令。プロセスの中に一つ、もしくは複数存在する。OpenMP での実行単位。他のスレッドとメモリを共有する



分散メモリ型並列化

- プロセスが基本的な要素
- 他のプロセスのメモリに直接アクセスできない
- 明示的なメッセージの受け渡しが必要

プロセス:
動作中のプログラム。OS から記憶領域等の資源を割り当てられている。
MPI における並列化の単位。内部に一つもしくは複数のスレッドを持つ。



Intel コンパイラと最適化

Intel コンパイラの使い方: 環境設定

- Intel コンパイラ

```
$ module load intel
```

- Intel MPI

```
$ module load intel-mpi
```

Intel コンパイラの使い方 (C)

- シリアル

```
$ icx [options] <source>
```

- OpenMP

```
$ icx -qopenmp [options] <source>
```

- MPI

```
$ mpiicx [options] <source>
```

- OpenMP-MPI ハイブリッド

```
$ mpiicx -qopenmp [options] <source>
```

Intel コンパイラの使い方 (C ++)

- シリアル

```
$ icpx [options] <source>
```

- OpenMP

```
$ icpx -qopenmp [options] <source>
```

- MPI

```
$ mpiicpx [options] <source>
```

- OpenMP-MPI ハイブリッド

```
$ mpiicpx -qopenmp [options] <source>
```

Intel コンパイラの使い方 (Fortran)

- シリアル

```
$ ifx [options] <source>
```

- OpenMP

```
$ ifx –qopenmp [options] <source>
```

- MPI

```
$ mpiifx [options] <source>
```

- OpenMP-MPI ハイブリッド

```
$ mpiifx –qopenmp [options] <source>
```

一般的な最適化オプション

オプション	内容
-O	-O2 と同じ
-O0	すべての最適化を無効にします
-O1	コードサイズを増やさないような最適化を行います
-O2	ベクトル化を含む最適化を有効にします。一般的に推奨される最適化レベルです。-O[n] オプションを指定しない場合のデフォルトです。
-O3	-O2 よりも積極的に最適化を行います。ループを用いて多くの浮動小数点演算を行ったり大量のデータを扱ったりするアプリケーションに効果的に働きます。
-Ofast	-O3 のほかに最適化に資するオプションを設定し、高速化を図ります。(gcc との互換性のために用意されている)

プロセッサ固有の最適化オプション

オプション	内容
-x<code>	<code> で指定した命令セットをサポートする専用コードを生成します。このオプションを設定した場合、生成されたバイナリは、下位の命令セットしかサポートしないプロセッサ上では実行できません。
-ax<code>	<code> で指定された命令セットをサポートするプロセッサ向けに、代替コードを生成します。このオプションを設定した場合、生成されたバイナリを下位の命令セットしかサポートしないプロセッサ上で実行すると、汎用コードでの実行になります。
-xHost	コンパイルを実行しているシステムでサポートされる最上位の命令セットを選択します。TSUBAME4.0 ではログインノードと計算ノードの CPU が異なりますので注意してください。

TSUBAME4.0 で推奨される最適化オプション

オプション	内容
-O3	ベクトル化などの最適化に加え、ループの融合、アンロール、IF 文への対応をはじめとする強力なループ変換など、積極的なベクトル化を行います。
-xCORE-AVX512	TSUBAME4.0 の CPU, AMD EPYC9654 は Intel AVX-512 命令をサポートしています。このオプションにより、AVX-512 命令セット対応のプロセッサ向けの最適化が行われます。

プロシージャ間最適化オプション

オプション	内容
-ipo	複数のソースファイルにまたがるインライン展開や、その他のプロシージャ間の最適化を行います。コンパイル中のコードに対してより多くの情報が得られるため、追加の最適化が可能です。条件により、コンパイル時間やコードサイズが大幅に増えることがあります。

浮動小数点数値演算の制御

オプション	内容
-fp-model=fast	計算結果に影響がある最適化を許可します。
-fp-model=precise	計算結果に影響しない最適化のみ許可します。
-fp-model=strict	最も厳密な浮動小数点モデルを採用します。厳密な浮動小数点例外セマンティクスを有効にします。

メモリモデル

オプション	内容
-mcmodel=small	コードとデータがアドレス空間の最初の 2 GB に収まることをコンパイラに指示します。(デフォルト)
-mcmodel=medium	コードがアドレス空間の最初の 2 GB に収まることをコンパイラに指示します。データにはこのようなメモリに関する制限を課しません。
-mcmodel=large	コード、データに関して、small や medium にあったようなメモリに関する制限を課しません。

- 2GB バイトを超えるようなグローバルメモリ、スタティックメモリを使うプログラムは、-mcmodel=medium や –mcmodel=large を指定してビルドしてください。
- -mcmodel=small もしくは -mcmodel=large を指定した場合、--shared-intel オプションも自動的に指定されます。

デバッグ情報、最適化レポートオプション

- デバッグ情報オプション

オプション	内容
-g	デバッグ情報の生成を行います。このオプションが指定された場合、他に指定がなければ -O0 オプションが設定されます。

- 最適化レポートオプション

オプション	内容
-qopt-report[=<N>]	最適化レポートを作成します。デフォルトでは、レポートは optrpt 拡張子を持つファイルに出力されます。レポートのレベル <N> を指定することもできます。0 (レポート無し) から 3 (最も詳細なレポート) を選択できます。
-qopt-report-file=<keyword>	最適化レポートの出力先を制御します。<keyword> には出力先のファイルパスを指定するか、stdout (標準出力への出力), stderr (標準エラー出力の出力) を指定します。

最適化レポート例

```
$ icx -O2 -qopt-report=2 -qopt-report-file=stdout \
    ./matmul.c
Global optimization report for : matmul

...<略>

LOOP BEGIN at ./matmul.c (18, 3)
    remark #15553: loop was not vectorized: outer loop
vectorization candidate.

    LOOP BEGIN at ./matmul.c (19, 5)
        remark #15553: loop was not vectorized: outer loop is not an
auto-vectorization candidate.

        LOOP BEGIN at ./matmul.c (20, 7)
            remark #15335: loop was not vectorized: vectorization
possible but seems inefficient. Use vector always directive or -vec-
threshold0 to override
            remark #25438: Loop unrolled without remainder by 8
        LOOP END
    LOOP END
LOOP END
```

```
$ cat -n matmul.c
...
    17
    18    for(i=0; i<SIZE; i++) {
    19      for (j = 0; j < SIZE; j++){
    20        for (k = 0; k < SIZE; k++) {
    21          c[i][j] = c[i][j] + a[i][k] * b[k][j];
    22        }
    23      }
    24    }
...
```

補足: AMD Optimizing C/C++ and Fortran Compilers (AOCC)

- 環境設定

```
$ module load aocc
```

- C

```
$ clang [options] <source>
```

- C++

```
$ clang-cpp [options] <source>
```

- Fortran

```
$ flang [options] <source>
```

- Intel Compiler とのオプションの対応

	Intel Compiler	AMD Compiler
最適化オプション	-O<n>	-O<n>
積極的な最適化を行うコンパイラオプション	-Ofast	-Ofast
TSUBAME4.0 アーキテクチャでの最適化オプション	-axCORE-AVX512	-march=znver4
デバッグ情報オプション	-g	-g
プロシージャ間最適化	-ipo	-flto (リンク時最適化)
OpenMP	-qopenmp	-fopenmp (C/C++) -mp (Fortran)

AMD EPYC 9xx4-series Processors Compiler Options Quick Reference Guide
https://www.amd.com/content/dam/amd/en/documents/developer/version-4-2-documents/aocc/aocc-4.2-quick-reference-guide.pdf

参考資料

- Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference: Compiler Options
  https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-1/compiler-options.html

- Intel® Fortran Compiler Classic and Intel® Fortran Compiler Developer Guide and Reference: Compiler Options
  https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2024-0/compiler-options-001.html

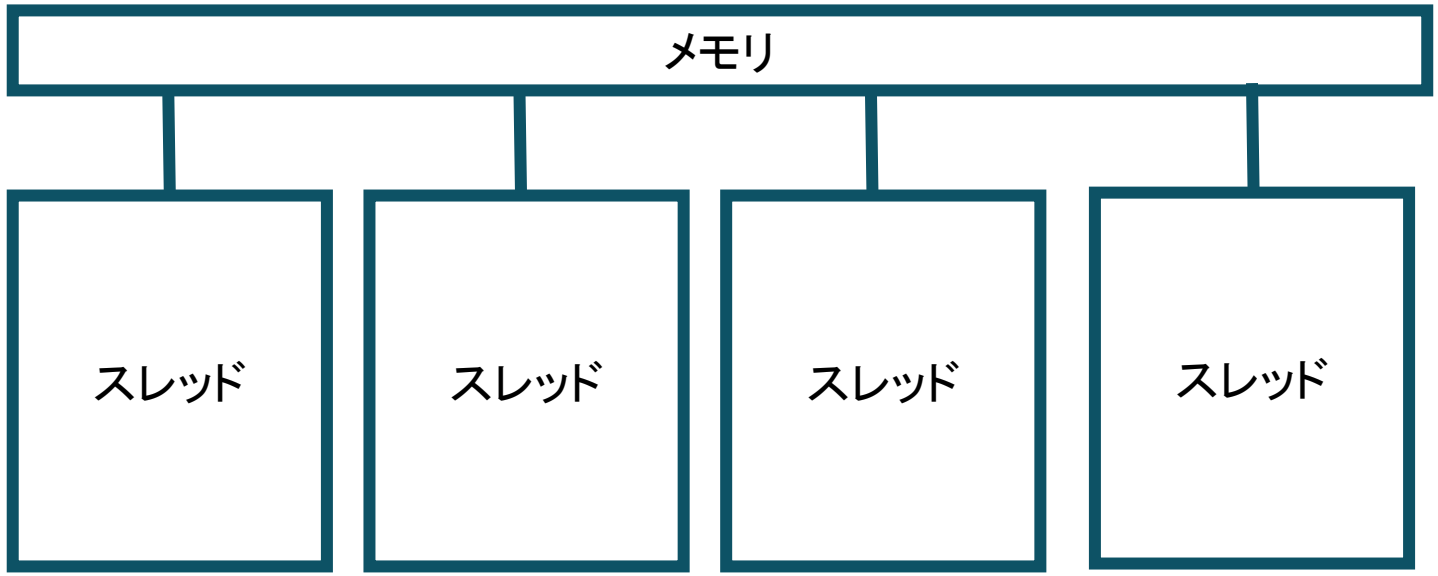
- AMD Optimizing C/C++ and Fortran Compilers (AOCC)
  https://www.amd.com/en/developer/aocc.html

- AMD EPYC 9xx4-series Processors Compiler Options Quick Reference Guide
  https://www.amd.com/content/dam/amd/en/documents/developer/version-4-2-documents/aocc/aocc-4.2-quick-reference-guide.pdf

OpenMP

OpenMPとは

- OpenMP とは、共有メモリ型並列計算機でマルチスレッド型の並列計算プログラムを作るために作られた API (Application Programming Interface)です。



OpenMPとは

- OpenMP のコンポーネント
  - 指示文 (C 言語では #pragma omp，Fortran では !$omp で始まる行)
  - ライブラリ
  - 環境変数

- 特徴
  - ソースコード内に指示行を記述することで並列化します。
  - 分散メモリ型の並列化に比べ、実装が容易、逐次的な実装が可能です。
  - 共有メモリ型なので、ノード間の並列化には使えません。
  - C/C++, Fortran で利用可能です。
  - Fork-join モデル
    - Parallel 構文で囲まれた場所のみが複数スレッドで並列実行され、それ以外は一つのスレッドで実行されます。

Fork-join model

C/C++

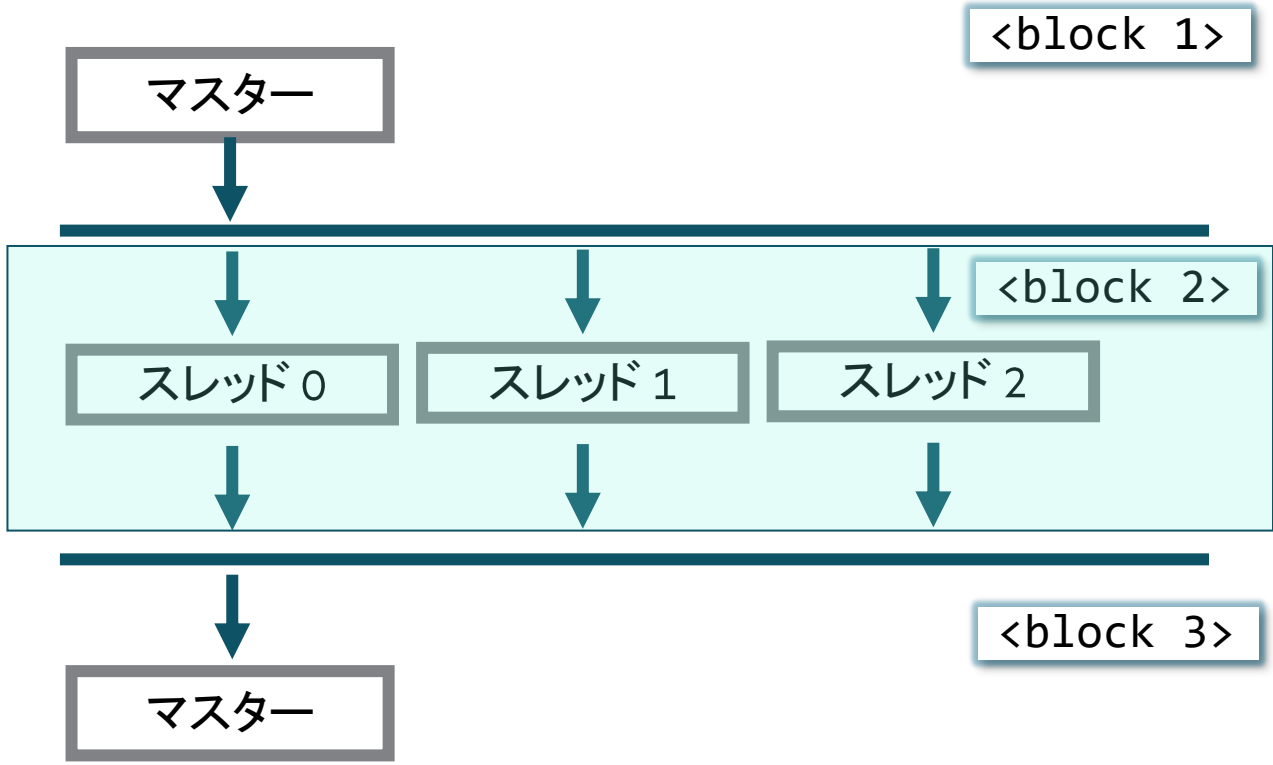
```
<block 1>
#pragma omp parallel
{
    <block 2>
}
<block 3>
```

並列領域

Fortran

```
<block 1>
!$omp parallel
    <block 2>
!$omp end parallel
<block 3>
```

並列領域



OpenMP のビルドと実行

- 環境設定

```
module load intel
```

- ビルド
  - C 言語

```
icx -qopenmp <source code>
```

  - C++

```
icpx -qopenmp <source code>
```

  - Fortran

```
ifx -qopenmp <source code>
```

オプション -qopenmp を与えると、OpenMP の指示文を解釈して並列化されたコードを生成します。
-qopenmp が与えられない場合、OpenMP の指示文は無視されます。

- 実行
  - 環境変数 OMP_NUM_THREADS

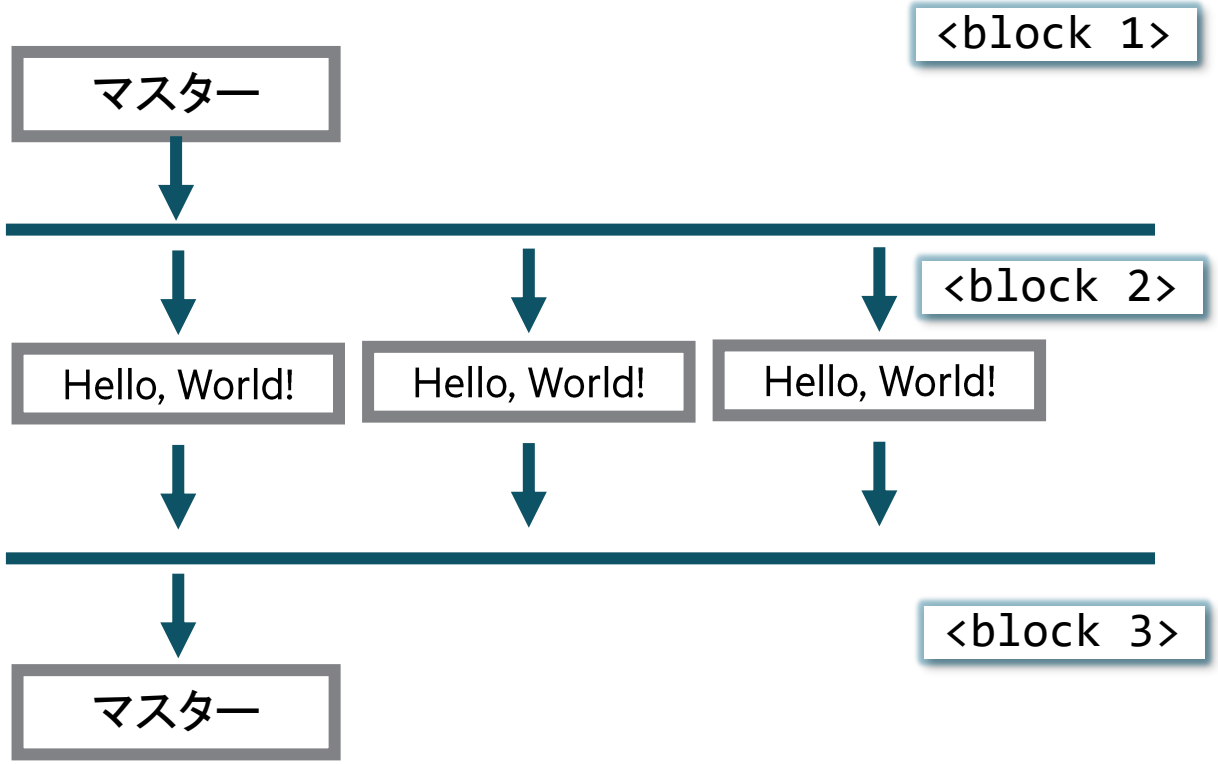
```
OMP_NUM_THREADS=<並列数>
```

OMP_NUM_THREADS が指定されない場合、intel コンパイラでコンパイルされたバイナリは利用可能なすべてのコアを利用します。

Hello World

```
#include<stdio.h>

int main(){
    #pragma omp parallel
    {
        printf("Hello, World!\n");
    }
}
```

```
$ module load intel
$ icx -qopenmp openmp.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello, World!
Hello, World!
Hello, World!
Hello, World!
$
```



OpenMP 指示文の書式

- C/C++

```
#pragma omp <ディレクティブ名> [節[, 節…]]
```

- Fortran

```
!$omp <ディレクティブ名> [節[, 節…]]
```

- OpenMP では並列化の指示を指示文で与えます。
- 指示文は、C/C++ では #pragma omp で、Fortran では !$omp で始まります。
- ディレクティブ名と、それを修飾する節で記述されます。
- コンパイラが OpenMP に対応していない場合 (もしくは OpenMP の機能を有効にしていない場合), 指示文は無視されます。

Parallel 指示文

```
#pragma omp parallel [節[, 節…]]
{
    <並列領域>
}
```

```
!$omp parallel [節[, 節…]]
    <並列領域>
!$omp end parallel
```

- 続くブロックが並列領域であることを指示するディレクティブです
- 並列領域を実行する OpenMP スレッドを作ります。

for/do 指示文

```
#pragma omp do [clause[, clause..]]
for(i=0;i<N;i++){
    <block>
}
```

```
$!omp do [clause[, clause..]]
do i=0, n
    <block>
enddo
$!omp end do
```

- 並列領域で使用します。
- 後続する for ループ/do ループを各スレッドで分担処理します。
- デフォルトでは、各スレッドが等しく処理を分担します。

例

```
double a[100], b[100], c[100];
…
#pragma omp parallel for
    for (i=0; i<100; i++){
        a[i] = b[i] + c[i];
    }
```

スレッド	index
0	0 - 24
1	25 - 49
2	50 - 74
3	75 – 99

for/do 指示文のオプション (節)

- 主な節

節	内容
private(list)	list の変数をスレッドにプライベートな変数にします。
shared(list)	list の変数を各スレッドから共有される変数にします。
schedule(kind, chunk)	kind で指定された方法でループをスレッドに割り当てます。 kind には static, dynamic, guided などが指定可能です。
reduction(operator, var)	var で指定された変数を operator で縮約演算します。
nowait	ループの最後での同期を行いません。

for/do 指示文: データスコープ

- OpenMP は共有メモリ型:
  基本的に変数は shared 変数(すべてのスレッドから等しくアクセスされる変数) です。ループ内に現れる変数も、for/do 指示文の直後のループ変数は例外として、shared 変数になります。
- そこで、必要な変数を private 変数 (スレッドごとに独立して確保される変数) にする宣言が必要になります。
- OpenMP では、private 変数を private 節により宣言することが可能です。

```
private(<変数名>[, <変数名>]…)
```

```
#pragma omp do private(j, factor)
for(i=0;i<N;i++){
    factor=b[i]
    for(j=0;j<N;i++){
        a[j] += factor * c[j]
    }
}
```

for/do 指示文: データスコープ

- そのほかのデータスコープ節

節	内容
firstprivate	並列領域開始時にマスタースレッドの値を全スレッドにコピーします。
lastprivate	並列領域終了時のマスタースレッドの値を並列実行時の最後の値とします。
copyprivate	single 指示文で指定された範囲を実行後、値を全スレッドにコピーします。
default	指示文や節で属性が指定されていない変数のデフォルトの属性を指定します。

for/do 指示文: reduction

- 右は配列 a の総和を計算するループです。

- スレッドに分割すると、それぞれのスレッドが s の値を書き換えるので、正しい値が計算されません。

- s のデータスコープを private や lastprivate にしても、問題は解決されません。

逐次

```
s=0
for(i=0;i<100;i++){
    s+=a[i]
}
```

スレッド 0

```
s=0
for(i=0;i<49;i++){
    s+=a[i]
}
```

スレッド 1

```
s=0
for(i=50;i<100;i++){
    s+=a[i]
}
```

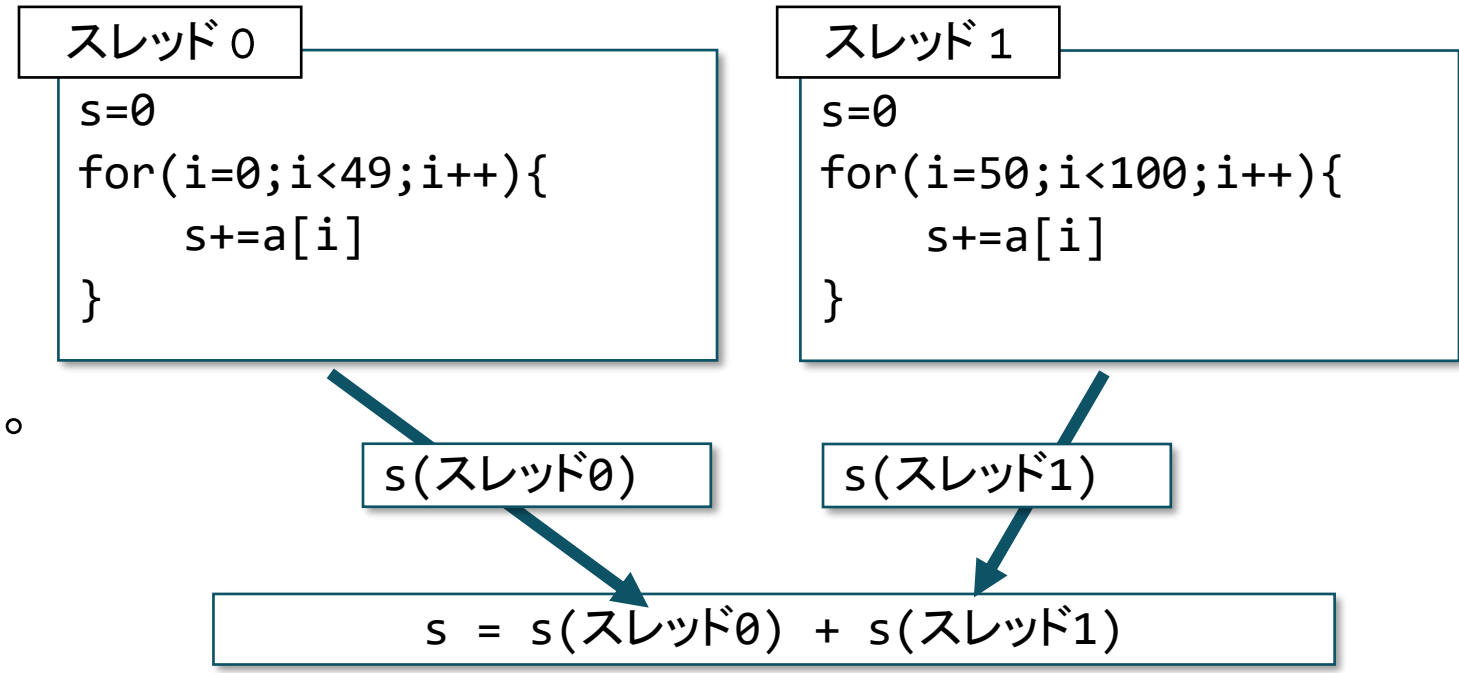
for/do 指示文: reduction

- Reduction 節を追加することにより、総和計算が可能になります。

```
reduction(演算子, 変数名)
```

- 並列計算中、各スレッドは一時的な private 属性の変数に計算結果を集約します。
- 並列計算終了後、指定された演算で値を集約して格納します。
- 計算順序が逐次プログラムと異なります。その結果、丸め誤差の影響で計算結果が逐次版と異なることがあります。

```
s=0
#pragma omp for reduction (+: s)
for(i=0;i<100;i++){
    s+=a[i]
}
```



for/do 指示文: reduction

```
#include<stdio.h>

int main(){
  double
   a[12]={3.,8.,12.,5.,6.,4.,9.,11.,2.,7.,10.,1.};
  double s;
  int i;
  s=0.;
#pragma omp parallel
  {
#pragma omp for reduction(+: s)
    for (i=0;i<12;i++){
      s+=a[i];
    }
  }
  printf("s= %lf\n",s);
  return 0;
}
```

- 逐次処理

```
$ icx reduction.c
$ ./a.out
s= 78.000000
```

- OpenMP reduction 節

```
$ icx -qopenmp reduction.c
$ export OMP_NUM_THREADS=4
$ ./a.out
s= 78.000000
```

- reduction 節省略時

```
$ icx -qopenmp reduction.c
$ export OMP_NUM_THREADS=4
$ ./a.out
s= 33.000000
```

for/do 指示文: スケジューリング

- スケジューリング節を追加することにより、スレッドへの処理の割り当てを変えることができます。

```
schedule(kind, chunksize)
```

- kind で割り当て方法を指定します。主なものは下記のとおりです。

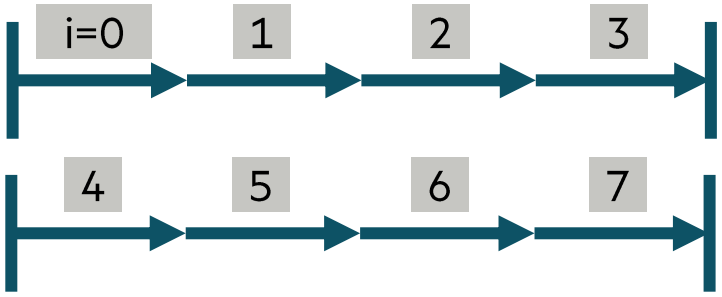
kind	内容
static	ループを chunksize のチャンクに分割し、スレッド番号順で各スレッドに割り付けます。
dynamic	各スレッドは chunksize の数だけループを実行し、実行が終わったら次のチャンクを要求します。

for/do 指示文: スケジューリング

Private 節を省略しています

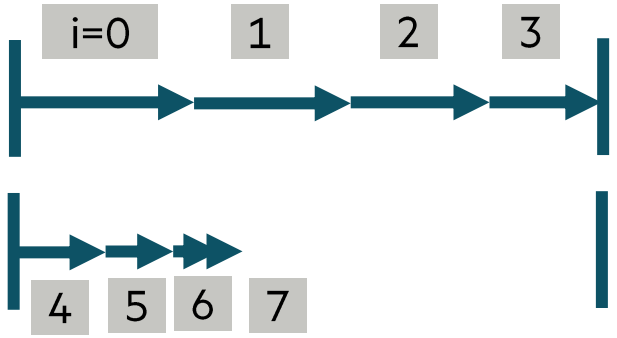
- デフォルトでは、ループは等分割されます。

```
#pragma omp do
for(i=0;i<8;i++){
    factor=b[i]
    for(j=0;j<8;j++){
        a[j] += factor * c[j]
    }
}
```



- 下記の場合は、スレッド により処理量が大きく異なります。

```
#pragma omp do
for(i=0;i<8;i++){
    t=a[i]
    for(j=i;j<8;j++){
        c[i][j] = c[i][j] / t
    }
}
```

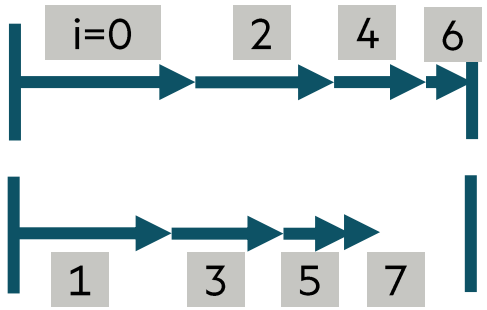


for/do 指示文: スケジューリング

Private 節を省略しています

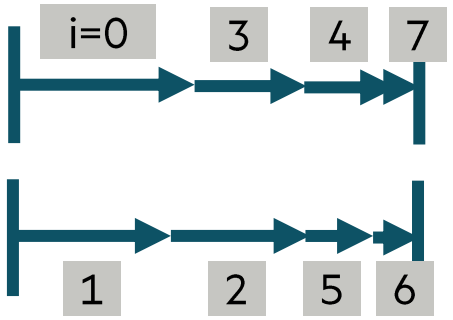
- shedule 指示節を使い、chunk サイズを減らすことにより、ロードインバランスを緩和することができます。

```
#pragma omp do schedule(static, 1)
for(i=0;i<8;i++){
    t=a[i]
    for(j=i;j<8;j++){
        c[i][j] = c[i][j] / t
    }
}
```



- スケジューリングに dynamic を使用すると、さらに緩和することができます。

```
#pragma omp do schedule(dynamic, 1)
for(i=0;i<8;i++){
    t=a[i]
    for(j=i;j<8;j++){
        c[i][j] = c[i][j] / t
    }
}
```



dynamic スケジューリングにより処理量をより均等に割り当てることが可能ですが、割り当て処理には static より時間がかかります。

そのほかの主な指示文

- スレッドの制御にかかわる指示文

指示文	内容
single	指定した範囲を一つのスレッドで実行します。
master	指定した範囲を master スレッドで実行します。
barrier	全スレッドの処理が終わるまで待ちます
critical	指定した範囲を同時に実行するスレッドを1 つに制限します。
atomic	直後の代入文について、複数のスレッドが安全に共有変数を更新できるようにします。

OpenMP の主な環境変数

環境変数	意味
OMP_NUM_THREADS	並列領域でのスレッド数を指定します。 並列領域が指定されない場合、利用できるすべてのコアを利用するよう試みます。
OMP_STACKSIZE	OpenMP の各スレッドが使うスタックサイズを設定します。 単位を B, K, M, G, T で指定できます。単位が指定されない場合、K (Kilobytes) になります。 デフォルト: 4M 推奨: 16M

OpenMP を使うプログラムで segmentation fault が発生した場合、OMP_STACKSIZE を増やすことで問題が解消されることがあります。

OpenMP の主なランタイム・ライブラリルーチン

ルーチン	意味
omp_set_num_threads	スレッド数を設定します。呼び出し後の parallel 領域で適用されるスレッド数に影響します。
omp_get_num_threads	現在のチームのスレッド数を取得します。
omp_get_thread_num	自身のスレッド番号を取得します。
omp_get_num_procs	デバイスで利用可能なプロセッサ数を取得します。

- ライブラリルーチンを使う場合は、下記のような include文/use 文が必要です
  - C/C++

```
#include<omp.h>
```

  - Fortran

```
use omp_lib
```

並列化できないループ

- ループ長が確定していないループ
  - C の while ループ、Fortran の do while ループなど
- ループ途中でのループ終了命令
- ループ内の依存性
  - 後方依存性
  - 前方依存性
  - 間接参照のあるループ
- 関連: 縮約演算
  - ループ内に依存性があるが、並列化できるループ

後方依存性

```
for(i=0;i<100;i++){
    a[i] = a[i-1] + b[i]
}
```

```
for(i=0;i<50;i++){
    a[i] = a[i-1] + b[i]
}
```

```
for(i=50;i<100;i++){
    a[i] = a[i-1] + b[i]
}
```

- i=50 の計算時、a[49] が必要であり、スレッド 1 の計算が終わるまで待たないと正しい計算結果が得られません。

前方依存性

```
for(i=0;i<100;i++){
    a[i] = a[i+1] + b[i]
}
```

```
for(i=0;i<50;i++){
    a[i] = a[i+1] + b[i]
}
```

```
for(i=50;i<100;i++){
    a[i] = a[i+1] + b[i]
}
```

- i=49 の計算時、値が更新される前の a[50] が必要であるため、スレッド 1 で a[50]がすでに更新されている場合には値が不正になります。
- 一時的にデータを格納する配列を作り、a を一旦その配列に格納することで、依存性が回避できる場合があります。

間接参照のあるループ

```
for(i=0;i<100;i++){
    a[index[i]] = b[i]+c[i]
}
```

- 配列 a を添え字配列 Index で参照しています。Index[i] に重複する値があれば、並列化した場合に逐次の場合と結果が変わる可能性があります。
- Index[i] に重複がないことが保証されていれば、並列化が可能です。

縮約演算

```
s=0
for(i=0;i<100;i++){
    s += a[i]
}
```

- i step の s を計算するためには i-1 ステップの s の値が求まっている必要があるので並列化は不可能に思えます。しかし、for 指示文の reduction 節の説明で見た通り、実際は並列化が可能です。

参考資料

- 片桐　孝洋 「並列プログラミング入門: サンプルプログラムで学ぶ OpenMP と OpenACC」
  東京大学出版会, 2015

- OpenMP: The OpenMP API specification for parallel programing
  https://www.openmp.org/
  - OpenMP Reference Guides
    https://www.openmp.org/resources/refguides/

MPI

MPI とは

- Message Passing Interface (MPI) とは、メッセージ・パッシングのライブラリの規格の一つです。

- 特徴
  - 分散メモリ型の並列化を行います。
  - ライブラリ関数を用い、明示的にメッセージ・パッシングを行うことで、並列化を行います。
  - ノード間の並列化にも、ノード内の並列化にも使えます。
  - SPMD (Single Program Multiple Data) モデル
    – 一つの共通のプログラムが、並列処理開始時にすべてのプロセッサ上で起動します。
    – MPMD（Multi Program Multiple Data）も可能です。
  - C/C++, Fortran から利用可能です。そのほかのプログラミング言語でも、モジュール/パッケージ/ライブラリが用意されていて、MPI が利用できる場合があります。(例: Python における mpi4py)
  - 数百の関数が定義されています。(ですが、最初は数種類覚えれば十分です)

MPI のビルド

- 環境設定

```
module load intel
module load intel-mpi
```

Intel のモジュールファイルに加えて、
intel-mpi もロードしてください

- C 言語

```
mpiicx <source file>
```

- C++

```
mpiicpx <source file>
```

- Fortran

```
mpiifx <source file>
```

- MPI の実行

```
mpiexec.hydra -np <プロセス数> ./a.out
```

Hello World

- ソースファイル

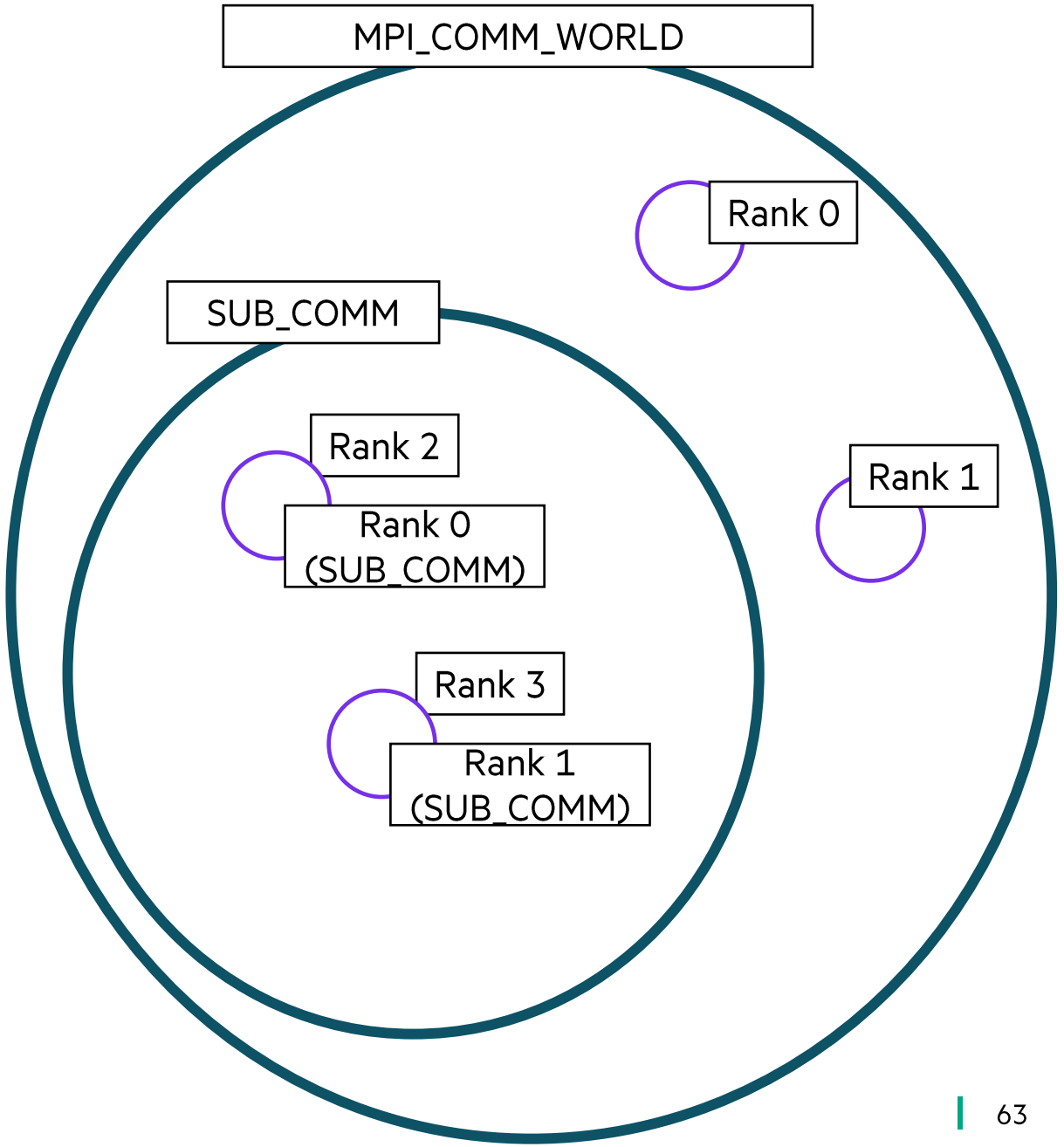
```
#include<stdio.h>
#include<mpi.h>

int main(int argc, char **argv ) {
    int myrank;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

    printf("Hello, from rank %d\n", myrank);

    MPI_Finalize();
}
```

- 実行例

```
$ mpiicx ./hello.c
$ mpirun -np 4 ./a.out
Hello, from rank 1
Hello, from rank 0
Hello, from rank 2
Hello, from rank 3
```

MPI の用語

- コミュニケータ
  - プロセスのグループ
  - MPI_COMM_WORLD ：全プロセスを含むコミュニケータ。MPI の初期化時に定義され、常に利用が可能です。
  - プログラム内でも必要に応じて定義可能です。
- ランク
  - コミュニケータ内でのプロセスの識別番号。0 から始まる整数



ランクとコミュニケータの組み合わせでプロセスを指定することができます。

MPI の関数

- システム関数
  - MPI を利用するために必要な、初期化、終了処理などの関数です。

- 1 対 1 通信関数
  - あるプロセスからあるプロセスへデータを移動するための関数です。MPI の通信の基本は 1 対 1 通信で、高度な通信関数も 1 対 1 通信関数を用いることで実装できます。
  - ブロッキング通信、ノンブロッキング通信に分けられます。

- 集団通信
  - あるグループに属する全プロセスがかかわる通信のための関数です。一つのプロセスから全プロセスへデータを送る MPI_BCAST や、全プロセスのデータを一つのプロセスに集める MPI_GATHER, 全プロセスのデータを加算するなどの処理を行う MPI_REDUCE などがあります。

MPI 関数: システム関数

- MPI_INIT
  - MPI を初期化します。
  - コミュニケータ MPI_COMM_WORLD が定義されます。

```
int MPI_Init(int *argc,char ***argv)
```

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

- MPI_FINALIZE
  - MPI を終了します。

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

MPI 関数: システム関数

- MPI_COMM_RANK(comm, rank)
  - comm: コミュニケータ
  - rank: comm 内のランク
  - コミュニケータ内での自身のランクを取得します。

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

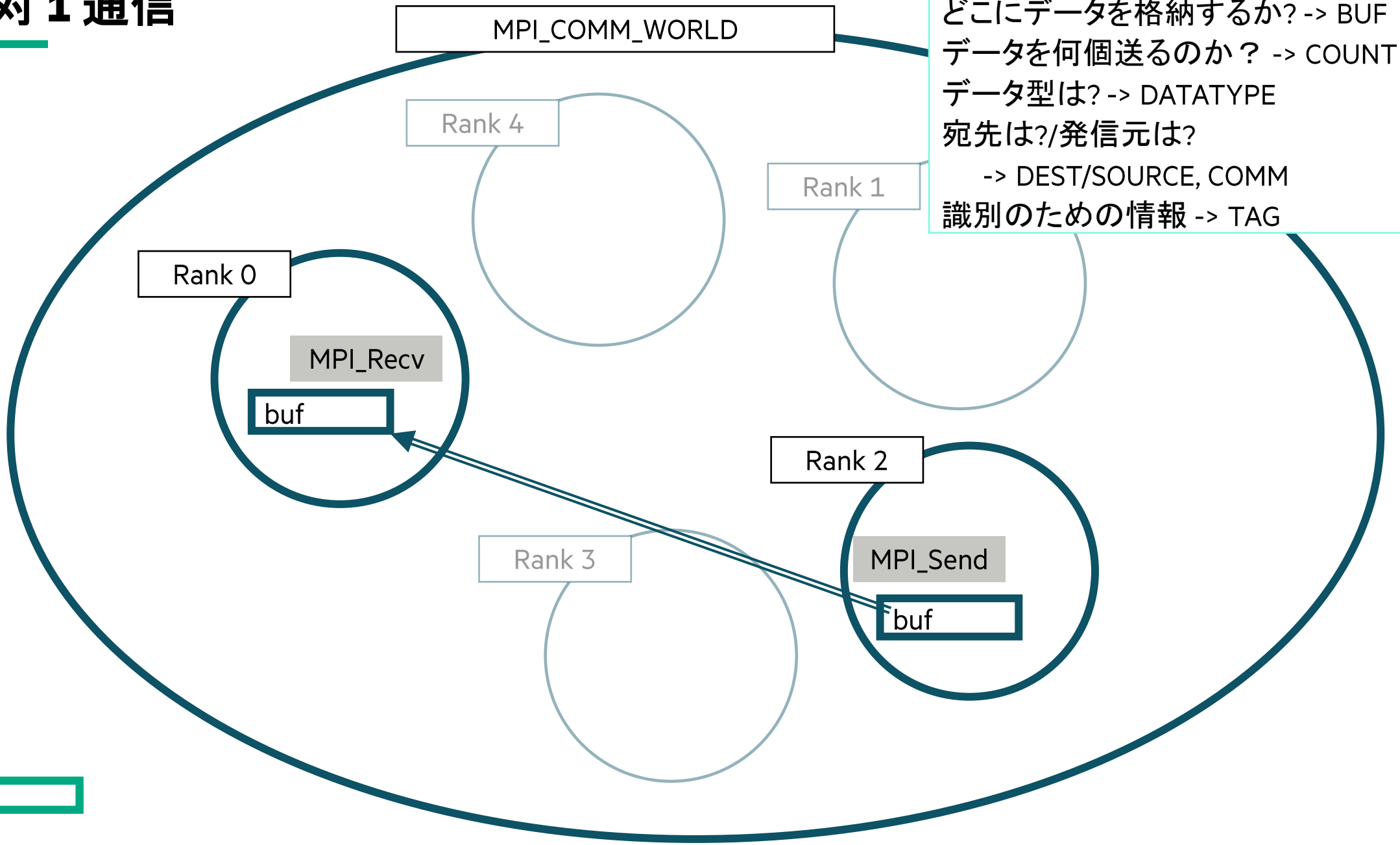
```
MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR
```

- MPI_COMM_SIZE(comm, size)
  - comm: コミュニケータ
  - size: コミュニケータ内のプロセス数
  - コミュニケータに含まれるプロセス数を取得します。

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR
```

1 対 1 通信



どのデータを送るのか? -> BUF
どこにデータを格納するか? -> BUF
データを何個送るのか？ -> COUNT
データ型は? -> DATATYPE
宛先は?/発信元は?
-> DEST/SOURCE, COMM
識別のための情報 -> TAG

MPI 関数: 1 対 1 通信関数

- MPI_SEND(buf, count, datatype, dest, tag, comm)
  MPI_RECV(buf, count, datatype, source, tag, comm)
  - buf: データの先頭アドレス
  - count: データの個数
  - datatype: データ型
  - dest: 送り先の rank
  - source: 送信元の rank
  - tag: タグ。送信と受信で一致させる
  - comm: コミュニケータ

```
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest,
int tag, MPI_Comm comm)
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG,
COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, &
    STATUS(MPI_STATUS_SIZE), IERROR
```

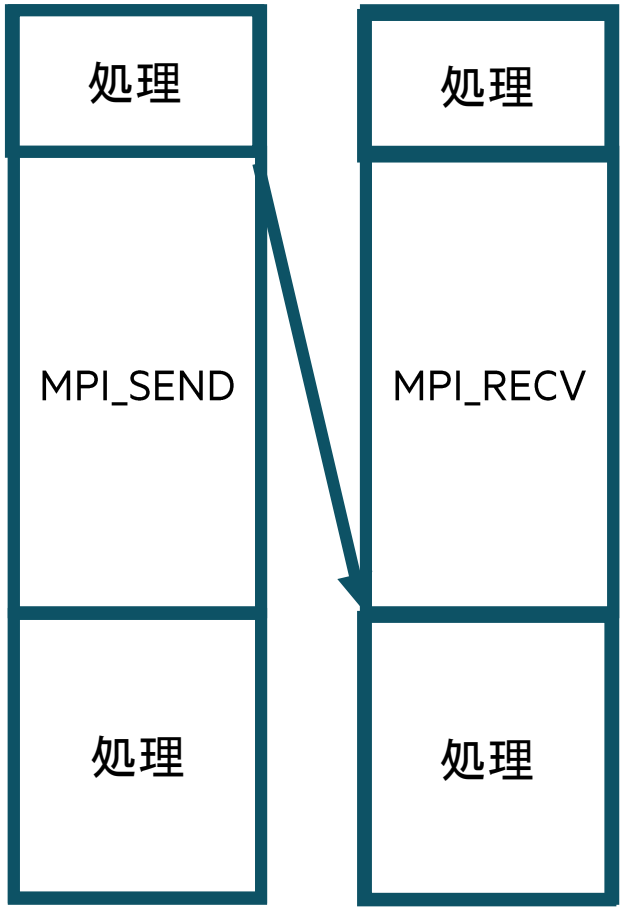
MPI 関数: 1 対 1 通信関数　プログラム例

```
#include<stdio.h>
#include"mpi.h"

int main(int argc, char **argv){
    int size, rank, buf, tag, root, src, i_src;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    root=0;
    tag=1;
    if( rank == 0) {
        for(src=1; src<size; src++) {
            MPI_Recv(&buf, 1, MPI_INT, src, tag, MPI_COMM_WORLD, &status);
            printf("Hello from %d\n", buf);
        }
    } else {
        buf=rank;
        MPI_Send(&buf, 1, MPI_INT, root, tag, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

```
$ mpiexec.hydra -np 4 ./a.out
Hello from rank 1
Hello from rank 2
Hello from rank 3
```

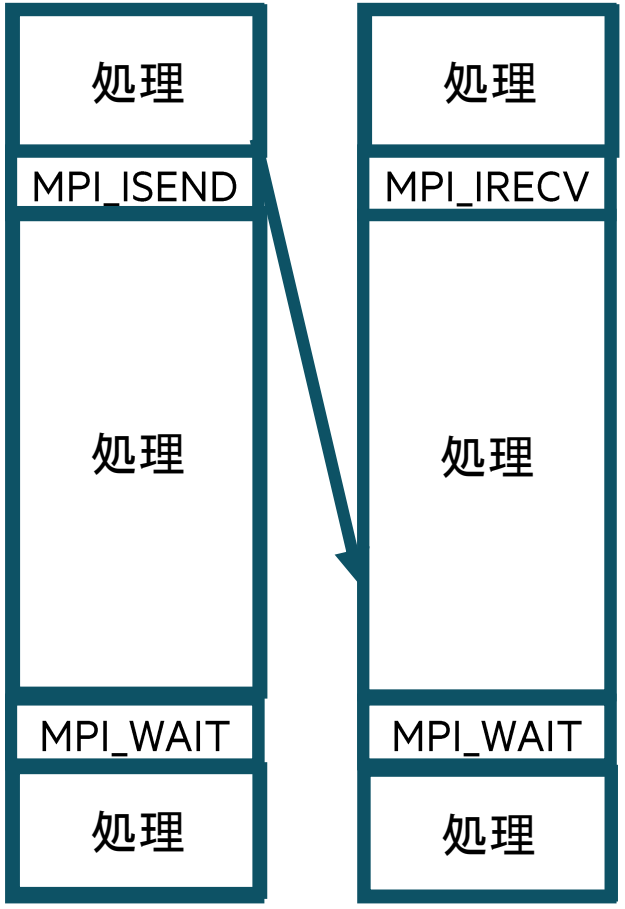
ブロッキング通信とノンブロッキング通信

- ブロッキング通信



ブロッキング通信では、通信が終了するまで他の処理を行うことはできません。

- ノンブロッキング通信



ノンブロッキング通信では、関数の呼び出しからすぐに処理が戻り、通信中も他の処理を実行することができます。MPI_WAIT で通信が終了するのを待ち、その後は通信が終了したことが保証されます。

ブロッキング通信とノンブロッキング通信

- MPI_ISEND(buf,count,datatype, dest, tag, comm, request)
  MPI_IRECV(buf,count,datatype, source, tag, comm, request)
  - buf: データの先頭アドレス
  - count: データの個数
  - datatype: データ型
  - dest/source 送り先/送信元の rank
  - tag: タグ。送信と受信で一致させる
  - comm: コミュニケータ
  - request: 送信/受信命令につけられた識別子

```
int MPI_Isend(const void *buf, int count,
   MPI_Datatype datatype, int dest,
   int tag, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count,
   MPI_Datatype datatype, int source,
   int tag,MPI_Comm comm, MPI_Request *request)
```

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, &
   COMM, REQUEST, IERROR)
   <type> BUF(*)
   INTEGER COUNT, DATATYPE, DEST, TAG, COMM, &
       REQUEST, IERROR
```

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG,  &
   COMM, REQUEST, IERROR)
   <type> BUF(*)
   INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,  &
       REQUEST, IERROR
```

ブロッキング通信とノンブロッキング通信

- MPI_WAIT(request, status)
  - request:送信/受信命令につけられた識別子
  - status: 完了した通信についての情報
  - 非同期通信の完了を待ちます

- MPI_WAITALL(cout, request_array, status)
  - count: 待つリクエストの個数
  - request_array:
    送信/受信命令につけられた識別子の配列
  - status: 完了した通信についての情報の配列
  - request_array で与えられたすべての
    非同期通信の完了を待ちます

```
int MPI_Wait(
      MPI_Request *request, MPI_Status *status)
```

```
MPI_WAIT(REQUEST, STATUS, IERROR)
   INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

```
int MPI_Waitall(int count,
    MPI_Request array_of_requests[],
    MPI_Status array_of_statuses[])
```

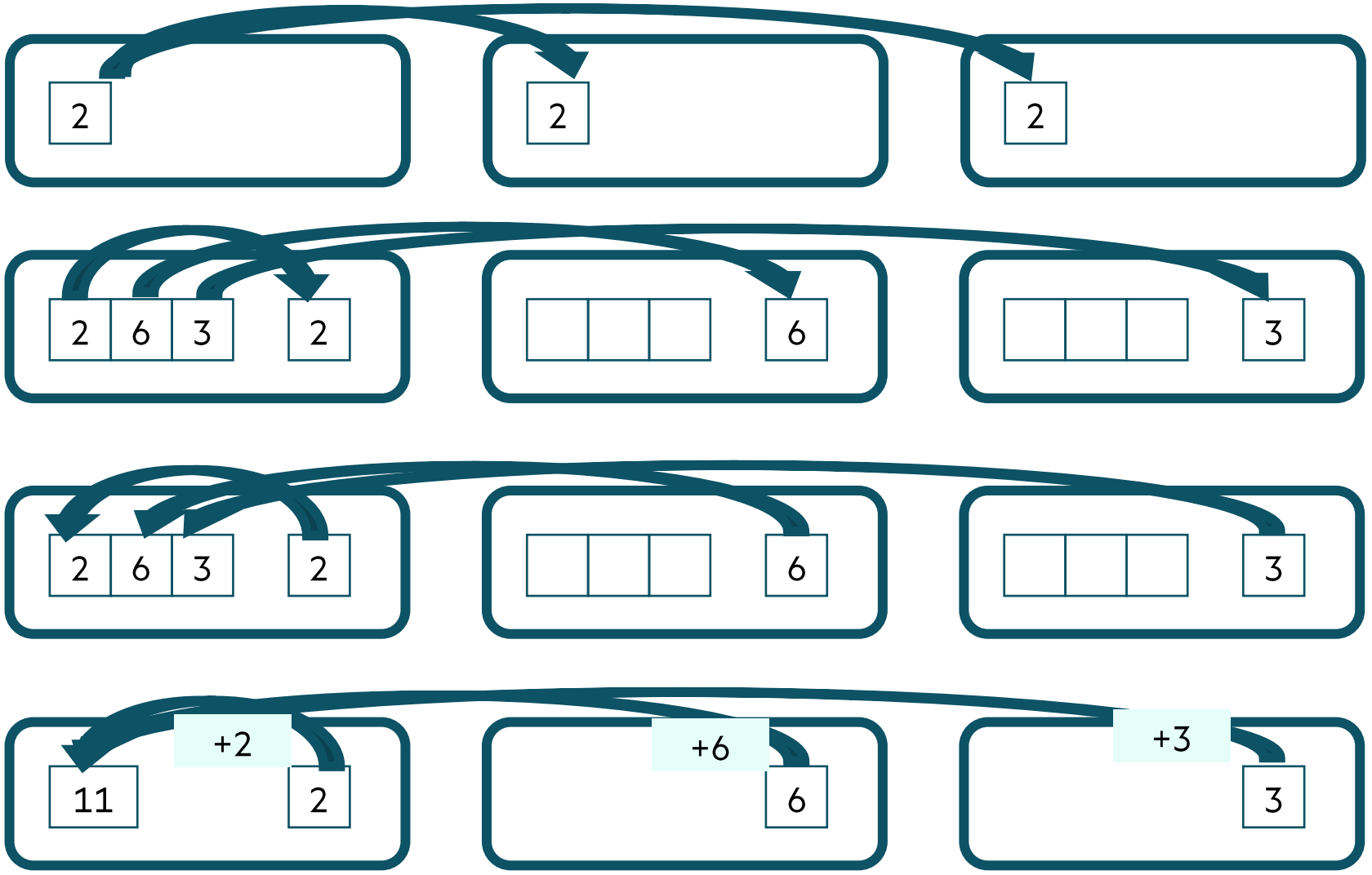
```
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS,
   ARRAY_OF_STATUSES, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), &
   ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), &
   IERROR
```

ノンブロッキング通信

```
    for(i=0; i<N; i++){
        a[i]=...  // a[] を使った処理
    }

    for(dest=0; dest<N_dest; dest++) {
        MPI_Isend(a データの送信)
    }
    for(src=0; src<N_src; src++) {
        MPI_Irecv(a データの受信)
    }

    for(i=0; i<N; i++){
        b[i]=...  //通信の終了を待つ必要のない処理
    }

    MPI_Waitall(すべてのノンブロッキング通信)

    for(i=0; i<N; i++){
        a[i]=...  // 受信したデータを使う処理
    }
}
```

MPI 関数: 集団通信

- MPI_BCAST
  - 全プロセスにコピー
- MPI_SCATTER
  - 全プロセスに分散
- MPI_GATHER
  - 全プロセスから収集
- MPI_REDUCE
  - 全プロセスの値を集約



集団通信例: MPI_Reduce

- MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
  - sendbuf: 送信バッファのアドレス
  - recvbuf: 受信バッファのアドレス
  - count: データの個数
  - datatype: データ型
  - op: 演算の種類
  - root: 結果を受け取るプロセス
  - comm: コミュニケータ
- コミュニケータ内の全プロセスからデータを受け取り、op に与えられる演算に従って集計した結果を root の sendbuf に格納します。
- コミュニケータ内のすべてのプロセスが同じ関数を呼ぶ必要があります。

主な op の値

op	内容
MPI_MAX	最大値
MPI_MIN	最小値
MPI_SUM	合計
MPI_PROD	積
MPI_LAND	論理積
MPI_LOR	論理和

MPI_REDUCE

- プログラム例

```
#include<stdio.h>
#include "mpi.h"

int main(int argc, char **argv){
  double a[12]={3.,8.,12.,5.,6.,4.,9.,11.,2.,7.,10.,1.};
  double s, result;
  int i, myrank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  s=0.;
  for (i=myrank*3; i<(myrank+1)*3; i++){
    s+=a[i];
  }
  MPI_Reduce(&s, &result, 1, MPI_DOUBLE,
      MPI_SUM, 0, MPI_COMM_WORLD);
  if (myrank == 0) printf("result= %lf\n",result);
  MPI_Finalize();
  return 0;
}
```

- 実行例

```
$ mpiicx ./reduction_mpi.c
$ mpirun -np 4 ./a.out
result= 78.000000
```

ハイブリッド MPI/OpenMP 並列化

- MPI, OpenMP の組み合わせ
  - MPI: ノード間、ノード内いずれの並列化にも利用可能
  - OpenMP: ノード内の並列化にのみ対応

- 並列化の方針
  - フラット MPI:
    - ノード内、ノード間いずれの並列化も MPI で記述
  - ハイブリッド MPI/OpenMPI:
    - ノード間は MPI で記述、ノード内は OpenMP, もしくは OpenMP と MPI の組み合わせで記述

例: 4 ノード、各ノード 4 コアのシステム

	MPI プロセス数	OpenMP スレッド数	
フラット	16	-	
ハイブリッド	8	2	各ノードに 2 MPI プロセス
	4	4	ノード内は OpenMP, ノード間は MPI

ハイブリッド MPI/OpenMP 並列化　プログラム例

```
#include<stdio.h>
#include<mpi.h>
#include<omp.h>

int main(int argc, char **argv){
    int rank, nprocs;
    int threadid, nthreads;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    nthreads=omp_get_num_threads();
#pragma omp parallel private(threadid)
    {
        threadid=omp_get_thread_num();
        printf("Hello thread %d of rank %d\n",threadid,rank);
    }

    MPI_Finalize();

}
```

```
$ mpiicx –qopenmp ./hybrid.c
$ qrsh <qrsh のオプション>
$ export OMP_NUM_THREADS=2
$ mpiexec.hydra -np 2 ./a.out |sort
Hello thread 0 of rank 0
Hello thread 0 of rank 1
Hello thread 1 of rank 0
Hello thread 1 of rank 1
```

参考資料

- 片桐　孝洋 「スパコンプログラミング入門: 並列処理とMPI の学習」
  東京大学出版会, 2013

- Peter S. Pacheco 著, 秋葉博訳 「MPI 並列プログラミング」
  培風館, 2001

- Message Passing Interface Forum, https://www.mpi-forum.org/

- Intel® MPI Library Developer Guide for Linux* OS
  https://www.intel.com/content/www/us/en/docs/mpi-library/developer-guide-linux/2021-11/overview.html

- Intel® MPI Library Developer Reference for Linux* OS
  https://www.intel.com/content/www/us/en/docs/mpi-library/developer-reference-linux/2021-11/overview.html

Linaro Forge

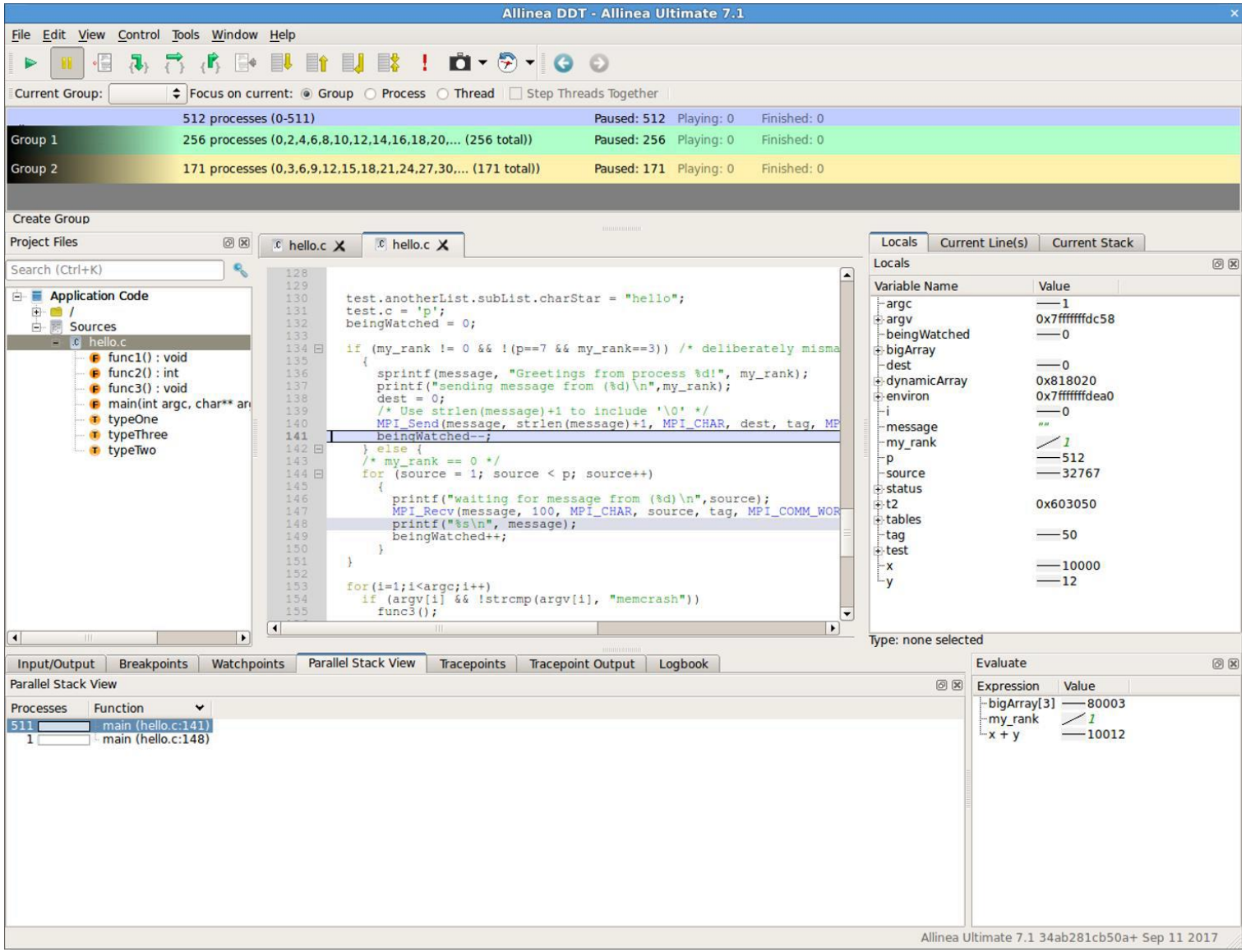
Linaro Forge

- Linaro DDT
  - マルチスレッド、並列アプリケーションに対応した C/C++, Fortran デバッガー

- Linaro MAP
  - ハイパフォーマンスなマルチスレッド/マルチプロセス向けプロファイラ

Linaro DDT

- プロセス制御の状況
- プロセスグループ
- ファイルや関数の検索
- ファイルと関数の一覧
- ソースコードビュー
- プロセス/スレッドの
  変数やスタック



Linaro DDT

- 多次元配列の値を視覚化して表示します。

Multi-Dimensional Array Viewer

Array Expression: tables[$i][$j] — Evaluate

Distributed Array Dimensions: None — How do I view distributed arrays? — Cancel

Staggered Array — What does this do? — Align Stack Frames

Auto-update

Range of $i		Range of $j	
From:	0	From:	0
To:	11	To:	11
Display:	Rows	Display:	Columns

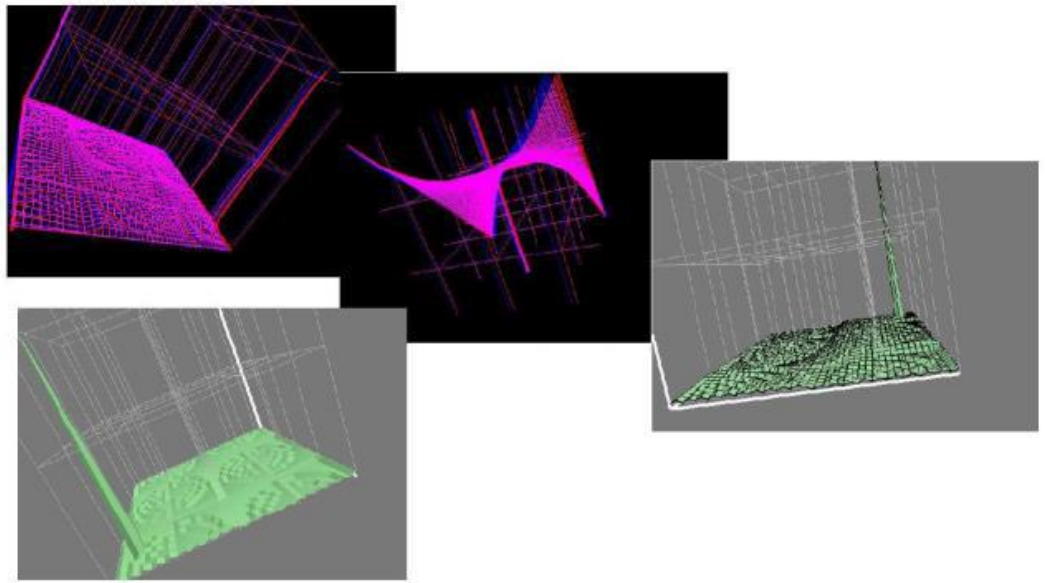
Only show if: — See Examples

Data Table | Statistics

Goto | Visualize | Export | Full Window

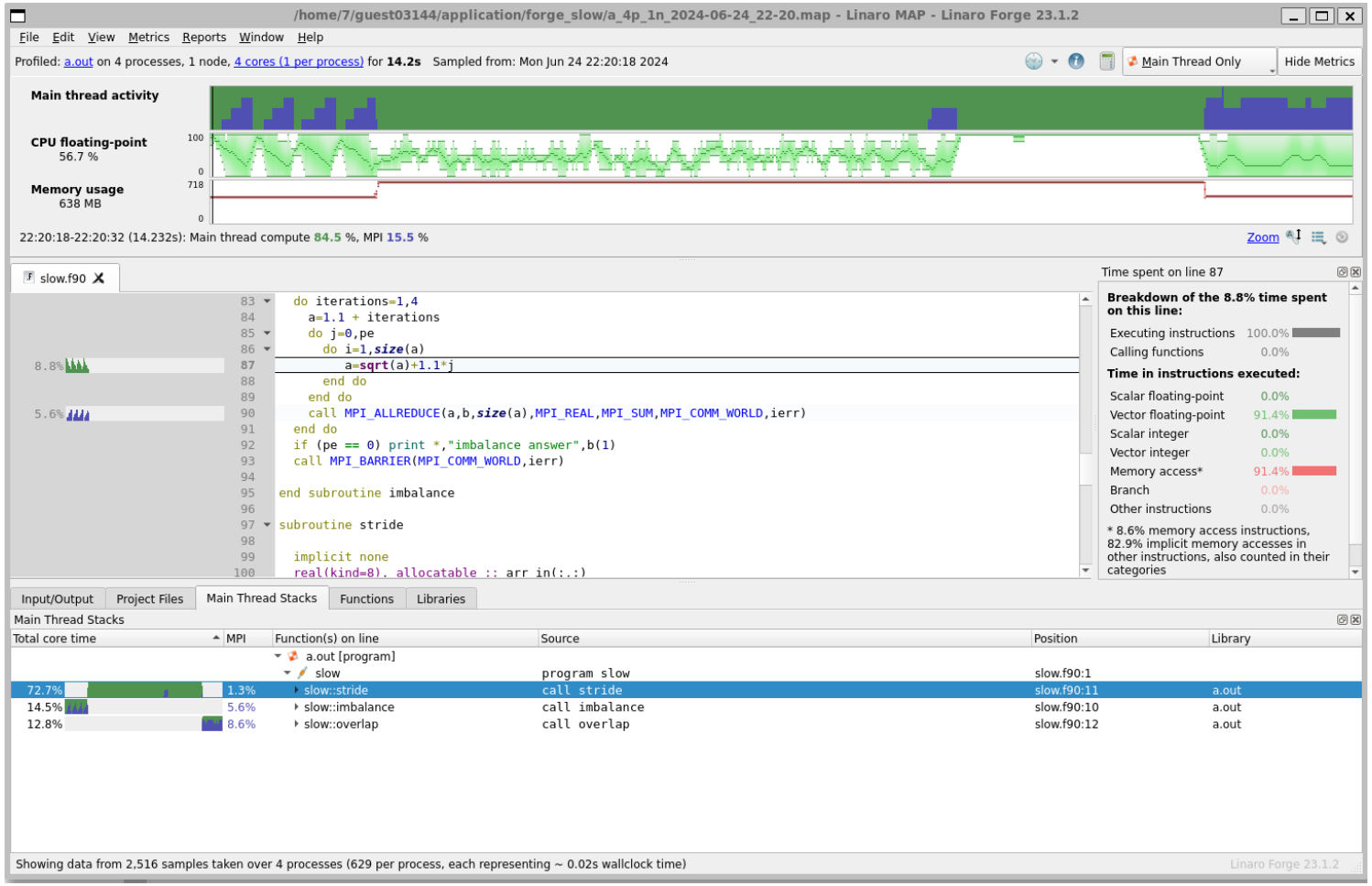
i \ j	0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	6	8	10	12	14	16	18	20	22	24
2	3	6	9	12	15	18	21	24	27	30	33	36
3	4	8	12	16	20	24	28	32	36	40	44	48
4	5	10	15	20	25	30	35	40	45	50	55	60
5	6	12	18	24	30	36	42	48	54	60	66	72
6	7	14	21	28	35	42	49	56	63	70	77	84
7	8	16	24	32	40	48	56	64	72	80	88	96
8	9	18	27	36	45	54	63	72	81	90	99	108

Help — Close



Linaro MAP

- クラスタ向けの性能解析ツールで、MPI, マルチスレッド環境に対して様々な機能を提供します。
- 実行に掛かった時間をソースコードレベルで確認、どの処理がボトルネックとなっているの確認できます。



Linaro Forge 利用例

- X 転送を有効にしてログインノードにログイン
- -g などのデバッグオプションを付けてビルド

```
module load intel intel-mpi
mpiifx –g –O3 slow.f90
```

- インタラクティブジョブを投入。

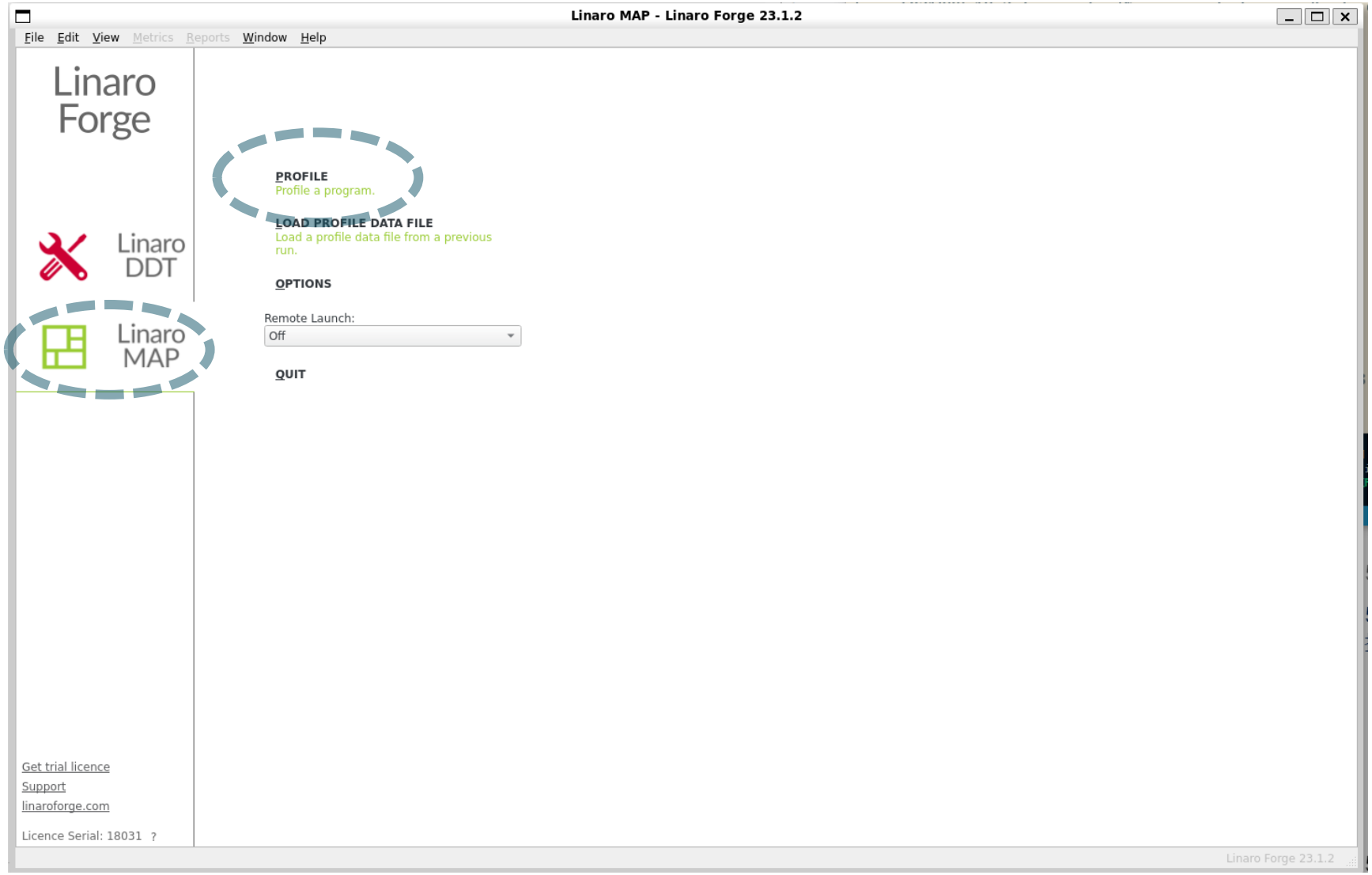
```
$ qrsh <qrsh のオプション>
$ cd <work directory>
```

- forge その他のモジュールのロードと forge の起動

```
$ module load intel intel-mpi forge
$ forge
```

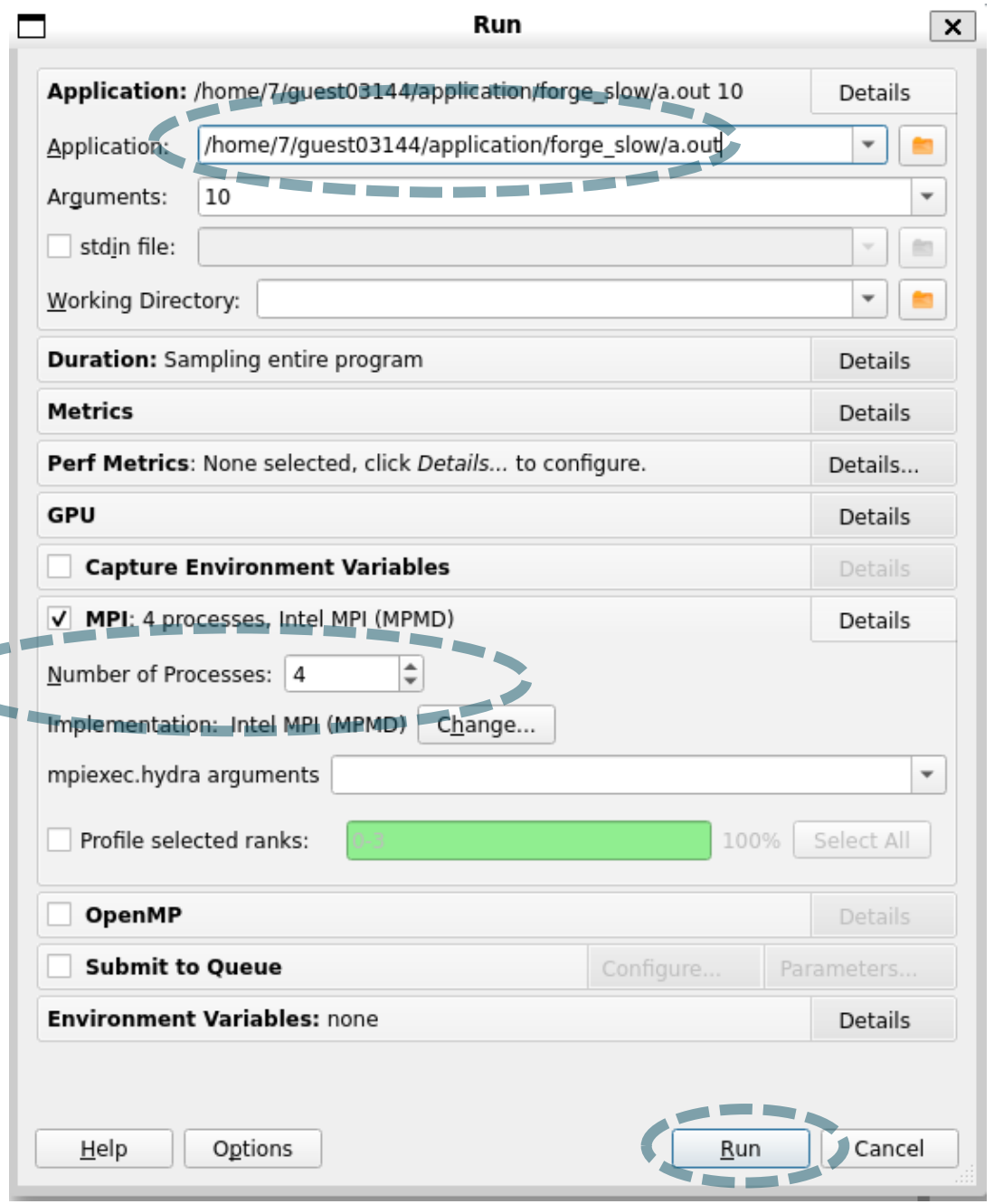
Linaro Forge 利用例

- Linaro MAP を選択
- Profile を選択

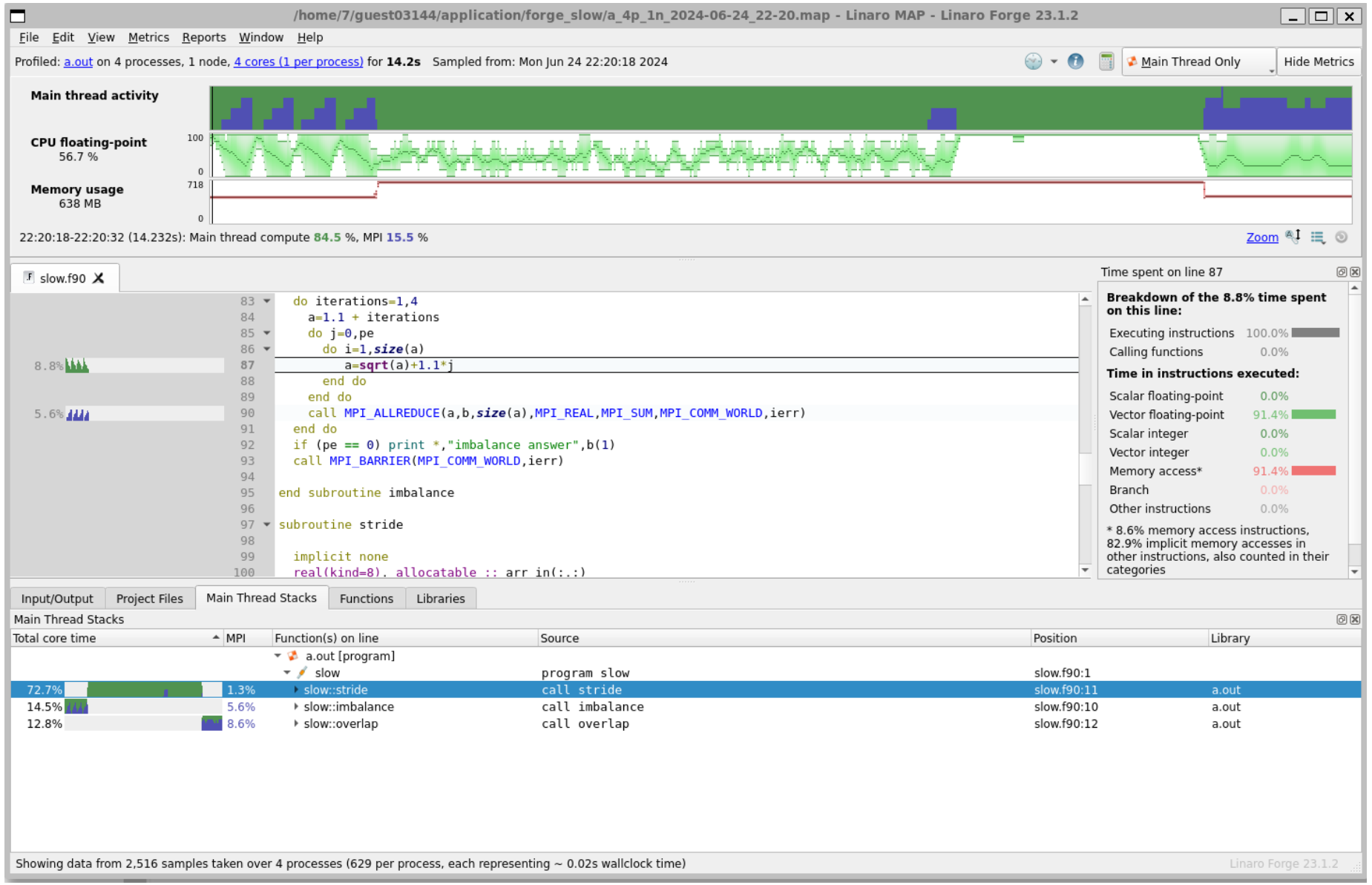


Linaro Forge 利用例

- バイナリの選択
- MPI のオプションの設定
- 実行



Linaro Forge 利用例



参考文献

- Linaro Forge User Guide
  /apps/t4/rhel9/isv/forge/23.1.2/doc/userguide-forge.pdf

Hands On