スカラチューニングと OpenMPによるコードの高速化

松本洋介
千葉大学理学研究科

謝辞
C言語への対応：簑島敬（JAMSTEC）

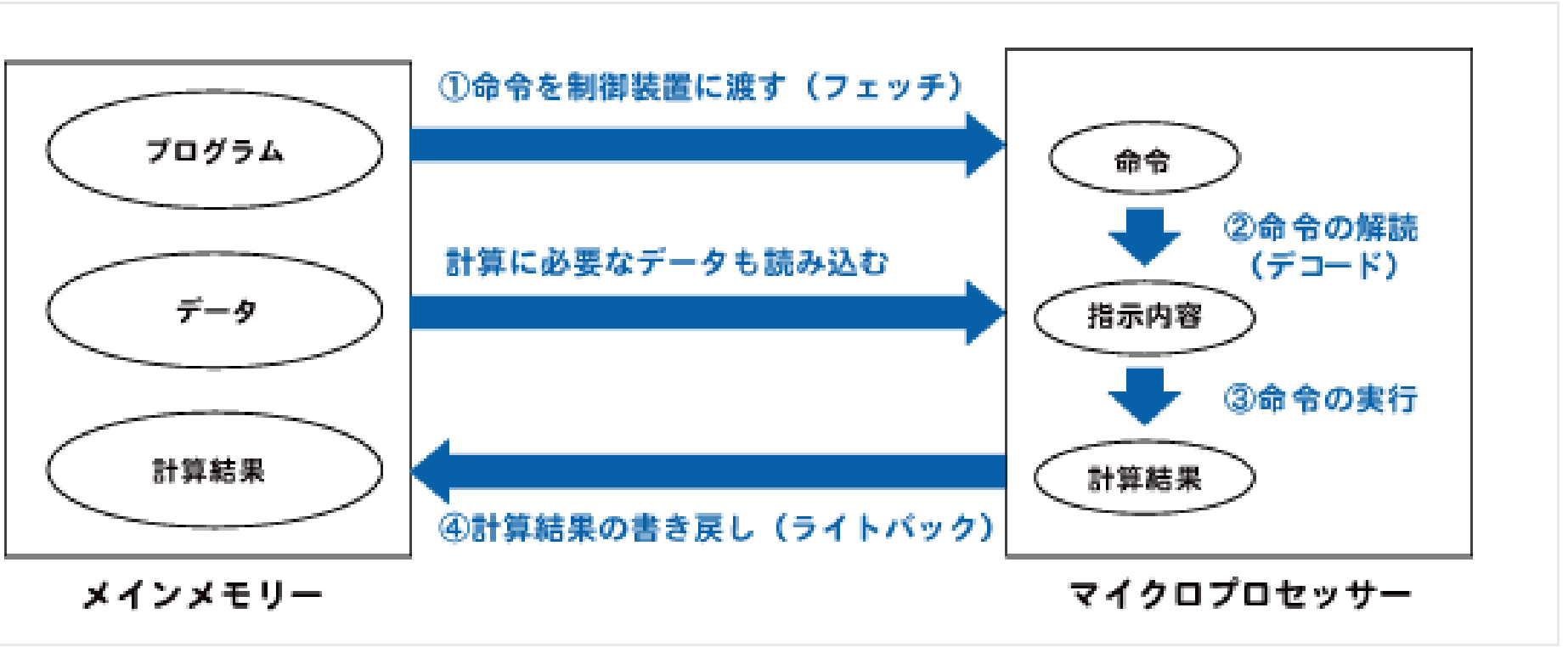
宇宙磁気流体・プラズマシミュレーションサマースクール
2015年8月4日　千葉大学統合情報センター

内容

- イントロダクション
- スカラチューニング
- OpenMPによる並列化
- 最近のHPC分野の動向
- まとめ

イントロダクション

命令実行の流れ



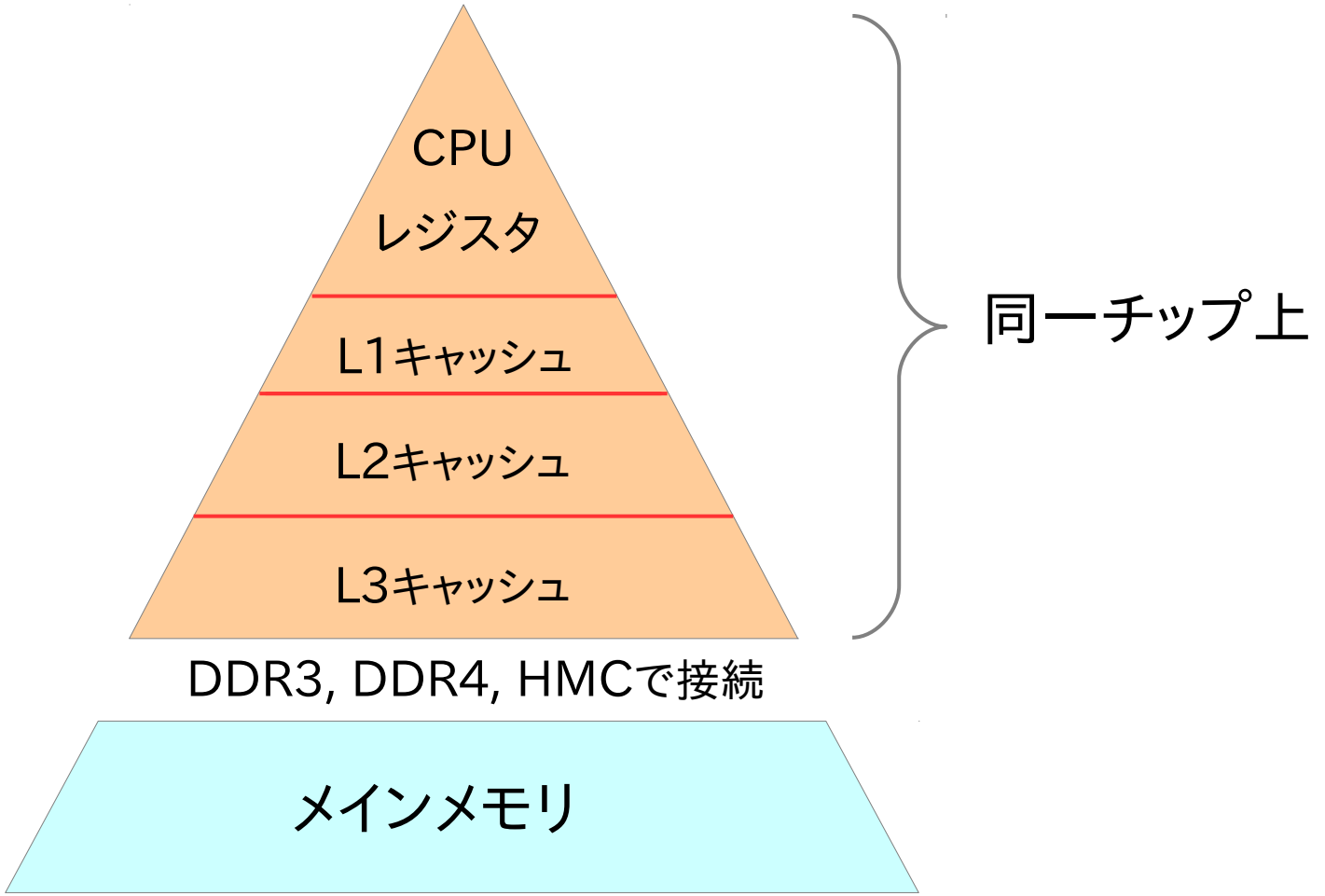
http://japan.intel.com/contents/museum/mpuworks/index.html

実行にかかる時間は主に、

1.命令による実行（②、③）
2.データの入出力（①、④）

による。

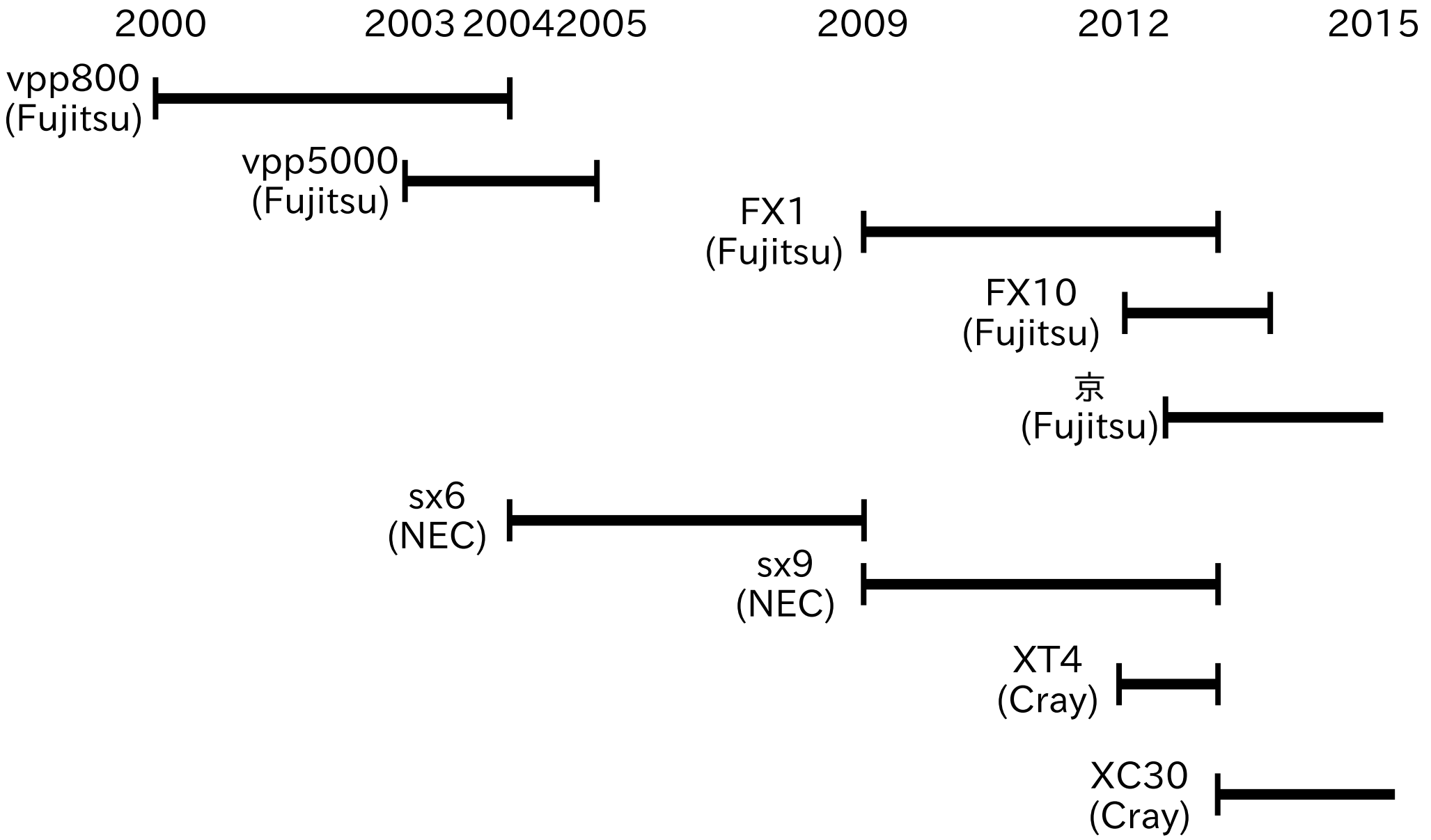
メモリの階層構造



「京」の場合

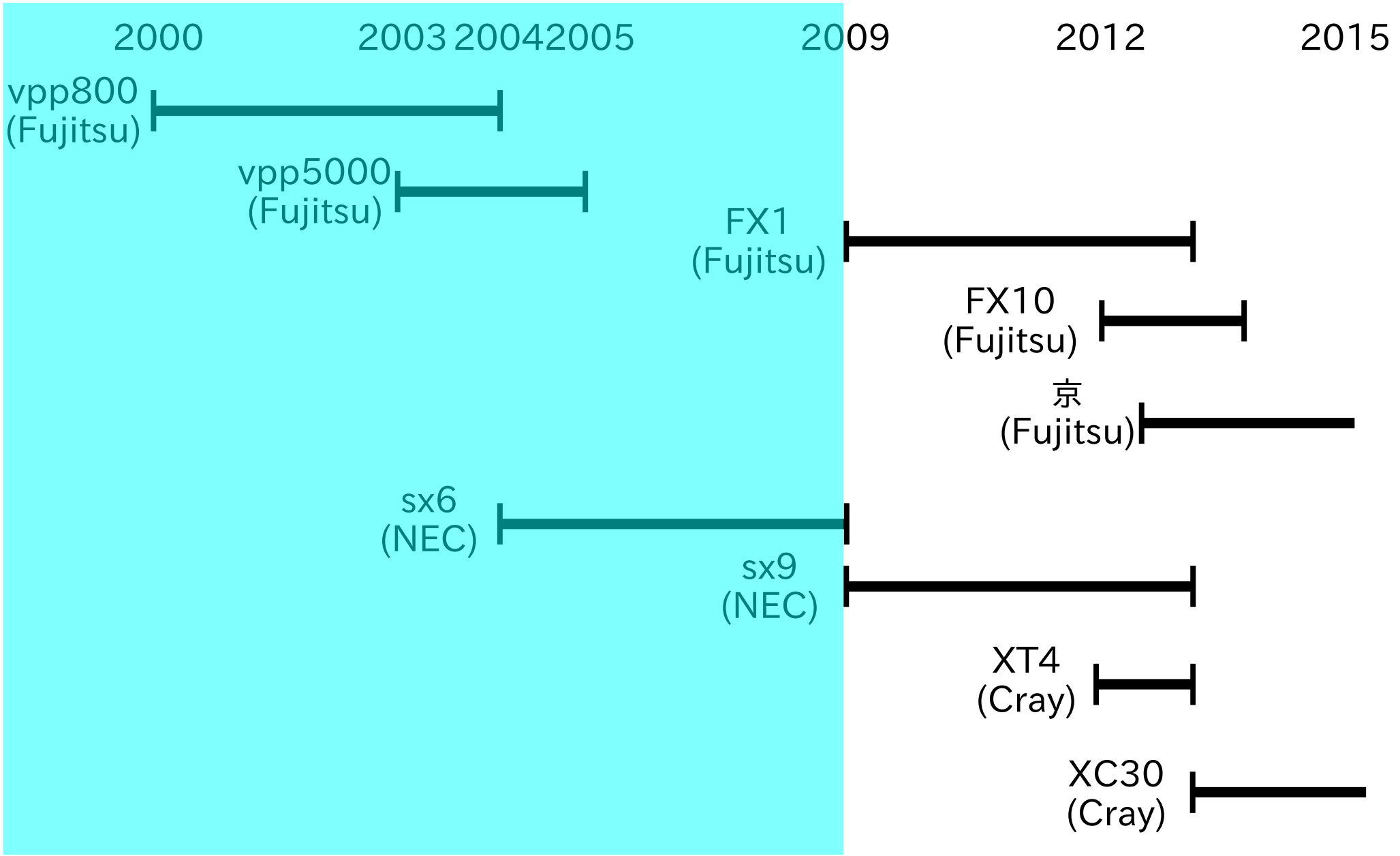
- register : サイクル
- L1$: 32 kB, 数サイクル
- L2$: 6MB, >180 GB/s
- Memory: 16GB, 64GB/s

私のスパコン利用暦

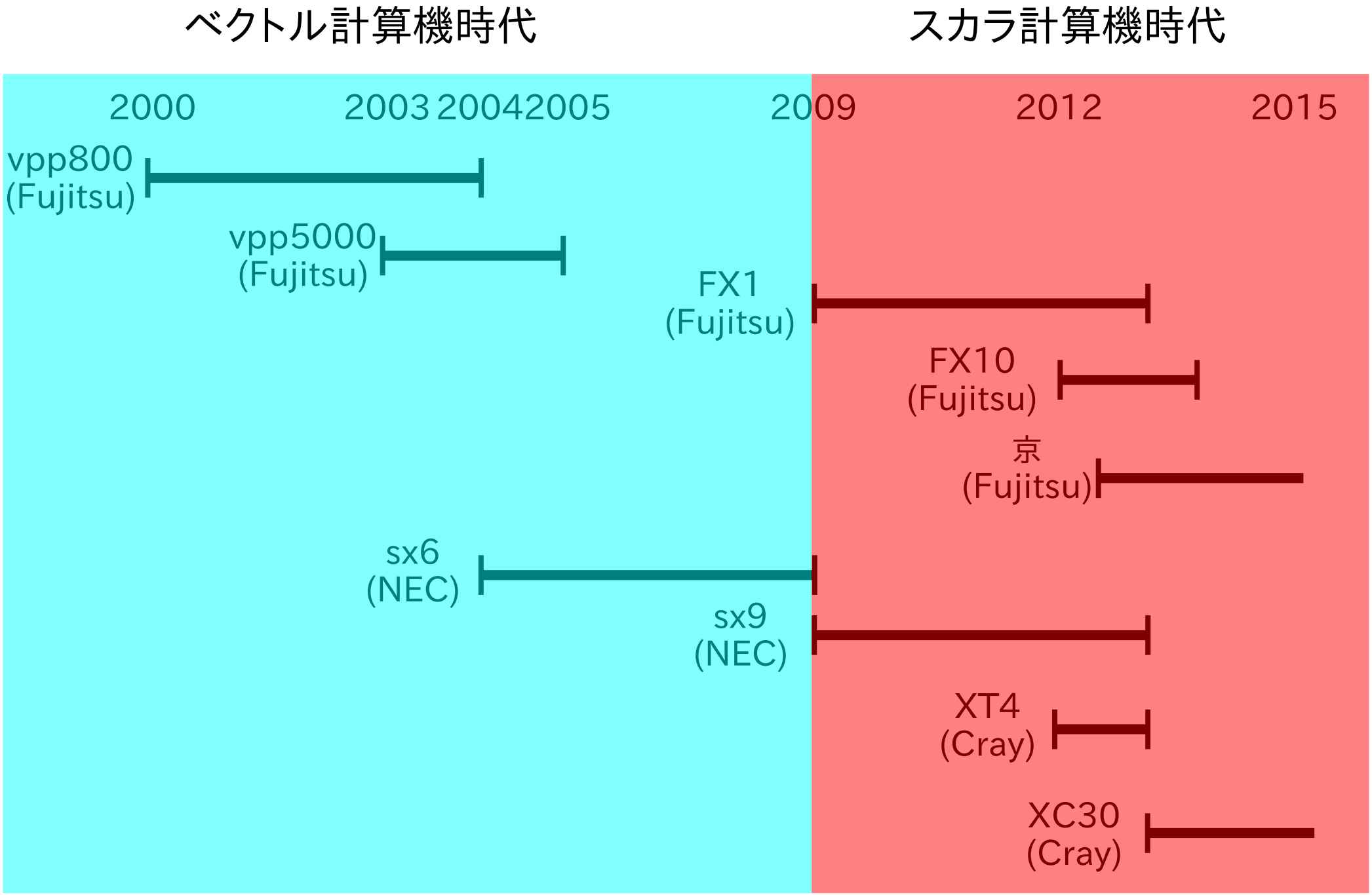

2000
2003
2004
2005
2009
2012
2015
vpp800
(Fujitsu)
vpp5000
(Fujitsu)
FX1
(Fujitsu)
FX10
(Fujitsu)
京
(Fujitsu)
sx6
(NEC)
sx9
(NEC)
XT4
(Cray)
XC30
(Cray)

私のスパコン利用暦

ベクトル計算機時代

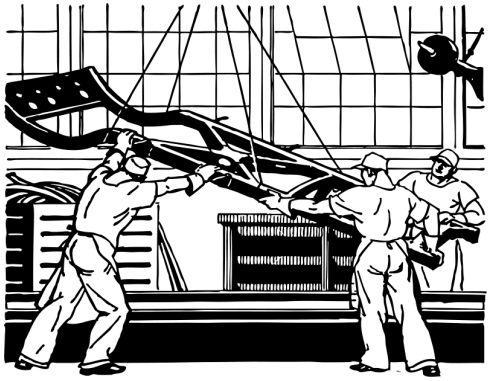


私のスパコン利用暦


ベクトル計算機時代
スカラ計算機時代
2000
2003
2004
2005
2009
2012
2015
vpp800
(Fujitsu)
vpp5000
(Fujitsu)
FX1
(Fujitsu)
FX10
(Fujitsu)
京
(Fujitsu)
sx6
(NEC)
sx9
(NEC)
XT4
(Cray)
XC30
(Cray)

スカラ／ベクトル？

- スカラ

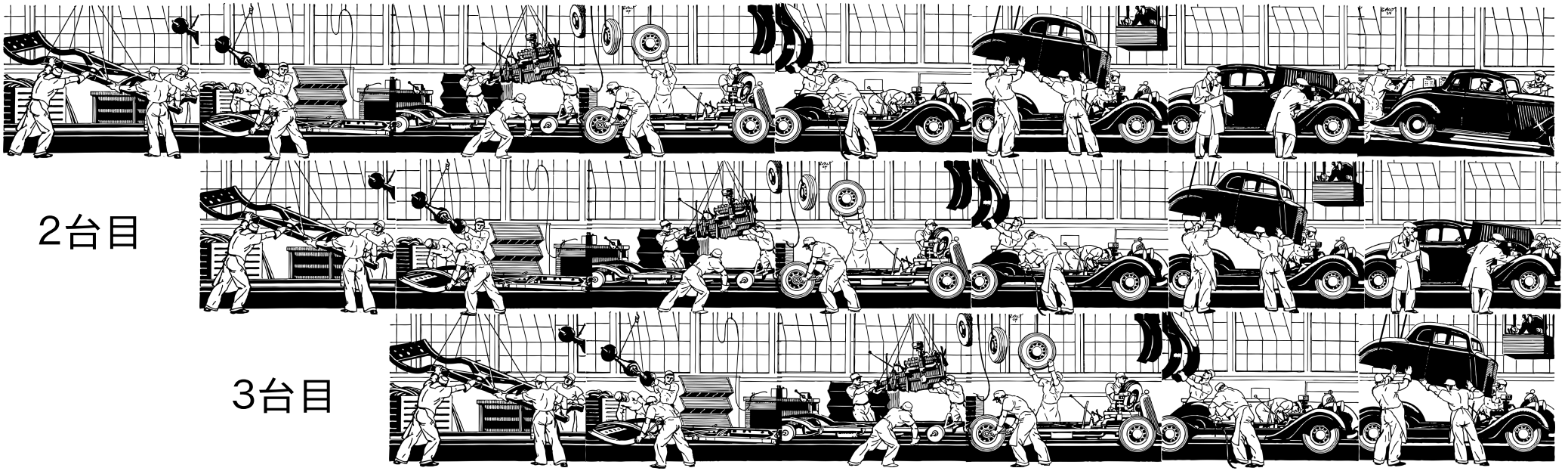


同じ車を何台も作る作業に例えると…

```
do k=1,100
do j=1,100
do i=1,100
   a(i,j,k) = c*b(i,j,k)+d(i,j,k)
enddo
enddo
enddo
```

- ベクトル（データのパイプライン処理）

サイクル



2台目

3台目

近年の演算処理の高速化のしくみ

- サイクル時間（1/周波数）の短縮（→周波数～3GHzで頭打ち）
- ベクトル化
  - パイプライン処理
  - SIMD（複数演算器）
- メモリ構造の階層化
- 並列化（マルチコア/MIMD）

近年の演算処理の高速化のしくみ

- ~~サイクル時間（1/周波数）の短縮（→~~周波数～3GHzで頭打ち）
- ベクトル化
  - パイプライン処理
  - SIMD（複数演算器）
- メモリ構造の階層化
- 並列化（マルチコア/MIMD）

近年の演算処理の高速化のしくみ

- ~~サイクル時間（1/周波数）の短縮（→~~周波数～3GHzで頭打ち）
- ベクトル化
  - パイプライン処理
  - SIMD（複数演算器）

  } ユーザから見たら同じ
- メモリ構造の階層化
- 並列化（マルチコア/MIMD）

近年の演算処理の高速化のしくみ

- ~~サイクル時間（1/周波数）の短縮（→~~周波数～3GHzで頭打ち）
- ベクトル化
  - パイプライン処理
  - SIMD（複数演算器）

  } ユーザから見たら同じ
- メモリ構造の階層化
- 並列化（マルチコア/MIMD）

近年の計算機では、**SIMD**化、**キャッシュヒット**率の向上、
マルチコアによる並列化が高速化のポイント

スカラチューニング

対象

- 宇宙磁気流体プラズマシミュレーションにかかわること
- すなわち、
  - 差分法：磁気流体（MHD）・ブラソフシミュレーション
  - 粒子法：電磁粒子（PIC）シミュレーション
- 行列の演算（例：LU分解など）は対象外
- Fortran, C

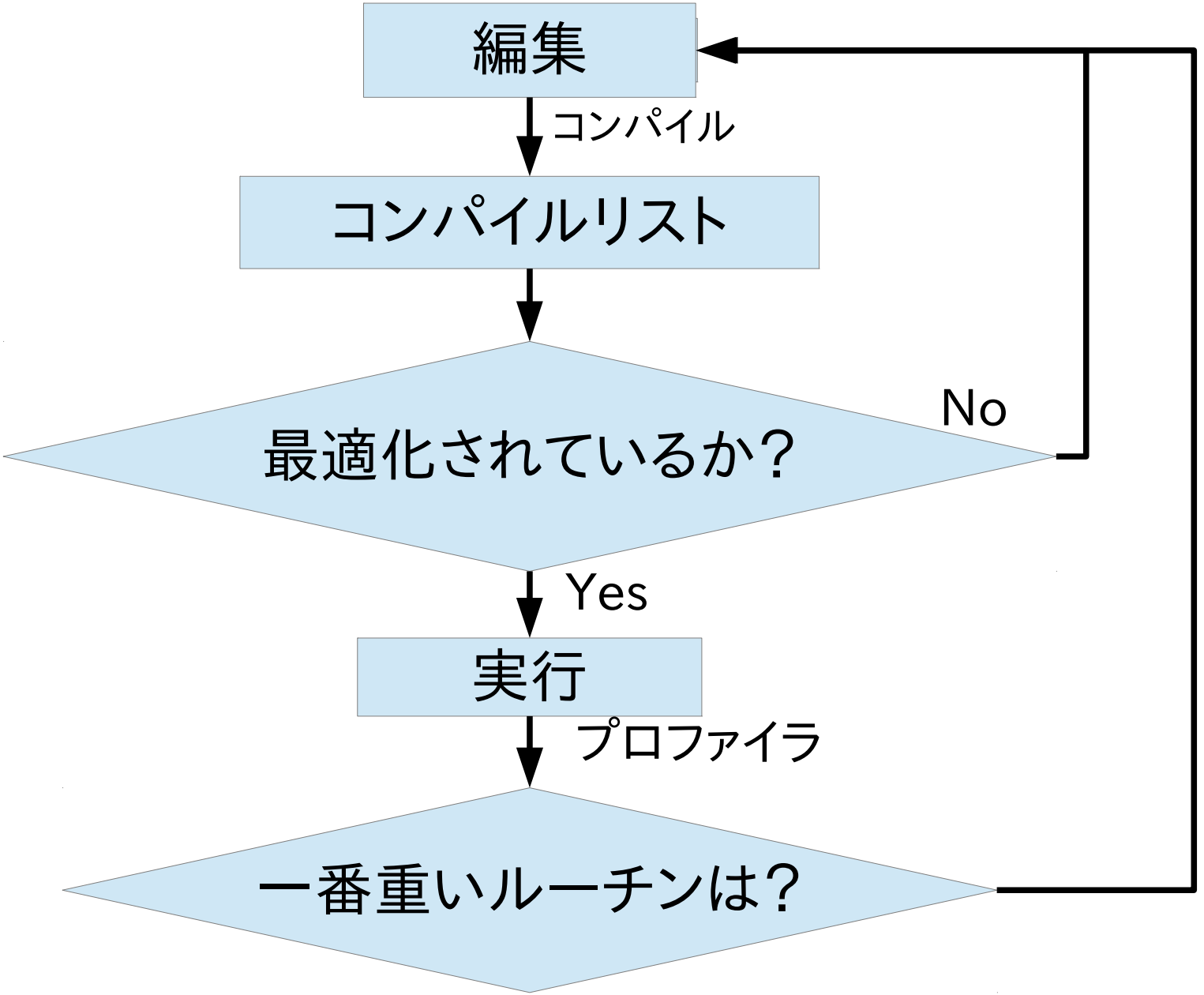
注意

一般に、チューニングすると可読性が損なわれます。まずは読みやすいコードを書き、充分テストしてバグを除いてからチューニングを行いましょう。

チューニングが必要？

- 無理にしなくて良いです。（好きでもしんどい）
- 最先端（大規模）シミュレーション研究では必須。なぜなら、、、
- 1ランで数週間→2倍の速度向上で10日単位の短縮
- 「京」などの大規模計算申請書類では、実行効率・並列化率などの情報が求められる。
- 実行効率15%以上あれば、計算機資源の獲得において、他分野との競争力になる。

チューニングの手順



コンパイルリスト

- 最適化情報の詳細を出力
  - インライン展開等の最適化
  - SIMD化
  - 並列化
- コンパイルオプション
  - gcc/gfortran: N/A
  - icc/ifort: -opt-report, -vec-report, -par-report

プロファイラの利用

- 各サブルーチンの経過時間を計測
- ホットスポット（一番処理が重いサブルーチン）から最適化
- 商用コンパイラ（intel, PGI, スパコン等）では、詳細情報（キャッシュミス率、FLOPS）が得られる
- GNUでは、gprof
- gprofの使い方
  - gfortran (gcc) -pg test.f90
  - ifort (icc) -p test.f90
  - ./a.out
  - gprof ./a.out gmon.out
    > output.txt

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
53.24	507.03	507.03	2048	0.25	0.25	__particle_MOD_particle__solv
33.32	824.40	317.37	2048	0.15	0.15	__field_MOD_ele_cur
6.81	889.23	64.83	2048	0.03	0.19	__field_MOD_field__fdtd_i
6.37	949.94	60.71	2048	0.03	0.03	__boundary_MOD_boundary__particle
0.24	952.20	2.26	2048	0.00	0.00	__field_MOD_cgm
0.02	952.37	0.17	3	0.06	0.06	__fio_MOD_fio__energy
0.01	952.51	0.14	4096000	0.00	0.00	__random_gen_MOD_random_gen__bm
0.00	952.55	0.04	1	0.04	0.18	__init_MOD_init__loading
0.00	952.56	0.01	57344	0.00	0.00	__boundary_MOD_boundary__phi

output.txt

スカラチューニングのポイント

- コンパイラ(≠人)にやさしいプログラム構造
  - ループ内で分岐は使わない(if文の代わりにmin, max, sign で、goto文は不可)
  - ループ内処理を単純にする(SIMD化促進)
  - 外部関数のインライン展開
- データの局所化を高める
  - 繰り返し使用するデータはなるべくひとまとめにして、キャッシュに乗るようにする。
  - 一時変数の再利用
  - 連続アクセス
  - ポインタは使わない(Fortran)

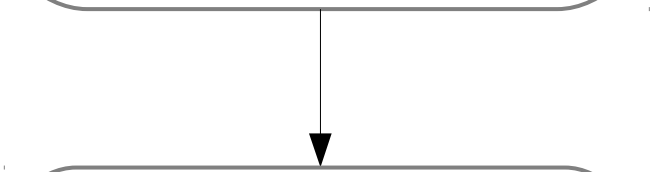
基本的なtips

- 割り算を掛け算に
  - a(i) = b(i)/c → c = 1.0/c ; a(i) = b(i)*c
- べき乗表記はなるべく使わない
  - a(i) = b(i)**2　→ a(i) = b(i)*b(i)
  - a(i) = b(i)**0.5 → a(i) = sqrt(b(i))
- 因数分解をして演算数を削減
  - y=a*x*x*x*x+b*x*x*x+c*x*x+d*x → y=x*(d+x*(c+x*(b+x*(a))))
  - 演算回数13 → 7
- 一時変数は出来る限り再利用（レジスタの節約）

分岐処理の回避1

Fortran:

```
do i=1,nx
   if(a /= 0.0)then
      b(i) = c(i)/a
   else
      b(i) = c(i)
   endif
enddo
```



```
if(a == 0.0) a=1.0
a = 1.0/a
do i=1,nx
   b(i) = c(i)*a
enddo
```

分岐処理の回避2（minmod関数）

Fortran:

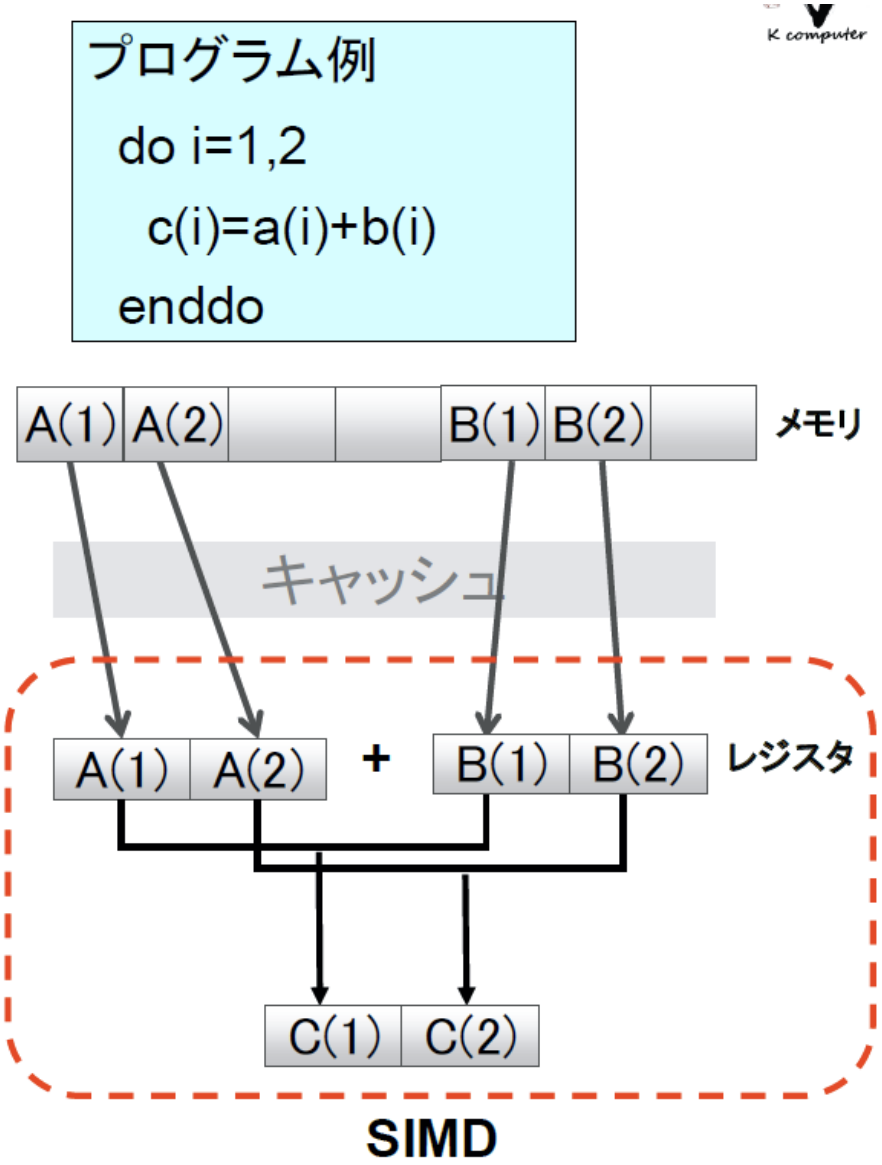
```
do i=1,nx
 if(a(i)*b(i) < 0.0)then
    c(i) = 0
 else
    c(i) = sign(1.0,a(i)) &
           *min(abs(a(i)),abs(b(i)))
 endif
enddo
```



```
do i=1,nx
 c(i) = sign(1.0,a(i))             &
       *max(0.0,                   &
            min(abs(a(i)),         &
            sign(1.0,a(i))*b(i)) &
           )
enddo
```

SIMD: Single Instruction Multiple Data

- ユーザレベルではベクトル化と同じ。ただし、ベクトル長は2～4と、ベクトル計算機のそれ（256）に比べてずっと短い。
- 最内側ループに対してベクトル化
- 最近ではループ内にIF文が入っていてもSIMD化してくれる（マスク付きSIMD化）。真率が高ければ効果的。
- コンパイルオプションで最適化
  - gcc/gfortran: -m{avx, sse4}
  - icc/ifort: -x{avx,sse4}



from K computer manual

SIMD化の阻害例

SIMD化されない → 書き方の工夫 → SIMD化される

例1: ループ番号間に依存性がある場合

```
a(1) = dx
do i=2,nx
 a(i) = a(i-1)+dx
enddo
```

```
do i=1,nx
 a(i) = i*dx
enddo
```

例2: ループ番号によって処理が異なる場合

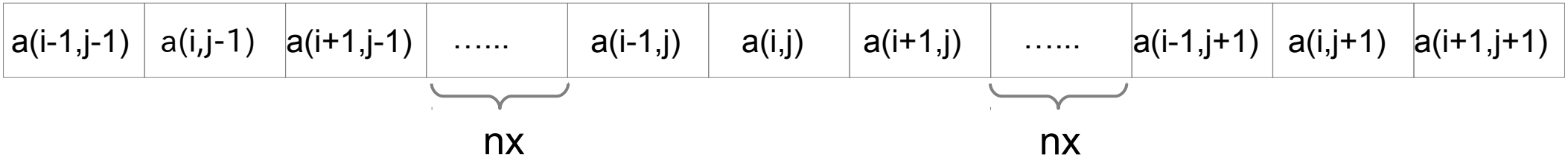
```
do i=1,nx
 if(a(i) < 0)then
   b(i-1) = c*a(i)
 else
   b(i+1) = c*a(i)
 endif
enddo
```

```
do i=1,nx
 w1 = 0.5*(1.0-sign(1.0,a(i)))
 w2 = 0.5*(1.0+sign(1.0,a(i)))
 b(i-1) = c*w1*a(i)
 b(i+1) = c*w2*a(i)
enddo
```

配列の宣言とメモリ空間1 (Fortran)

```
dimension a(nx,ny)
```

配列aのメモリ空間上での配置は、




```
do i=1,nx
   do j=1,ny
      a(i,j) = i+j
   enddo
enddo
```

```
do j=1,ny
   do i=1,nx
      a(i,j) = i+j
   enddo
enddo
```

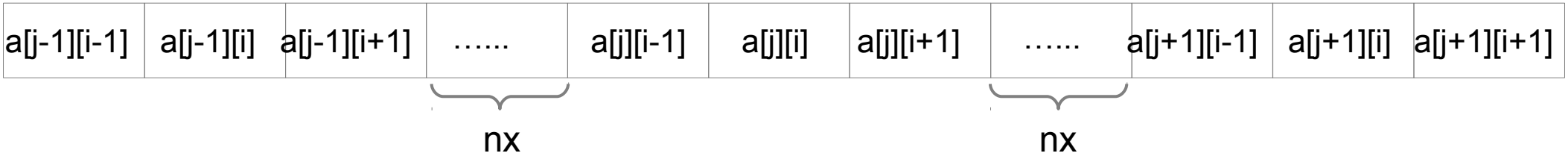
間隔nxで飛び飛びにアドレスにアクセスすることになるので、メモリへの書き込みが非常に遅い

アドレスに連続アクセスするので、メモリへの書き込みが速い

配列の宣言とメモリ空間1 (C/C++)

```
float a[ny][nx];
```

配列aのメモリ空間上での配置は、




```
for(i=0;i<nx;i++){
   for(j=0;j<ny;j++){
      a[j][i] = i+j;
   }
}
```

```
for(j=0;j<ny;j++){
   for(i=0;i<nx;i++){
      a[j][i] = i+j;
   }
}
```

間隔nxで飛び飛びにアドレスにアクセスすることになるので、メモリへの書き込みが非常に遅い

アドレスに連続アクセスするので、メモリへの書き込みが速い

配列の宣言とメモリ空間2

連続の式：$\rho^{n+1} = f(\rho^n, v_x^n, v_y^n)$

次のステップに進むためには、自分自身（ρ）の他に速度場が必要

```
dimension rho(nx,ny), vx(nx,ny), vy(nx,ny), ...
```

と変数を個別に用意する代わりに、

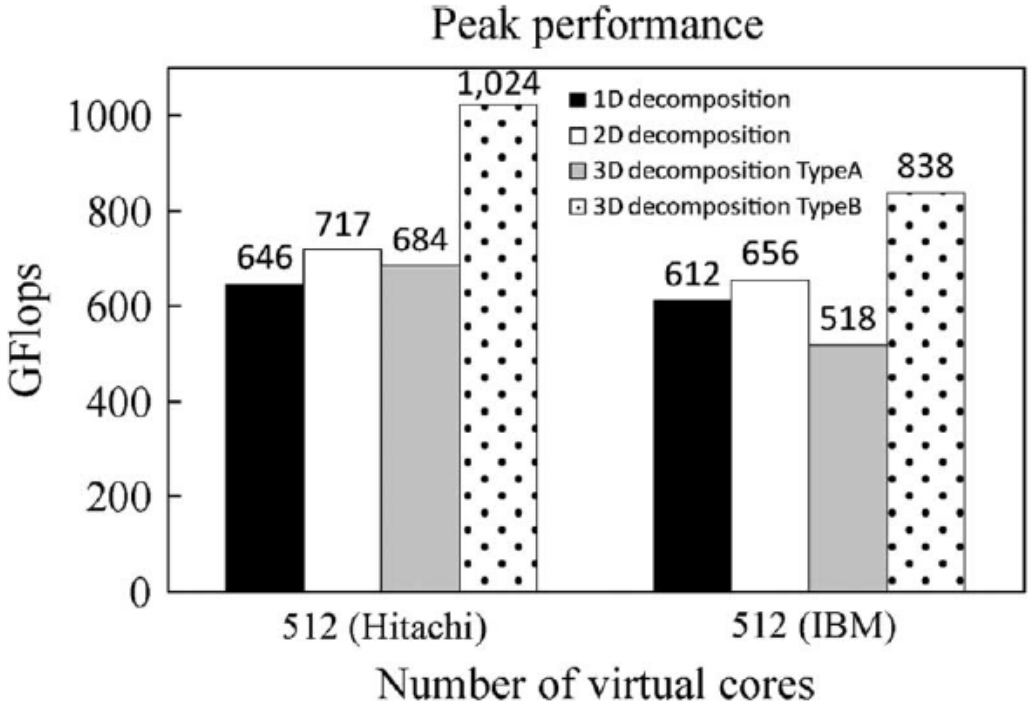
```
dimension f(8,nx,ny) ! 1:rho, 2:p, 3-5:v, 6-8:B
```

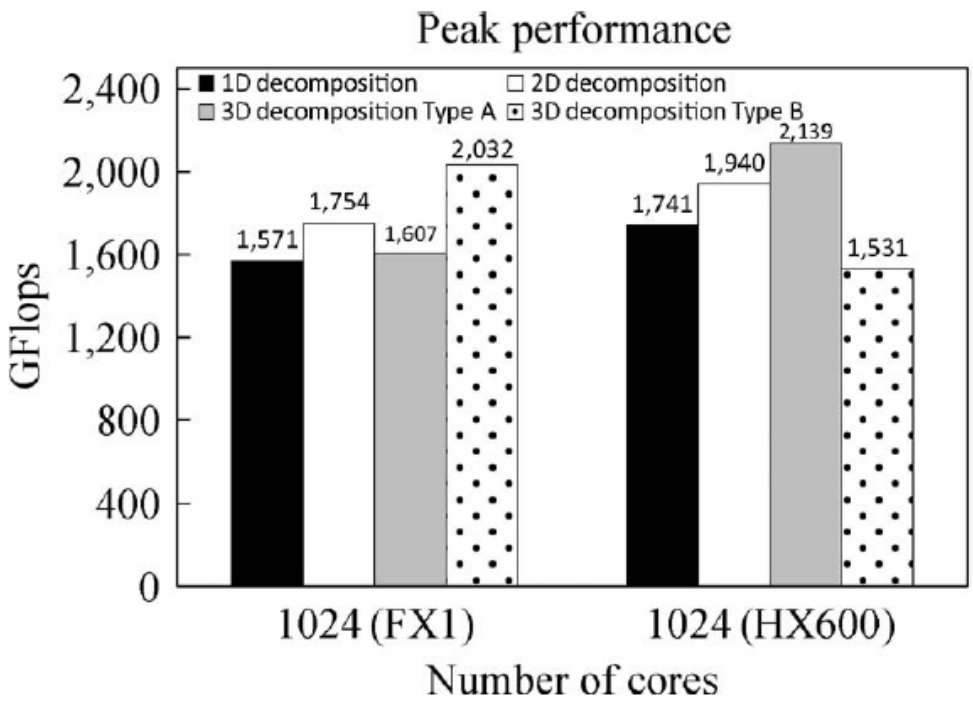
のように、一つの変数にまとめて配列を用意する。このようにすると、必要となる各物理量がメモリ空間上の近い位置に配置される（キャッシュラインにのりやすい）。→システム方程式を解くための工夫

配列の宣言とメモリ空間2（続き）

TypeA: f(nx,nx,nz,8)

TypeB: f(8,nx,nx,nz)





近年のキャッシュ重視型のスパコンにおいて効果的

Fukazawa *et al.*, *IEEE Trans. Plasma Sci.*, 2010.

配列の宣言とメモリ空間3（C言語）

C言語で静的に配列を宣言する場合は、

```
float a[ny][nx];
```

とするが、領域分割の並列計算では動的に（mallocで）配列を確保する場合が多く、上記の宣言では難しい。2次元配列を1次元配列として宣言する方が、メモリ空間上で連続的に領域を確保できる。

```
double *a;
a=(double*)malloc(sizeof(double)*nx*ny);

for (j=0;j<ny;j++){
   for (i=0;i<nx;i++){
      a[nx*j+i] = i+j;
   }
}
```

キャッシュと配列ブロック

- 2次キャッシュ容量～数MB
  - 倍精度で100x100x100グリッド分程度
  - MHD計算の1ノードあたりの配列数としては、まだちょっと足りない
  - PIC計算では、セル内の粒子が必要なグリッド上の場のデータがキャッシュに乗らない
- 配列ブロック
  - 必要なデータだけをキャッシュに収まる程度の別の小さな配列に事前に格納
  - PIC法で(i,j,k)セルに属する粒子が必要な場の情報をあらかじめパック

```
tmp(1:6,-1:1,-1:1,-1:1) = f(1:6,i-1:i+1,j-1:j+1,k-1:k+1)
```

インライン展開

- 外部（ユーザー定義）関数はプログラムの可読性向上に一役。しかし、、

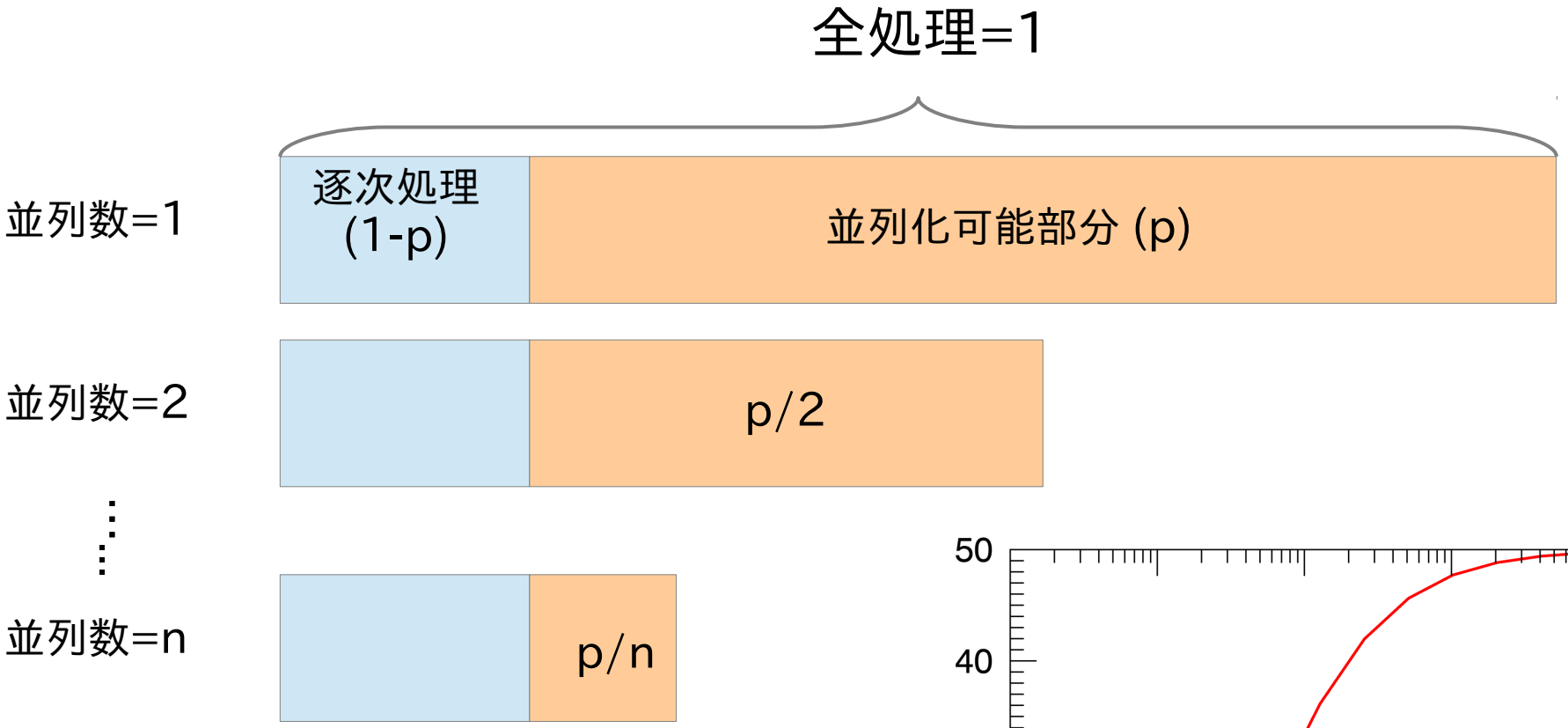
```
do i=1,nx
   a(i) = myfunc(b(i))
enddo
```

のように、ループ内で繰り返し呼び出す場合、呼び出しのオーバーヘッドが大きい。関数内の手続きが短い場合は、内容をその場所に展開する→インライン展開

- コンパイル時に指定（同一ファイル内に定義される関数）
  - gcc/gfortran: -O3 もしくは -finline-functions
  - icc: -O{2,3}, ifort: -finline
- コンパイル時に指定（別ファイル内に定義される関数）
  - icc/ifort: -fast もしくは -ipo

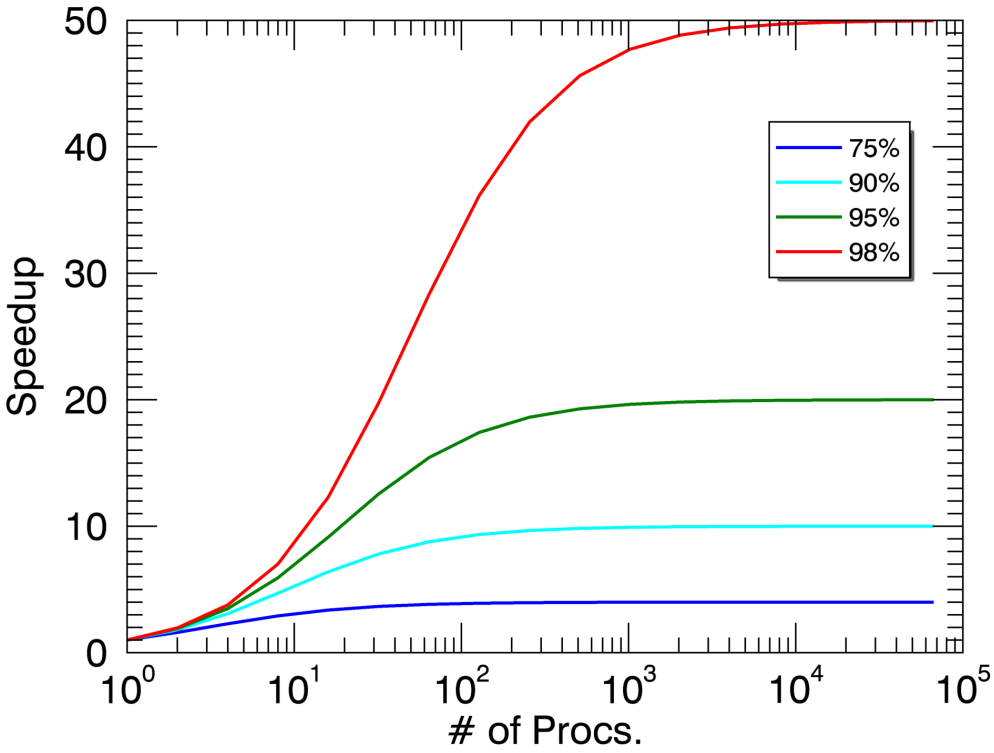
OpenMPによるコードの並列化

アムダールの法則



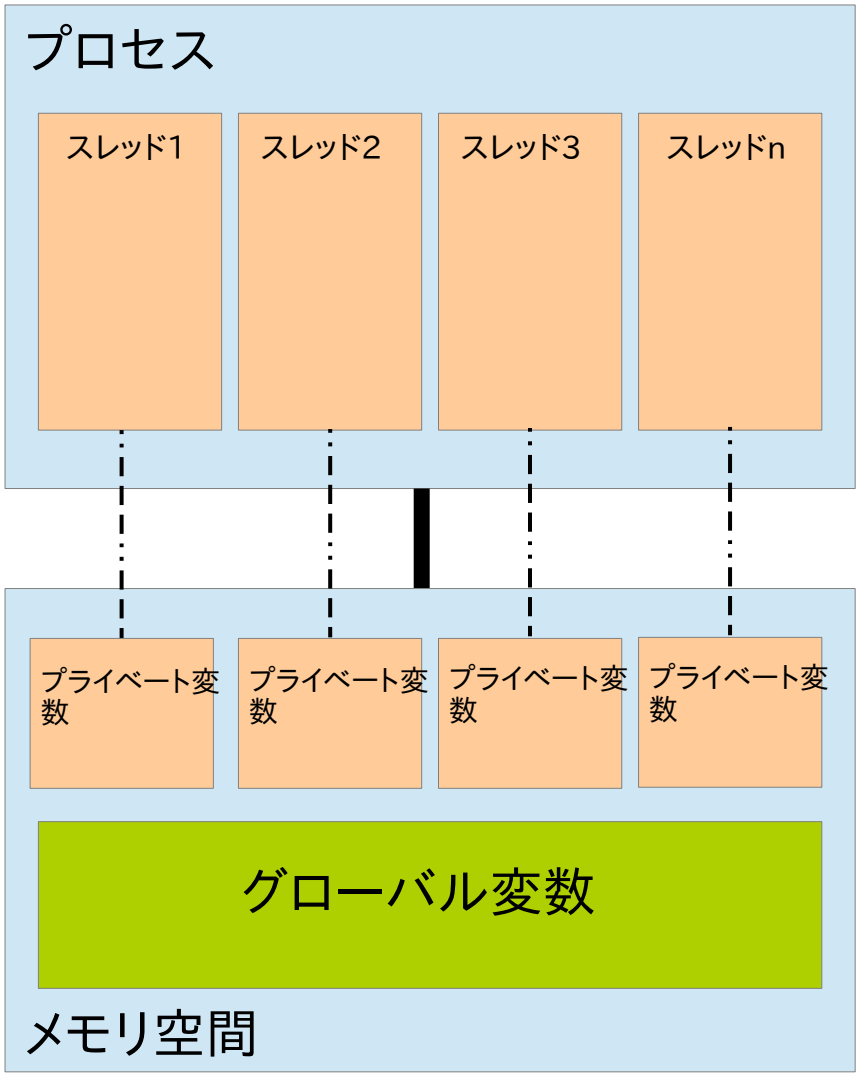
$$\text{性能向上率} = \frac{1}{(1-p)+\frac{p}{n}}$$

少なくも並列化率**p>0.95**である必要あり

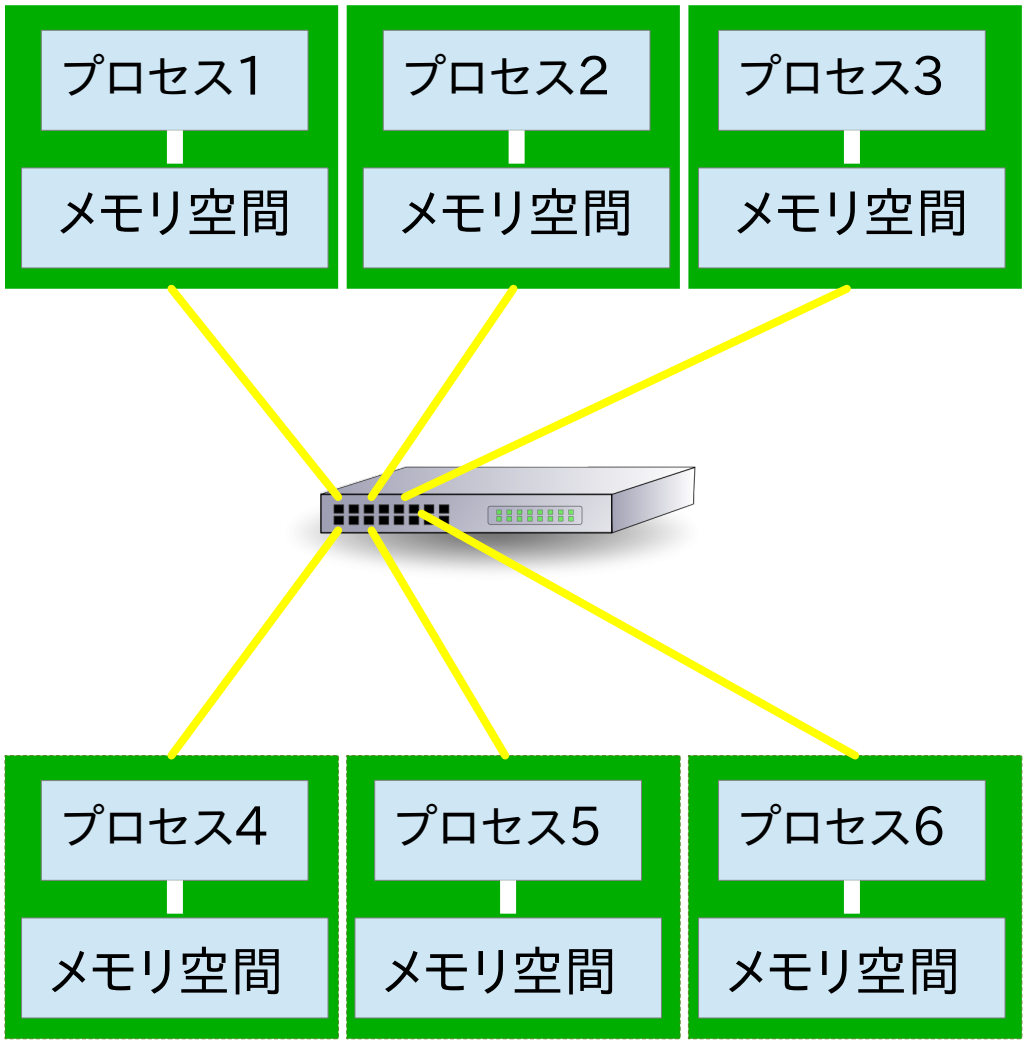


スレッド並列とプロセス並列

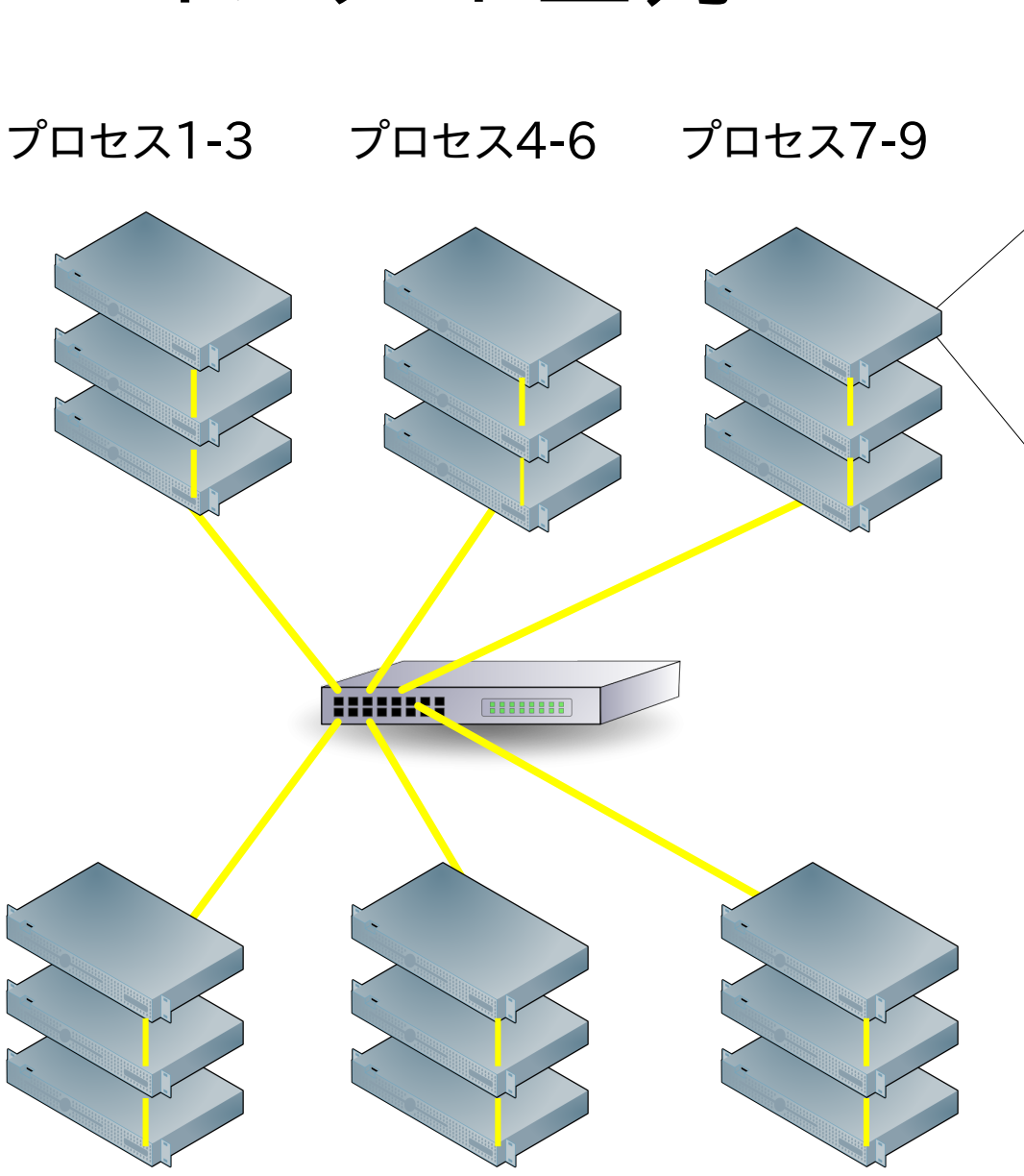
スレッド並列

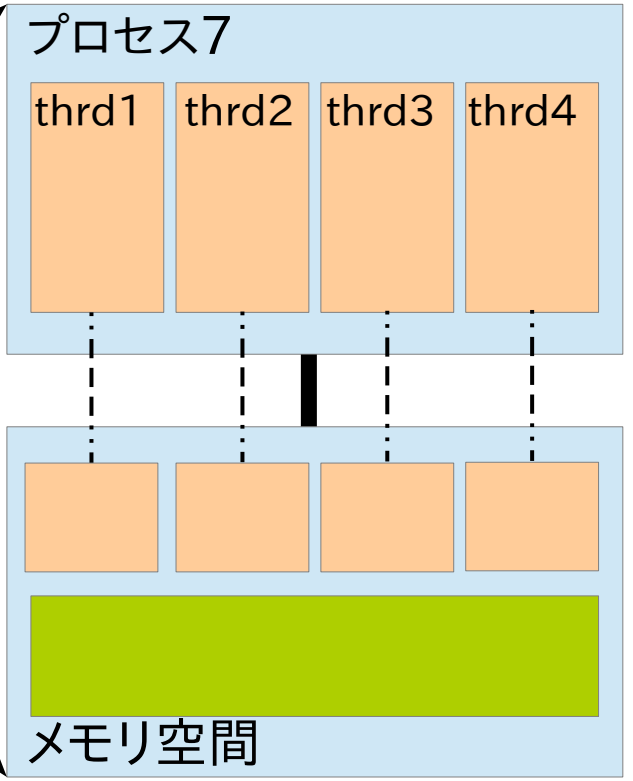


プロセス並列



ハイブリッド並列





- この例では全72並列
- プロセス間はMPIによる通信
- 各プロセスに4スレッド
- スレッド数分プロセス数を削減
- MPIによる通信／同期待ちのオーバヘッドを軽減
- 出力ファイル数の削減

- スレッド並列計算を行うためのAPI
- コンパイルオプションで有効
  - gcc/gfortran: -fopenmp
  - icc/ifort: -openmp
- プログラムに指示行を挿入（オプション無効時はコメント行と見なされる（C言語は警告される場合も）
- 自動並列化に比べて柔軟に最適化が可能
- 標準規格なため、マシン／コンパイラに依らずポータブル
- 2015年8月現在、OpenMP 4.1。SIMD化の指示行、アクセラレータ（後述）への対応
- http://www.openmp.org

スレッド数の設定

- 基本的にはシェルの環境変数 $OMP_NUM_THREADS でスレッド数を指定する
  - setenv OMP_NUM_THREADS 8
  - 指定しなければ、システムの全コア数
- プログラム内部で関数で設定（omp_lib/omp.hをインクルードする必要あり）

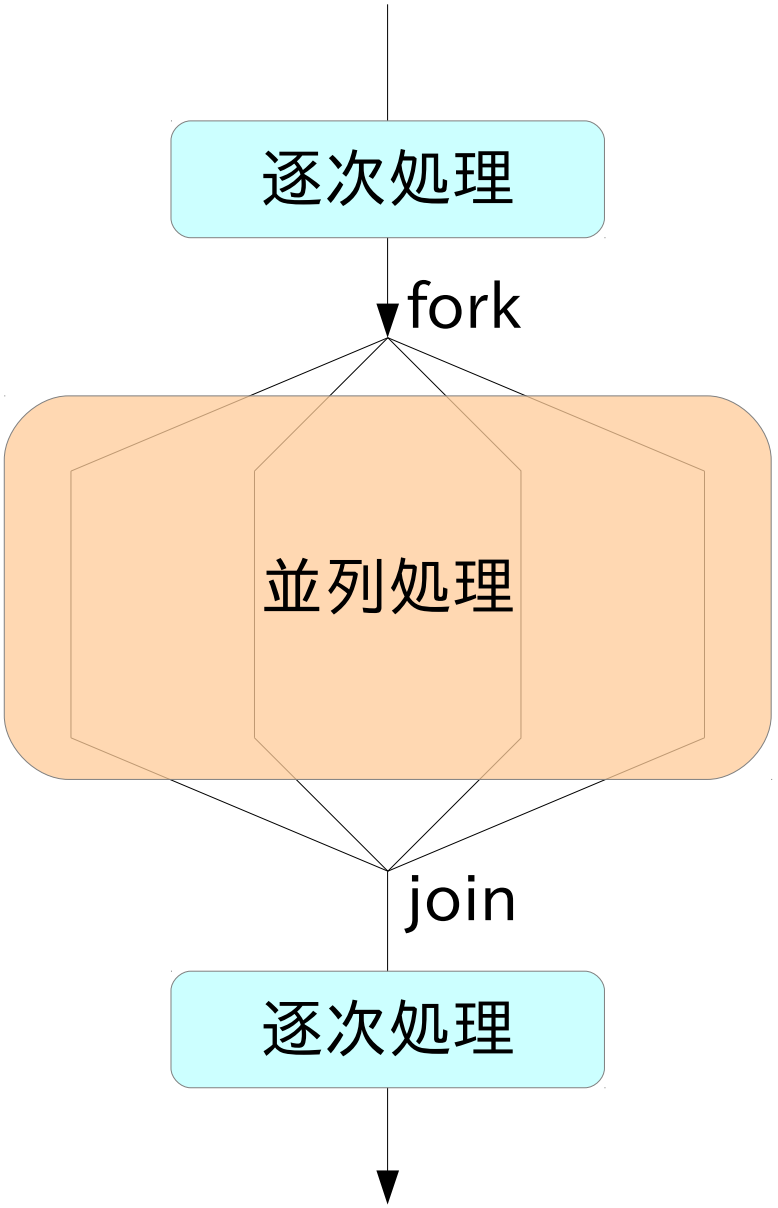
Fortran:

```
!$use omp_lib
integer, parameter :: nthrd =
8

call
omp_set_num_threads(nthrd)
```

C:

```
#include <omp.h>
int nthrd=8;

omp_set_num_threads(nthrd);
```

全体の流れ:fork-join モデル



Fortran:

```
program main

write(*,*) 'serial region'

!$OMP PARALLEL
….
….
….
….
….
write(*,*) 'parallel region'
….
….
….
….
….

!$OMP END PARALLEL

write(*,*) 'serial region'

stop
end
```

C:

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

int main(void)
{
 puts(“serial region”);

#pragma omp parallel
 {
  …
  …
  …
  puts(“parallel region”);
  …
  …
  …
 }

 puts(“serial region”);

 return 0;
}
```

ループの並列化

*$OMP_SCHEDULE／SCHEDULE句で分担方法変更可

```
!$OMP PARALLEL DO
  do i=1,100
     b(i) = c*a(i)
  enddo
!$OMP END PARALLEL DO

call mysub(b)

!$OMP PARALLEL
!$OMP DO
  do i=1,100
     d(i) = c*b(i)
  enddo
!$OMP END DO
!$OMP DO
  do i=1,100
     e(i) = c*d(i)
  enddo
!$OMP END DO
!$OMP END PARALLEL
```

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

 mysub(b);

#pragma omp parallel
 {
#pragma omp for
  for (i=0;i<100;i++){
   d[i]=c*b[i];
  }

#pragma omp for
  for (i=0;i<100;i++){
   e[i]=c*d[i];
  }
 }
```

i=1-100を
各スレッドが
均等に分担

スレッドの立ち上げは
なるべくまとめて

pragma omp for の
直後のforループが並列
処理される。間に”{“を
入れてはならない

多重ループの並列化

```
 do j=1,100
!$OMP PARALLEL DO
   do i=1,100
     b(i,j) = c*a(i,j)
   enddo
!$OMP END PARALLEL DO
 enddo
```

スレッドの立ち上げが100回も行われ、オーバーヘッドが大きい

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

```
!$OMP PARALLEL DO &
!$OMP PRIVATE(i)
 do j=1,100
   do i=1,100
     b(i,j) = c*a(i,j)
   enddo
 enddo
!$OMP END PARALLEL DO
```

最外ループを並列化
内側ループのカウンタ変数 *i* はプライベート宣言が必要。

```
#pragma omp parallel
{
#pragma omp for private(i)
 for (j=0;j<100;j++){
  for (i=0;i<100;i++){
   b[j][i]=c*a[j][i];
  }
 }
}
```

多重ループの並列化（続き）

- 多重ループでは**最外ループを並列化**するのが基本。ループの内側に指示行を入れると、外側ループの回転数分スレッドのfork/joinが行われ、スレッド立ち上げのオーバーヘッドが大きくなる。

- 内側にあるループのカウンタ変数（i, j, ..）はスレッド固有の変数とする必要があるため、PRIVATE宣言をする。そうしないと、スレッド間で上書きしてしまう。

グローバル／プライベート変数

スレッド間でtmpを上書きしまうので正しい結果が得られない

```
!$OMP PARALLEL DO
 do i=1,100
   tmp = myfunc(i)
   a(i) = tmp
 enddo
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for
 for (i=0;i<100;i++){
  tmp=myfunc(i);
  a[i]=tmp;
 }
```

Cの場合はループ内で変数宣言すれば問題なし。

```
!$OMP PARALLEL DO &
!$OMP PRIVATE(tmp)
 do i=1,100
   tmp = myfunc(i)
   a(i) = tmp
 enddo
!$OMP END PARALLEL DO
```

```
#pragma omp parallel{
#pragma omp for private(tmp)
 for (i=0;i<100;i++){
  tmp=myfunc(i);
  a[i]=tmp;
 }
}
```

```
#pragma omp parallel for
 for (i=0;i<100;i++){
  double tmp;
  tmp=myfunc(i);
  a[i]=tmp;
 }
```

ループ内変数の演算 (REDUCTION)

```
 sum = 0.0
!$OMP PARALLEL DO &
!$OMP REDUCTION(+:sum)
  do i=1,10
     sum = sum+i
  enddo
!$OMP END PARALLEL DO
```

```
sum=1.0;
#pragma omp parallel for reduction(+:sum)
 for (i=0;i<10;i++){
  sum+=i;
 }
```

総和(+)以外には、最大(max)、最小(min)が実用上使われる。

単スレッド処理 (SINGLE)

```
!$OMP PARALLEL
!$OMP DO
  do i=1,100
     b(i) = c*a(i)
  enddo
!$OMP END DO

!$OMP SINGLE
 call output(b)
!$OMP END SINGLE

!$OMP DO
  do i=1,100
     d(i) = c*b(i)
  enddo
!$OMP END DO
!$OMP END PARALLEL
```

スレッドの立ち上げを最初に一回だけ

途中で逐次処理が入る場合はSINGLEで対処

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

#pragma omp single
 {
   output(b);
 }

#pragma omp for
 for (i=0;i<100;i++){
  d[i]=c*b[i];
 }
}
```

スレッドの立ち上げ回数はなるべく少なく。データ入出力など、途中で逐次処理が必要な場合に使う。

バリア同期の回避 (NOWAIT)

```
!$OMP PARALLEL
!$OMP DO
  do i=1,100
     b(i) = c*a(i)
  enddo
!$OMP END DO NOWAIT

!$OMP DO
  do i=1,100
     d(i) = c*b(i)
  enddo
!$OMP END DO

!$OMP DO
  do i=1,200
     e(i) = c*d(i)
  enddo
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

ループの終わりで暗黙に行われるスレッド間の同期待ちをNOWAITで回避

次のループではスレッドに対する変数 d の割り当て範囲が変わるので、同期が必要(注意)

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

#pragma omp for
 for (i=0;i<100;i++){
  d[i]=c*b[i];
 }

#pragma omp for nowait
 for (i=0;i<100;i+=2){
  e[i]=c*d[i];
 }
}
```

スレッド数が大きい場合に高速化に寄与する

OpenMP実装上の注意点

- ユーザが並列処理箇所を明示するため、並列計算に伴う問題発生はプログラマが責任を負う（自動並列化との違い）。
- 並列処理してはいけない箇所でも、明示したら並列化されてしまう
- スレッド内でグローバル/プライベート変数を間違えると結果が不定
- NOWAITで必要な同期を忘れると結果が不定
- 同じプログラムを数回は実行して、結果が変わらないことの確認が必要
- 実装は簡単だけど、デバッグに注意が必要

最近のHPC分野の動向

TOP500 (2015年6月現在)

“ペタFLOPS・メガW時代”

TOP 10 Sites for June 2015

For more information about the sites and systems in the list, click on the links or view the complete list.

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

http://www.top500.org/lists/2015/06/

TOP500 (2015年6月現在)

“ペタFLOPS・メガW時代”

TOP 10 Sites for June 2015

For more information about the sites and systems in the list, click on the links or view the complete list.

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

http://www.top500.org/lists/2015/06/

GREEN500（性能／消費電力）

PEZY（日本のベンチャー企業）が1ー3位独占！

The Green500 List

Listed below are the June 2015 The Green500's energy-efficient supercomputers ranked from 1 to 10.

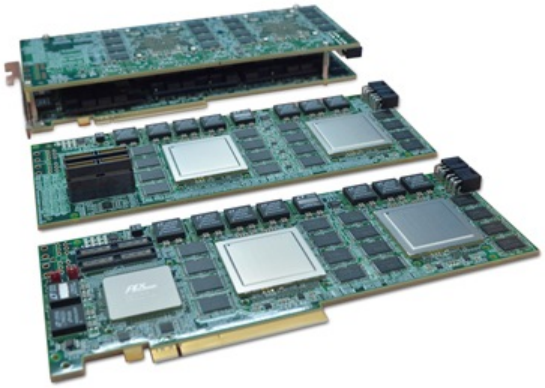
Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	7,031.58	RIKEN	Shoubu - ExaScaler-1.4 80Brick, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SC	50.32
2	6,842.31	High Energy Accelerator Research Organization /KEK	Suiren Blue - ExaScaler-1.4 16Brick, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband, PEZY-SC	28.25
3	6,217.04	High Energy Accelerator Research Organization /KEK	Suiren - ExaScaler 32U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC	32.59
4	5,271.81	GSI Helmholtz Center	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150	57.15
5	4,257.88	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	39.83

http://www.green500.org/lists/green201506

先週土曜日発表があったばかり

GPGPU vs. MIC vs. PEZY-SC

- NVIDIA TESLA
  - ゲーム用途のGPUをHPCに応用（GPGPU)
  - CUDA/OpenACCによるプログラミング
  - SIMD
  - PGI Fortranでも可能（NVIDIAが買収）
- Intel Xeon Phi
  - x86互換のコプロセッサ（~ 60 core）
  - 既存のコードから容易に拡張可能
  - SIMD/MIMD
  - 最新のランキングでは、TOP20に残らず
- PEZY-SC
  - 日本のベンチャー企業が設計
  - メニーコアプロセッサ（1024個）
  - MIMD
  - C/C++しかコンパイラがないようです



エクサFLOPS・メガW時代へ

- 電力消費量はこれ以上増やせないので、これから専用CPUと組み合わせたスパコンが国内でも増えてくる
- 汎用／専用CPU構成のヘテロジニアスなシステムへ
- →ユーザのプログラム負担が増える可能性
- シミュレーション研究者の宿命だが、5-10年くらいの周期でスパコンシステムのトレンドに振り回される
  - ベクトル vs. スーパースカラ
  - MPI vs. HPF (High Performance Fortran)
  - 私は2009年に手持ちのコード（MHD/PIC）をスクラッチから再コーディング
- スパコン情勢に注意しつつ、研究を進めましょう

まとめ

- スカラチューニング
  - 高速化のためのCPUの機能（SIMD）をいかに使い倒すか
  - キャッシュチューニング
- OpenMPによるスレッド並列化
  - 指示行を最外ループの手前にいれるだけ（簡単！）
  - スレッド並列化によりプロセス数を減らし、プロセス間通信のオーバーヘッドを軽減：ハイブリッド並列化
- 今後の展望
  - 次世代のスパコンでは電力消費量問題が顕在化
  - 汎用／専用CPUで構成されるヘテロジニアスシステムに
  - →ハイブリッド並列化はますます必須

参考資料

- プロセッサを支える技術、Hisa Ando著、技術評論社
- 各スパコンマニュアル
- http://www.nag-j.co.jp/openMP/index.htm
- http://accc.riken.jp/hpc/training/