並列計算とは/OpenMP の初歩（１）

臼井 英之

システム情報学研究科　　計算科学専攻

2018年 5月 17日

本資料は2015年度担当の谷口先生の資料を
参考にさせていただいています。

今日の内容

- 並列計算とは
- 並列計算機の分類・この演習で使う計算機
- OpenMP による並列化
  - Hello World の並列化と並列計算機上での実行方法
  - Do ループの並列化 (omp do)
  - Fortran ならではの記法の並列化 (omp workshare)

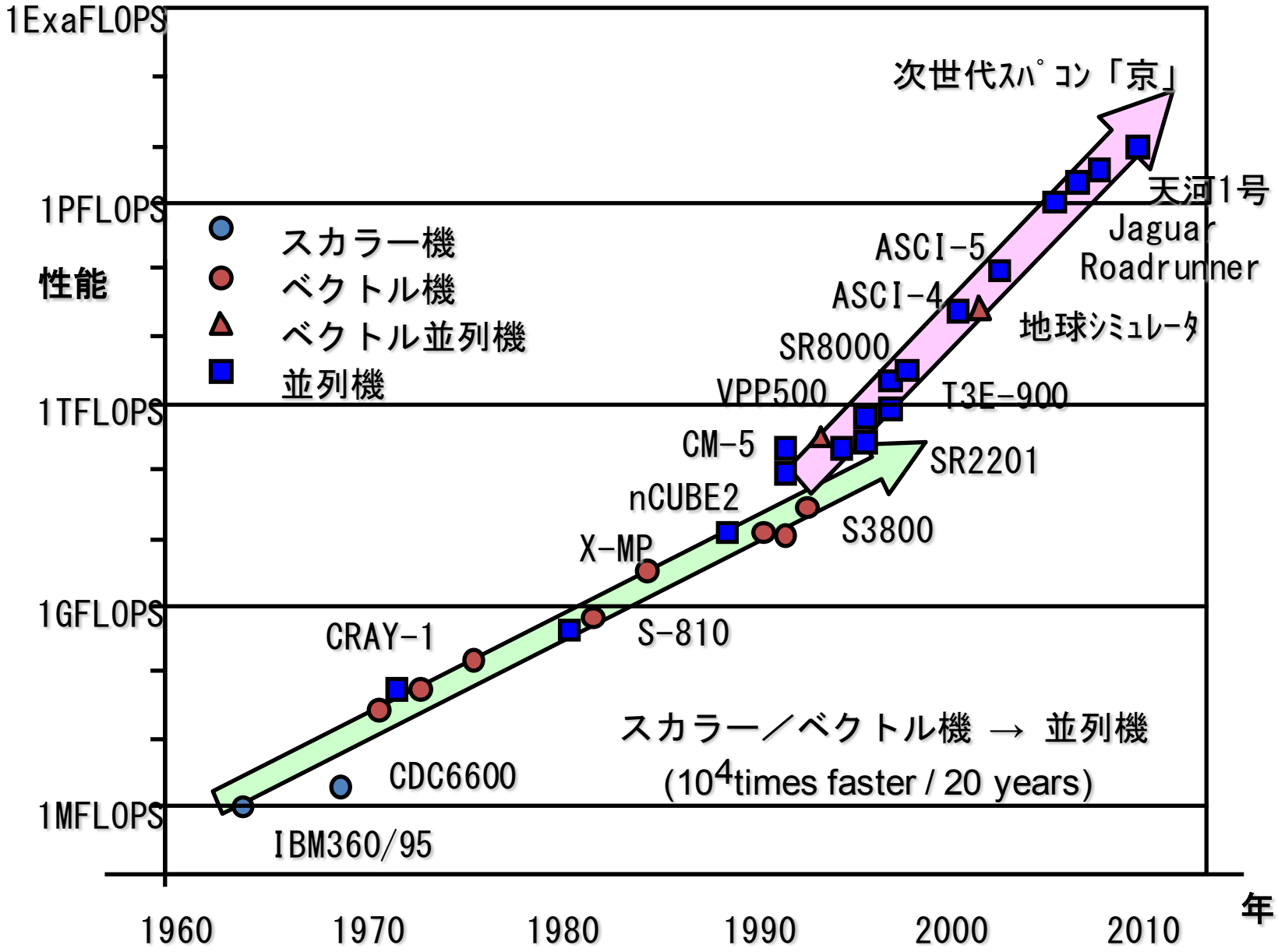
並列計算とは

なぜ並列計算が必要か？

- 本格的な科学技術計算は計算時間が膨大！
- 単一 CPU の速度向上の限界
  - 制作技術の限界
  - 信号伝搬速度の限界
  - 電力性能比向上の限界

  など
- 「優秀な一人の作業員」→「数万人の普通の作業員」
  - 全員に効率よく働いてもらうのは難しい！
  - 「仕事の配分」や「作業員間の連絡」を工夫.

スーパーコンピュータの性能動向



TOP500 によるスーパーコンピュータ性能比較

- LINPACK ベンチマークにより世界のスーパーコンピュータ上位 500 システムを順位付け（http://top500.org）

  LINPACK ベンチマーク:
  LU 分解により連立一次方程式の解を求めるベンチマークプログラム.

- 毎年２回，国際会議で TOP500 リストを発表
  - International Supercomputing Conference（ISC）：ドイツ開催
  - The International Conference for High Performance Computing, Networking, Storage and Analysis（SC）：米国開催
- 1993 年に開始．過去のリストも上記のウェブページに掲載.

2017年 11 月のリスト（上位5システム）

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	**Gyoukou** - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan	19,860,000	19,135.8	28,192.0	1,350
5	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209

頑張って勉強しましょう！

並列計算は必須！

- 科学技術計算（=大規模計算）を行うために
- マルチコアプロセッサの一般化（=パソコンでも並列計算.）

並列計算には工夫が必要！

- 「優秀な一人の作業員」→「数万人の普通の作業員」
  - 全員に効率よく働いてもらうのは難しい！
  - 「仕事の配分」や「作業員間の連絡」に工夫が必要.
- どのアルゴリズムが良いのか？
  - 計算量の小さいアルゴリズム ↔ 並列性のあるアルゴリズム
  - 評価指標（計算速度，並列化効率）

並列計算機の分類・この演習で使う計算機

並列計算機のアーキテクチャ

- 並列計算機

  複数のプロセッサ（ノイマン・アーキテクチャ）が，何らかの方法で接続されいて，協調して動作する計算機システム

- 分類方法
  - SIMD と MIMD（Flynn の分類, 1966 年）

    命令実行の流れとデータの流れに着目した分類方法
  - 共有メモリ型と分散メモリ型　メモリ空間

    に着目した分類方法
  - ホモジニアス型とヘテロジニアス型　ハードウェアの

    均質性に着目した分類方法

SIMD と MIMD

SIMD（Single Instruction Stream, Multiple Data Stream）型

全プロセッサが，それぞれ異なるデータに対し，同じ命令を実行.

例：初期（〜1990 年頃）の商用計算機，グラフィックス・プロセッサ.

- 命令実行の制御回路が 1 つで済むので，プロセッサ数の増加が容易.
- 命令実行に柔軟性がないため，用途が限られる.

MIMD（Multiple Instruction Stream, Multiple Data Stream）型

各プロセッサが，それぞれ異なるデータに対し，異なる命令を実行.

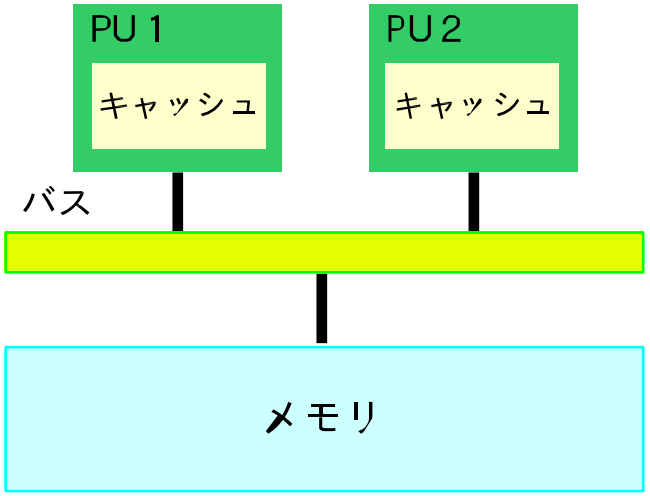
例：最近の商用並列計算機（π-computer），マルチコアプロセッサ.

- 各プロセッサに制御回路が必要.
- 命令実行の柔軟性は非常に高い.

共有メモリ型と分散メモリ型

共有メモリ型並列計算機

複数のプロセッサがバスを通してメモリを共有
→ どのプロセッサも同じメモリ領域にアクセス可能



特徴

- メモリ空間が単一のためプログラミングが容易
- プロセッサ数が多すぎると，アクセス競合により性能が低下
  → 2〜16 台程度の並列が一般的

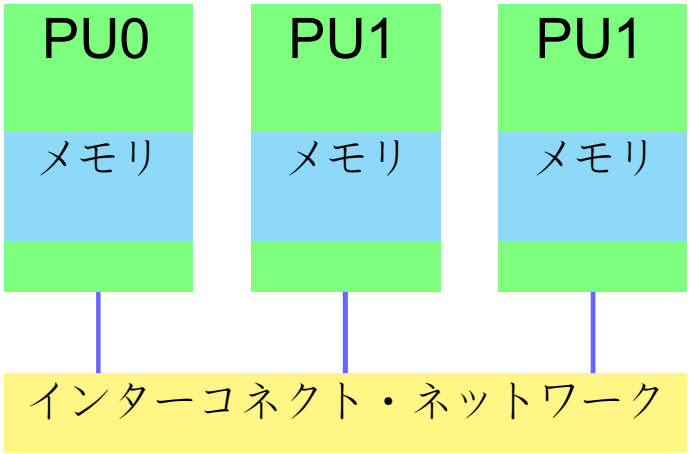
プログラミング言語

- OpenMP（FORTRAN/C/C++ ＋指示文）を使用
- MPI を利用することも可能

共有メモリ型と分散メモリ型（続き）

分散メモリ型並列計算機

各々がメモリを持つ複数のプロセッサをネットワークで接続
→ 各プロセッサはそれぞれ自分の持つメモリのみにアクセス可能



特徴

- 数千～数万プロセッサ規模の並列が可能
- プロセッサ間へのデータ分散を意識したプログラミングが必要

プログラミング言語

- FORTRAN/C/C++ ＋ MPI を使用

ホモジニアス型とヘテロジニアス型

ホモジニアス型の並列計算機

- 全ノードが同じ構成を持つシステム.
- すべてのタスクが均質であるような並列プログラム向き.
  例：SMP クラスタなど

ヘテロジニアス型の並列計算機

- 異なるアーキテクチャ，異なる性能のノードで構成されるシステム.
- 特定の計算部分を高速に実行するプロセッサを付加する場合が多い.
  例：グラフィックス専用プロセッサ（GPGPU），MD-GRAPE など

計算科学演習 I で使う並列計算機

π コンピュータ

演習用端末
Fujitsu ESPRIMO
K552/D

六甲台キャンパス

学内
LAN

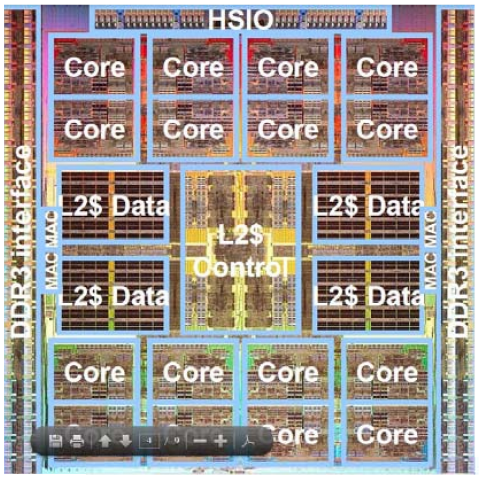
ログインサーバ
Fujitsu Primergy RX300 S6 π-computer
Xeon E5645@2.4GHz,
6 コア × 2sockets
メモリ 94GB

Fujitsu PRIMEHPC FX10
96 ノード，ノードあたり CPU:
SPARC64 IXfx@1.65GHz,
16 コア，211.2GFLOPS
メモリ：32GB/ノード

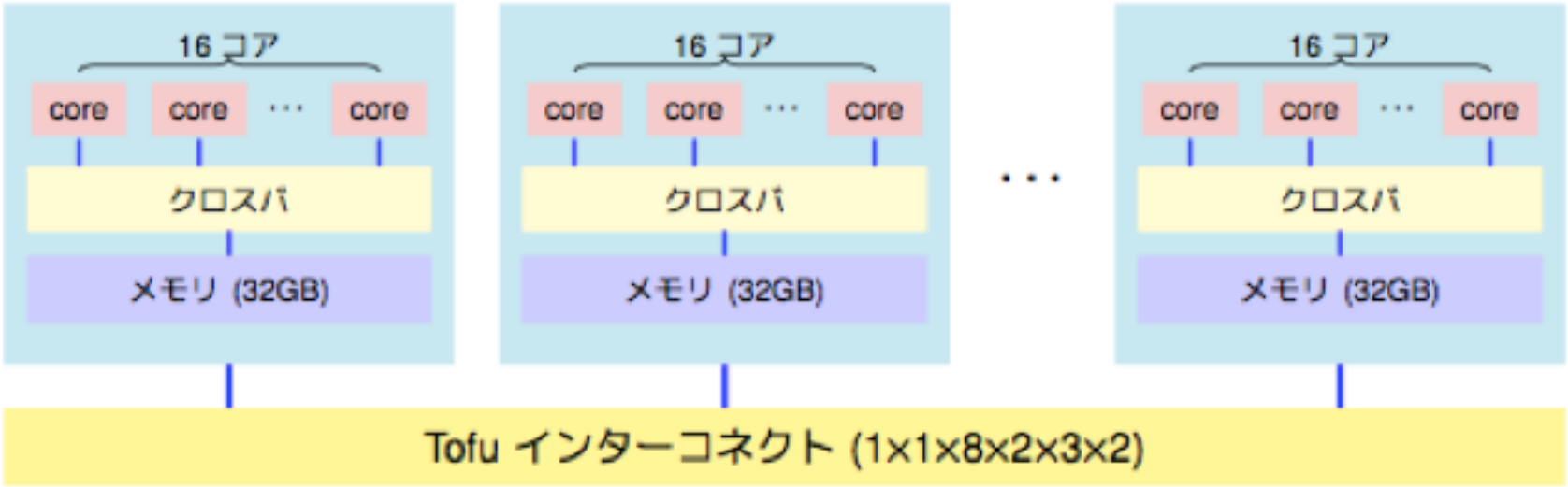
神戸大学統合研究拠点

πコンピュータ

SPARC64 IXfx（SPARC64 V9 + HPC-ACE）



- コア数：16 コア　動作周波数：1.65GHz
- キャッシュ L1-I,　L1-D：32KB/コア,
- L2：12MB/ソケット
- メモリバンド幅：85GB/s
- 40nm CMOS, 21.9 mm × 21.9 mm

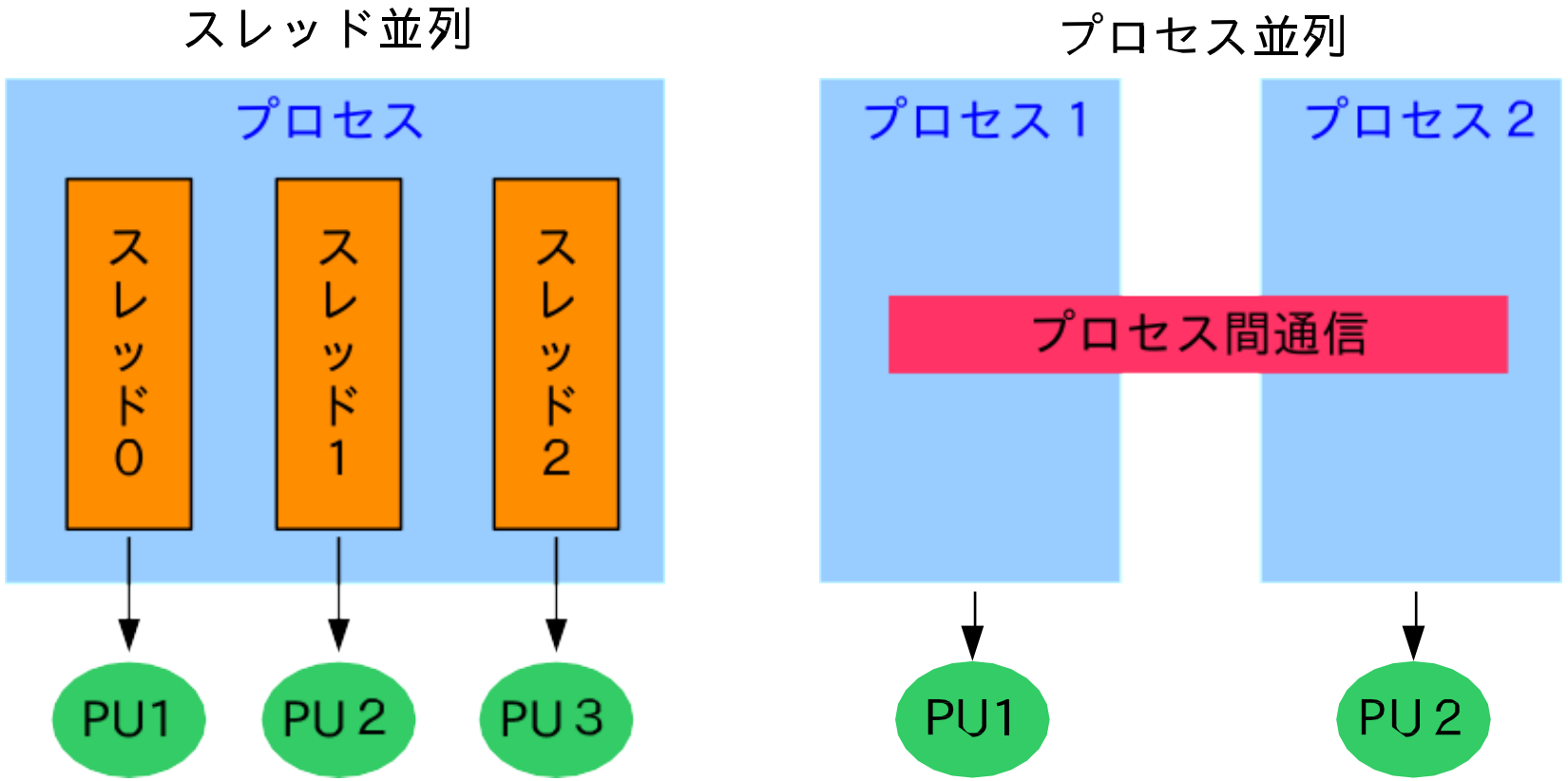


OpenMP による並列化

共有メモリ型並列計算機における並列方式

【スレッド並列】

- スレッド：プロセス内の処理実行の流れ
- 同一プロセス内の各スレッドは同じメモリ空間にアクセス
- OpenMP によるプログラミングが標準的

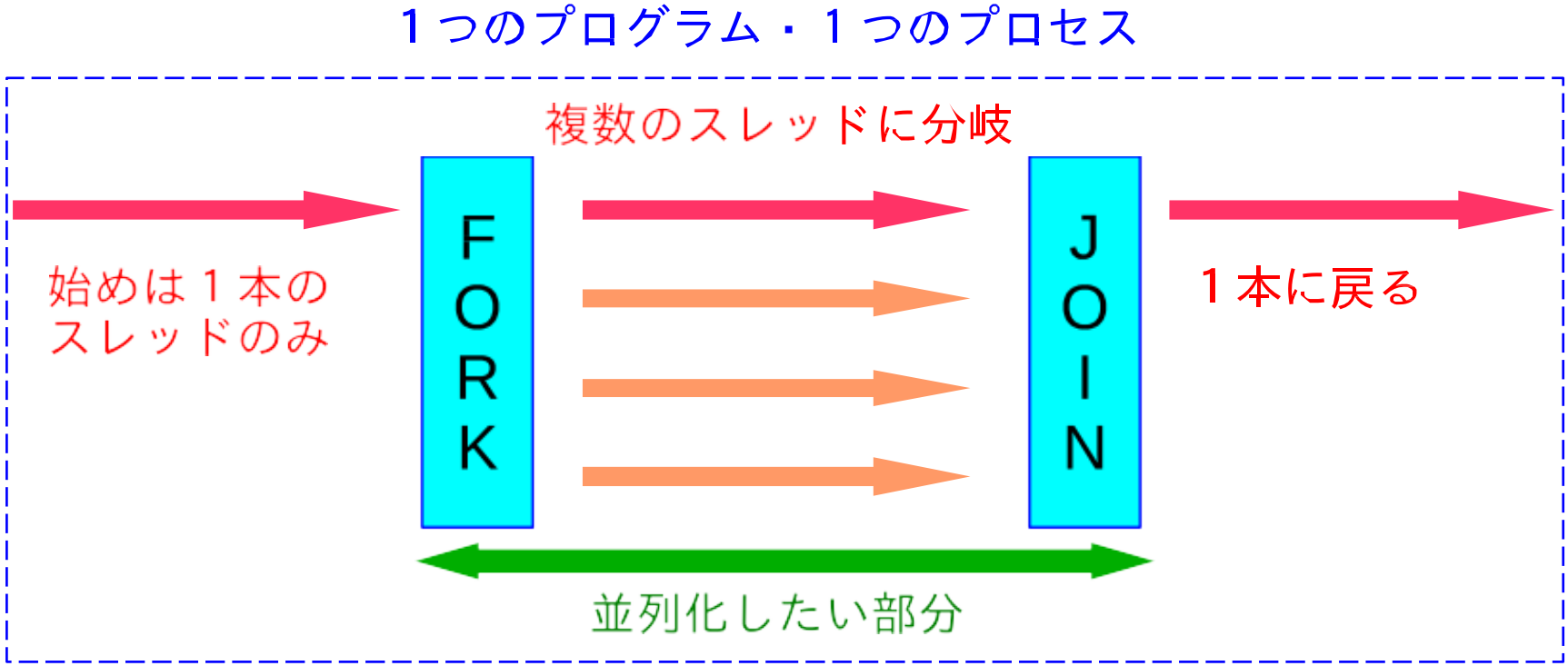


OpenMP

- 共有メモリ型並列計算機向け並列計算ライブラリ
  - 指示行を挿入するだけで並列化が可能
    - （始めから並列計算用プログラムを書くのではなく）
      逐次コードを修正していく形でプログラミングが可能
    - 逐次プログラムとしても実行可
    - 比較的，デバッグが簡単
  - 移植性に優れる
    プログラムを修正しなくても，様々な共有メモリ型並列計算機で実行可
  - 解説書が豊富
  - 逐次実行部分が多くなりがち ➡速くなりにくい
- 米国のコンパイラメーカーを中心に仕様を決定
  - 1997 FORTRAN Ver. 1.0 API
  - 1998 C/C++ Ver. 1.0 API
  - 2000 FORTRAN Ver 2.0 API
  - 2002 C/C++ Ver 2.0 API
  - 2005 FORTRAN C/C++ Ver 2.5 API
  - 2008 FORTRAN C/C++ Ver 3.0 API
  - 2013 Ver 4.0 Released! （アクセラレータ機能追加など）

OpenMP の実行モデル：Fork–Join モデル

- １つのスレッド（マススタースレッド）でスタート
- 並列化部分の開始時 ➡複数のスレッドに分岐（Fork）
- 並列化部分の終了時 ➡マススタースレッドのみに戻る（Join）



OpenMP の構成要素

- 元の（FORTRAN/C/C++ で書かれた）プログラム
- 指示文（ディレクティブ）
  - 並列化すべき場所・並列化方法を指定
  - FORTRAN では !$omp で開始
    例）

```
!$omp parallel
```

- ライブラリ関数

  例）並列実行部分でスレッド数を取得する関数：omp_get_num_threads()
- 環境変数
  - 並列実行部分で使うスレッド数などを指定するのに利用

    例）スレッド数を指定する環境変数： OMP_NUM_THREADS.

演習１（準備）：Hello World を並列化してみよう！

まず，逐次版プログラムを用意

- 今日の演習用のディレクトリ（例えば enshu-openmp1）を作成

```
% mkdir enshu-openmp1
% cd enshu-openmp1
```

- emacs 等で，以下のプログラムを作成し，hello.f90 などの名前で保存

```
program hello_world
implicit none
print*, "Hello World!"
end program
```

- frtpx でコンパイル.

```
% frtpx hello.f90
```

- ./a.out では**実行できません**．実行方法はこれから.

注意：計算ノードで計算を行うのでコンパイル・実行方法が変わります

大型計算機上でのプログラム実行：キューイングシステム

スパコンは共有財産！ ➡ ジョブの管理が必要

キューイングシステム

負荷状況・リソース使用量を監視し，ユーザが投入したジョブを適切な計算ノードに割り当て，実行するソフトウェア．

プログラム実行の流れ

1. ジョブスクリプトを作成
2. ジョブを投入
3. （ジョブの状態を確認）
4. 結果を確認

```
% ./a.out
```

で実行するのでは ない

注）自分のパソコンなどで実行する場合は ./a.out でＯＫ．

演習１（続き）ジョブスクリプトの作成

演習で使うキュー

キュー名	最大ノード数	経過時間制限
small	12	10 分

ジョブスクリプトの例（OpenMP 版）

```
#!/bin/bash
#PJM -N ”jobname”
#PJM -L ”rscgrp=small”
#PJM -L ”node=1”
#PJM -L ”elapse=2:00”
#PJM -j
export  OMP_NUM_THREADS=1
./a.out
```

シェルを指定
ジョブ名を指定
投入先のキュー名を指定
使用ノード数を指定

スレッド数を指定
実行プログラム名を指定

【演習】上のスクリプトのジョブ名などを適切に修正し，hello.sh などの名前で保存.

演習１（続き）：ジョブの投入

- ジョブの投入

```
pjsub （ジョブスクリプト名）
```

- ジョブの状態確認

```
pjstat
```

JOB ID	JOB NAME	MD	ST	USER	START DATE	ELAPSE LIM	NODE REQUIRE
17583	jobname	NM	RUN	user	(05/19 16:23)	0000:02:00	1
17584	jobname	NM	QUE	user	(05/19 16:33)	0000:02:00	1

- ジョブのキャンセル

```
pjdel （ジョブ番号）
```

【演習】

- Hello World のジョブを投入してみよう！

```
% pjsub hello .sh
```

[INFO] PJM 0000 pjsub Job 17583 submitted. などと表示
（”17583” の部分がジョブ番号）

- うまくいけば ジョブ名.o? （? はジョブ番号）というファイルが作成され，その中に”Hello World!”が出力されます（**cat** などで確認）.

演習１（これで最後）：OpenMP を用いた Hello World の並列化

- Hollow World プログラムに赤文字部分を追加して（追加するだけで）並列化

```
program hello world
implicit none
integer :: omp_get_thread_num
!$omp parallel
print *, "My id is ", omp_get_thread_num(), " Hello World !"
!$omp end parallel
end program
```

- OpenMP を用いていることを明示してコンパイル

```
% frtpx -Kopenmp hello.f90
```

- ２スレッドで実行：hello.sh の OMP_NUM_THREADS の値を２に書きかえて

```
% pjsub hello.sh
```

プログラムの解説

- 並列リージョン
  - ２つの指示文 !$omp parallel と !$omp end parallel で囲まれた部分を並列リージョンという.
  - 並列リージョン内では（OMP_NUM_THREADS）個のスレッドが同じコードを実行.
  - 各スレッドは固有のスレッド番号をもつ. これを用いて，各スレッドに異なる処理を行わせることができる.
  - スレッド番号は omp_get_thread_num() によって取得できる.

```
program hello
 implicit none
!$omp parallel
   …
   …     並列リージョン
   …
!$omp end parallel
end program
```

- 変数・配列の参照・更新
  - すべてのスレッドが同じ変数・配列を参照できる.
  - 複数のスレッドが同時に同じ変数を更新しないよう，注意が必要.

（同じ配列の，異なる要素を同時に更新するのは OK ）

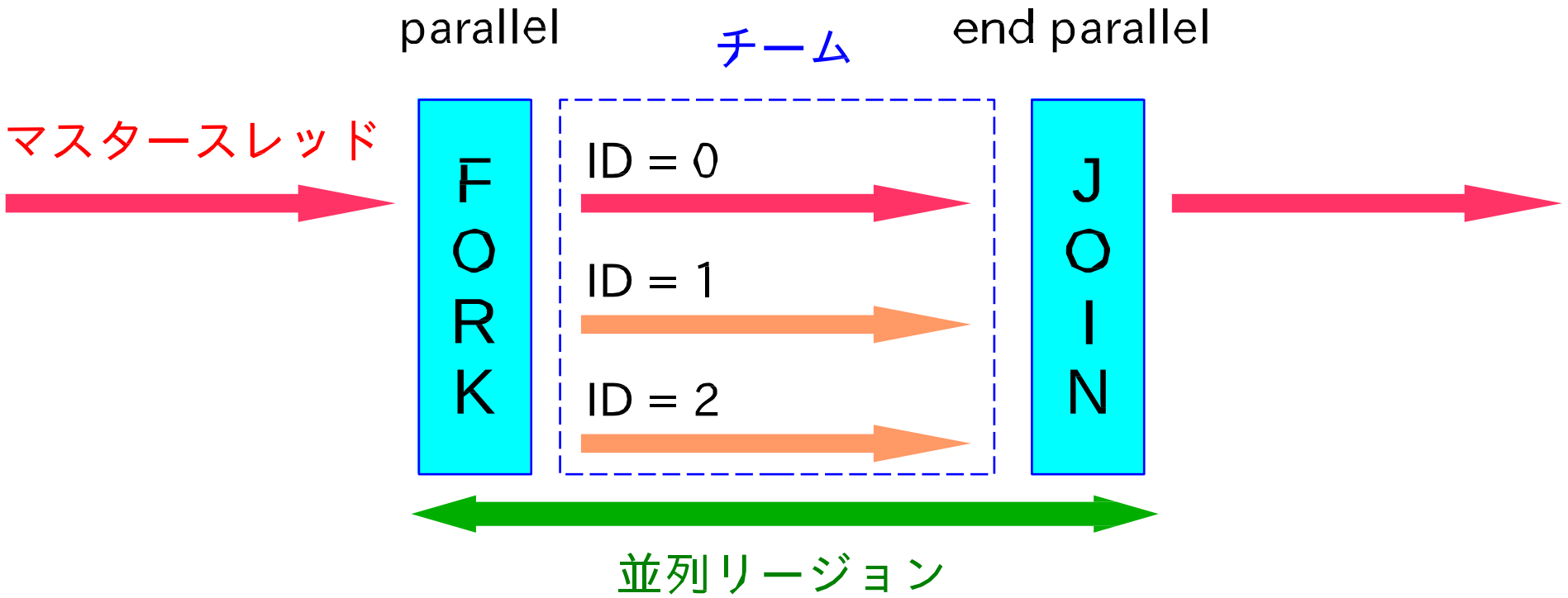
OpenMP 並列化プログラムの基本構成例

```
program main
implicit none

（逐次実行部分）

!$omp parallel

（並列化したい部分）

!$omp end parallel

（逐次実行部分）

end program
```

並列リージョン：　複数のスレッドにより，
並列実行される部分

マルチスレッドでの実行のイメージといくつかの用語

- プログラム実行開始時はマスタースレッドのみ
- PARALLEL 指示文により複数のスレッドを生成
  - スレッド ID：0～OMP_NUM_THREADS－1 に値をもつ，各スレッド　に割り振られる固有の番号.
  - チーム：並列実行を行うスレッドの集まり.
  - スレッド生成後，全てのスレッドで冗長実行
- END PARALLEL 指示文によりマスター以外のスレッドが消滅



DO ループの分割（!$omp do）

```
program main
implicit none
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
real(DP), dimension(100000) :: a, b
integer :: i

!$omp parallel
!$omp do
do i=1,100000
   b(i) = a(i)
end do
!$omp end do
!$omp end parallel

end program
```

直後の DO ループを複数のスレッドで分割して実行せよ，という意味
（!$omp end do は省略可）

２スレッドで実行した場合

スレッド０
```
do i=1,50000
   b(i) = a(i)
end do
```
スレッド１
```
do i=50001,100000
   b(i) = a(i)
end do
```
（分割の仕方はコンパイラ依存）

演習２：omp do を使ってみよう！

課題

- 次のスライドのプログラムを作成.
- スレッド数を 1, 2, 4 と変えてみて経過時間を計測.

【時間計測の方法】omp_get_wtime 関数を利用.

- 倍精度で omp_get_wtime, time0, time1 を定義し,
- 測定したい部分を time0=omp_get_wtime() と time1=omp_get_wtime()ではさむ（今回は !$omp parallel の前と !$omp end parallel の後に挿入）.
- time1- time0 が経過時間（秒単位）.
- 時間計測用のジョブスクリプトは, ２つ後のスライドに掲載してあります.

```
time0=omp_get_wtime ()
!$omp parallel
!  (時間計測する部分)
! $omp end parallel
time1=omp_get_wtime()
 ｐｒｉｎｔ＊, time1−time0
```

演習２のプログラム

```
program axpy
implicit none
integer , parameter : : SP = kind (1 . 0)
integer , parameter : : DP = selected_real_kind (2*precision ( 1 . 0_SP ))
real (DP) , dimension ( 100000 ) : : x , y , z
real (DP) : : a
integer : : i
!
! a, x, y の値を各自で自由に設定.
!
!$omp parallel
!$omp do
do i = 1 , 100000
   z ( i ) = a*x ( i ) + y ( i )        ベクトルの加算 z⃗ = a x⃗ + y⃗
end do
!$omp end do
!$omp end parallel
!
! 経過時間の確認
!
print *, z(1)
end program
```

時間計測用ジョブスクリプトの例

```
#!/bin/bash
#PJM -N "jobname"
#PJM -L "rscgrp=small"
#PJM -L "node=1"
#PJM -L "elapse=2:00"
#PJM -j

export FLIB_CNTL_BARRIER_ERR=FALSE

for opn in 1 2 4
do
export OMP_NUM_THREADS=$opn
./a.out
done
```

opn を変えながら do 内を実行

スレッド数を opn に設定

実行プログラム名を指定

演習３：!$omp parallel do

- do ループの並列化は，parallel と do をまとめて
```
!$omp parallel
!$omp do
do i=1,100000
   b(i) = a(i)
end do
!$omp end do
!$omp end parallel
```
のように書いても良い．

```
!$omp parallel do
do i=1,100000
   b(i) = a(i)
end do
!$omp end parallel do
```

- **!$omp end parallel do** は省略しても良い．

【演習３】 演習２のプログラムを parallel do を使って書き換えてみよ．

― 注意 ―
omp do は並列実行できない場合も自動的に分割してしまう！

```
program invl
implicit none
integer, parameter::n=100
integer, dimension(n)::a
integer :: i

a(1) = 0
!$omp parallel do
do i=2,n
a(i) = a(i-1) + 1
end do
!$omp end paralleldo

print *, a(n)

end program
```

2スレッドで実行

スレッド0
```
do i=1,50
  a(i) = a(i-1) + 1
end do
```
スレッド1
```
do i=51,100
  a(i) = a(i-1) + 1
end do
```

本当は a(50) の結果がないと実行できない！

正しい結果：９９

do ループの並列化のまとめ

- do ループを並列化するには並列化したいループの前に
  !$omp parallel do を置けば良い.
  - ループ変数の動く範囲が OMP_NUM_THREADS 個に分割され,
  - 各ブロックはそれぞれ１スレッドにより実行.
  - 分割のされ方はコンパイラ依存. 同じプログラムの中でも変わり得る.
- ただし, 並列化してよいループかどうかはプログラマが判断.

並列化してはダメなループの例）再帰参照を含むループ

```
do i=2,100
    x(i)=a*x(i-1)+b
end do
```

１つ前に計算した要素の値を使って, 現在の要素を計算.

ただし, 一見ダメそうでも, よく考えれば並列化できる場合も.

omp workshare

演習２のプログラムは次のように書いてもよい：

```
!$omp parallel
!$omp do
do i=1,100000
   z(i) = a*x(i) + y(i)
end do
!$omp end do
!$omp end parallel
```

```
!$omp workshare
   z(:) = a*x(:) + y(:)
!$omp end workshare
```

(!$omp end workshare は省略不可)

- FORTRAN のみで使える書き方.
- コンパイラによっては matmul （行列積）なども並列化してくれる.

```
!$omp workshare
   C = matmul(A, B)
!$omp end workshare
```

のように書くと並列化してくれるコンパイラもある.

宿題

【課題】

- 演習２のプログラムを workshare を用いて書き換え,
- スレッド数を 1, 2, 4 と変えてみて経過時間を計測してみよ.

プログラムと実行結果（スレッド数を 1, 2, 4 としたときの経過時間）とその検討内容を１つのテキストファイル（例えば result.txt）に入れて, その内容を臼井までメール.

【メールの送り方例】

```
% mail –s 自分のアカウント名 kobeuniv.compra1@gmail.com < result.txt
```

【締切】5月23日（水）, 午後５時.

宿題(トライしてみて)

【課題】

演習２のプログラムのループをスレッド数 2 で並列化し、制御変数 i が奇数の場合にはスレッド0, 制御変数 i が偶数の場合にはスレッド 1 が処理するようにせよ.

(うまく処理されているか確認できるならしてください。)

プログラムを前の課題の結果の最後に入れて、そのファイル(result.txt)の内容を臼井までメール.

【メールの送り方】

```
% mail –s 自分のアカウント名 kobeuniv.compra1@gmail.com < result.txt
```

【締切】5月23日（水）, 午後５時.