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
千葉大学大学院理学研究院

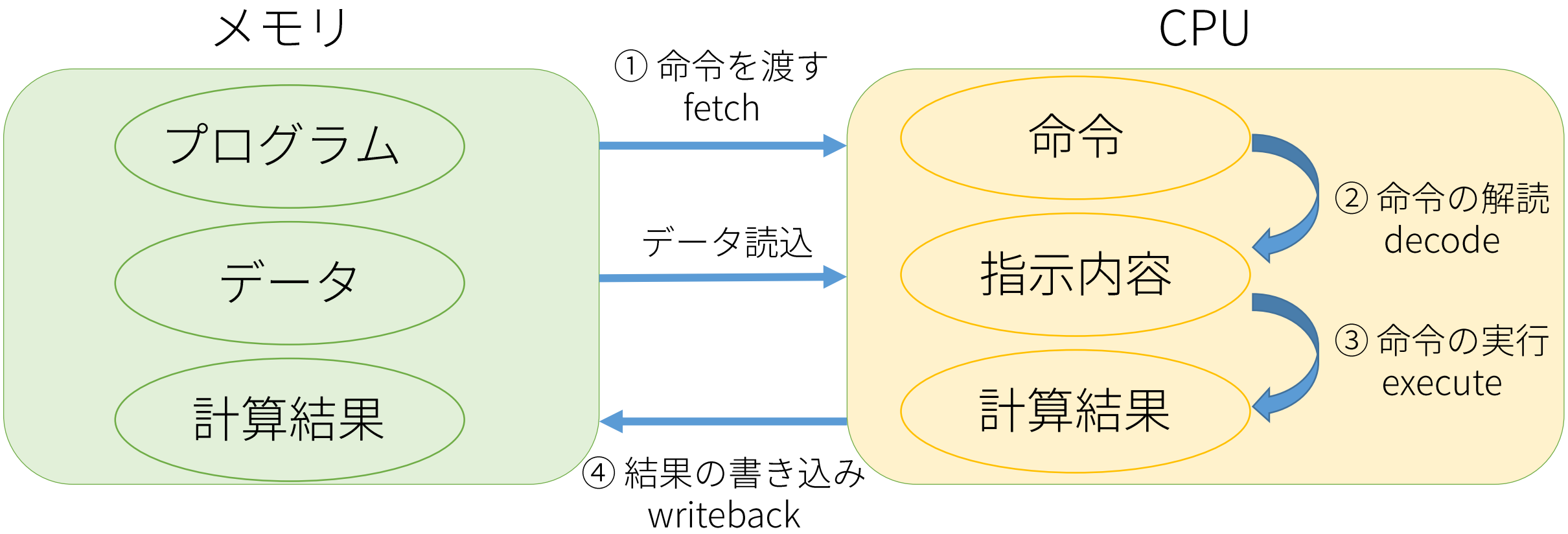
謝辞
C言語への対応：簑島敬（JAMSTEC）

内容

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

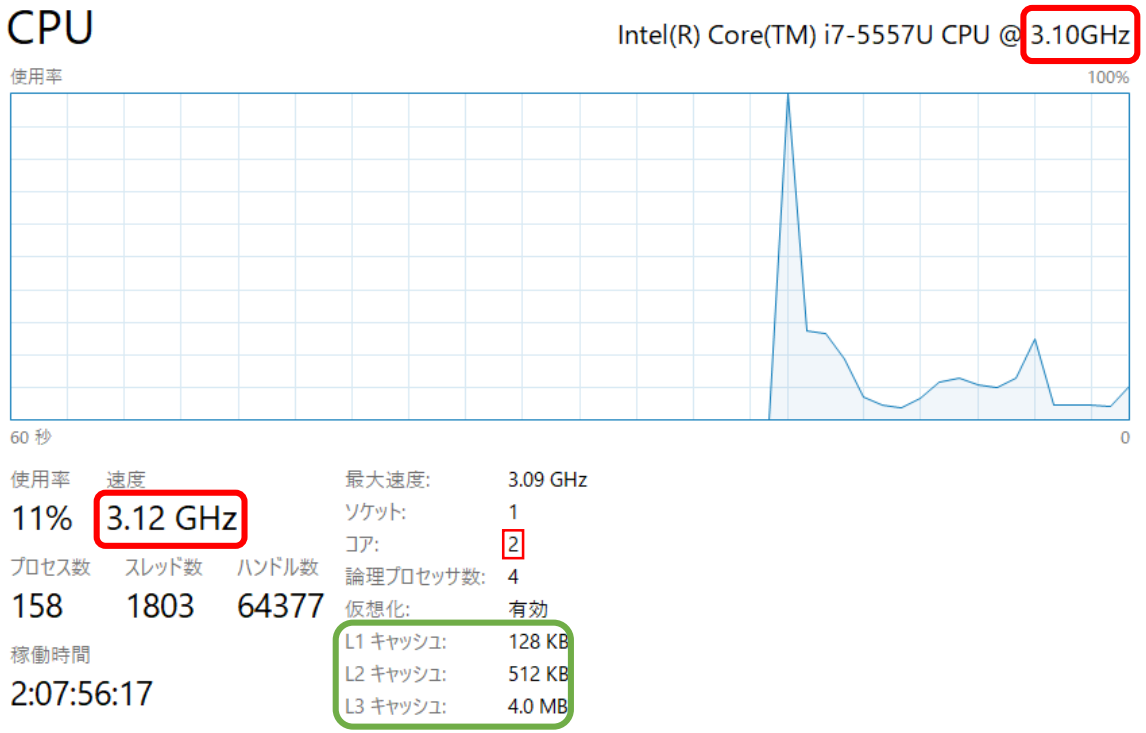
イントロダクション

CPUが命令を実行する流れ



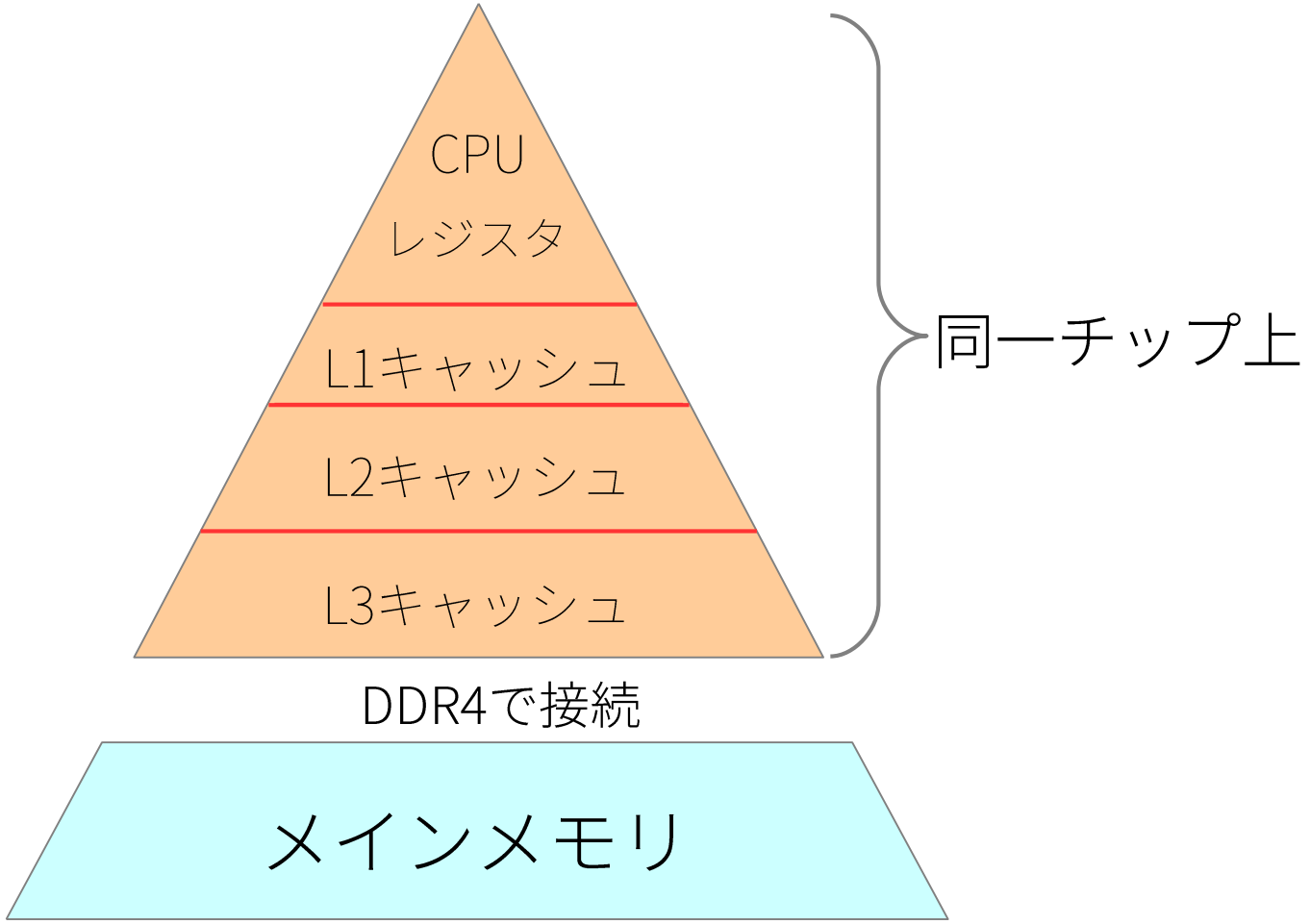
計算に時間がかかるところは、
1. 命令の実行（②、③）
2. データの読み込み（①、④）

CPUの命令処理性能



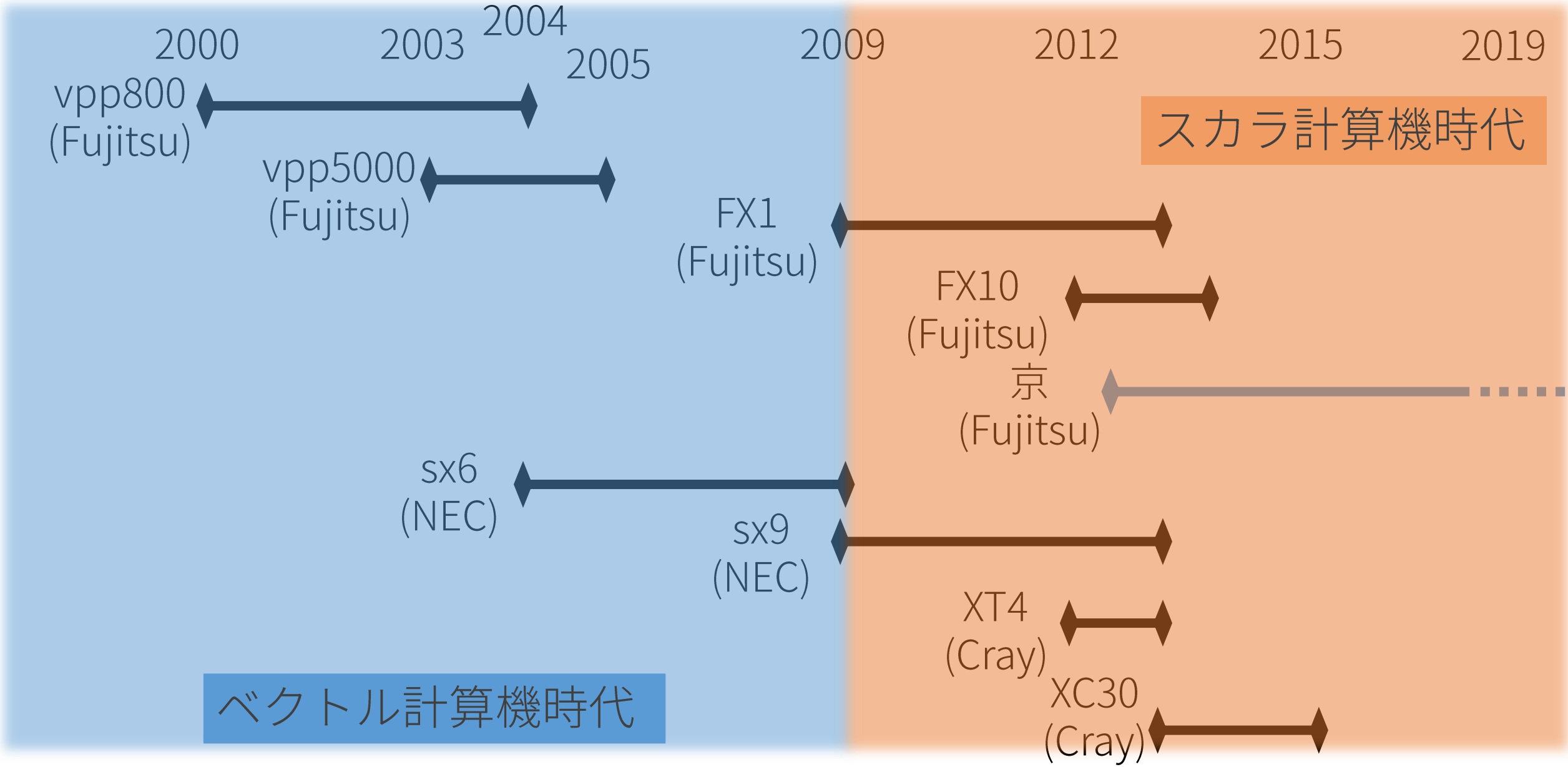
- 3.1 GHz : 1秒間に3.1 x 10^9 回の処理（①,…,④）を実行できる。処理単位をクロック（サイクル）という
- FLOPS: 1秒間に浮動小数点演算（足し算、掛け算など）をできる回数
- 最近のIntel CPUは、クロックあたり16FLOPS
- CPUの性能の指標として、例えば、16 x 3.1GHz x 2 core = 96 GFLOPS
- 緑枠はキャッシュの情報（次ページ）

メモリの階層構造



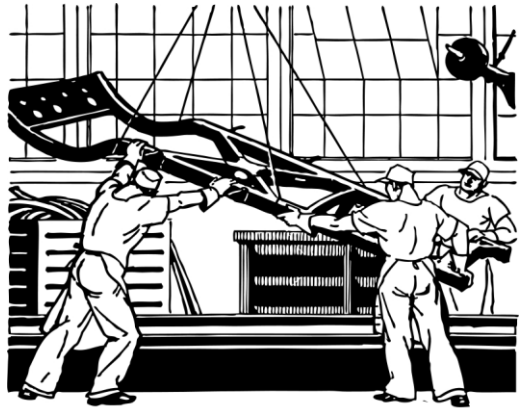
- CPUに近いほど高速低容量
- レジスタ, 32bit / 64bit、1クロック
- L1$ 数10 kB, 数クロック
- L2$ ～MB, > 100GB/s
- メモリ数10GB, 数10GB/sec
- CPUが計算をするときは、まず近いところから必要なデータを探す。なければメモリから取り出す。 →キャッシュミス

私のスパコン利用歴


2000
2003
2004
2005
2009
2012
2015
2019
vpp800
(Fujitsu)
vpp5000
(Fujitsu)
スカラ計算機時代
FX1
(Fujitsu)
FX10
(Fujitsu)
京
(Fujitsu)
sx6
(NEC)
sx9
(NEC)
XT4
(Cray)
XC30
(Cray)
ベクトル計算機時代

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
```

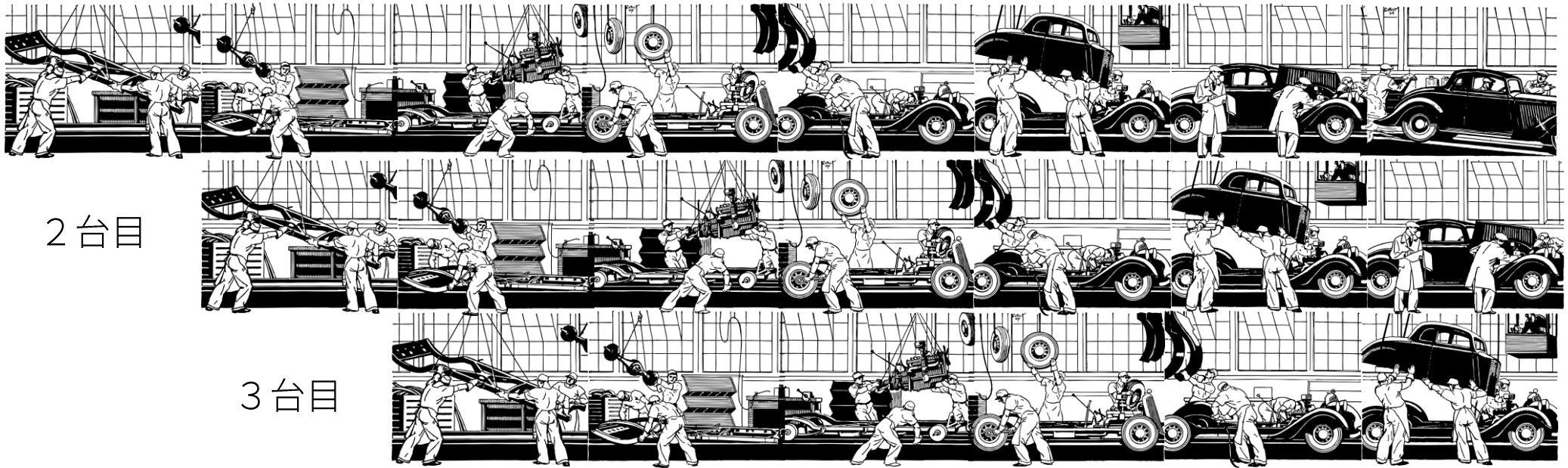
ベクトル（データのパイプライン処理）

時間

1台目

2台目

3台目



これまでのCPU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（→～3GHzで頭打ち）
- ベクトル化（複数のデータを同時に同じ計算すること）
  - ➢パイプライン処理
  - ➢SIMD（複数レジスタと演算器）

  } ユーザーから見たら同じ
- メモリ構造の階層化（キャッシュの有効利用）
- 並列化（マルチコア）

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

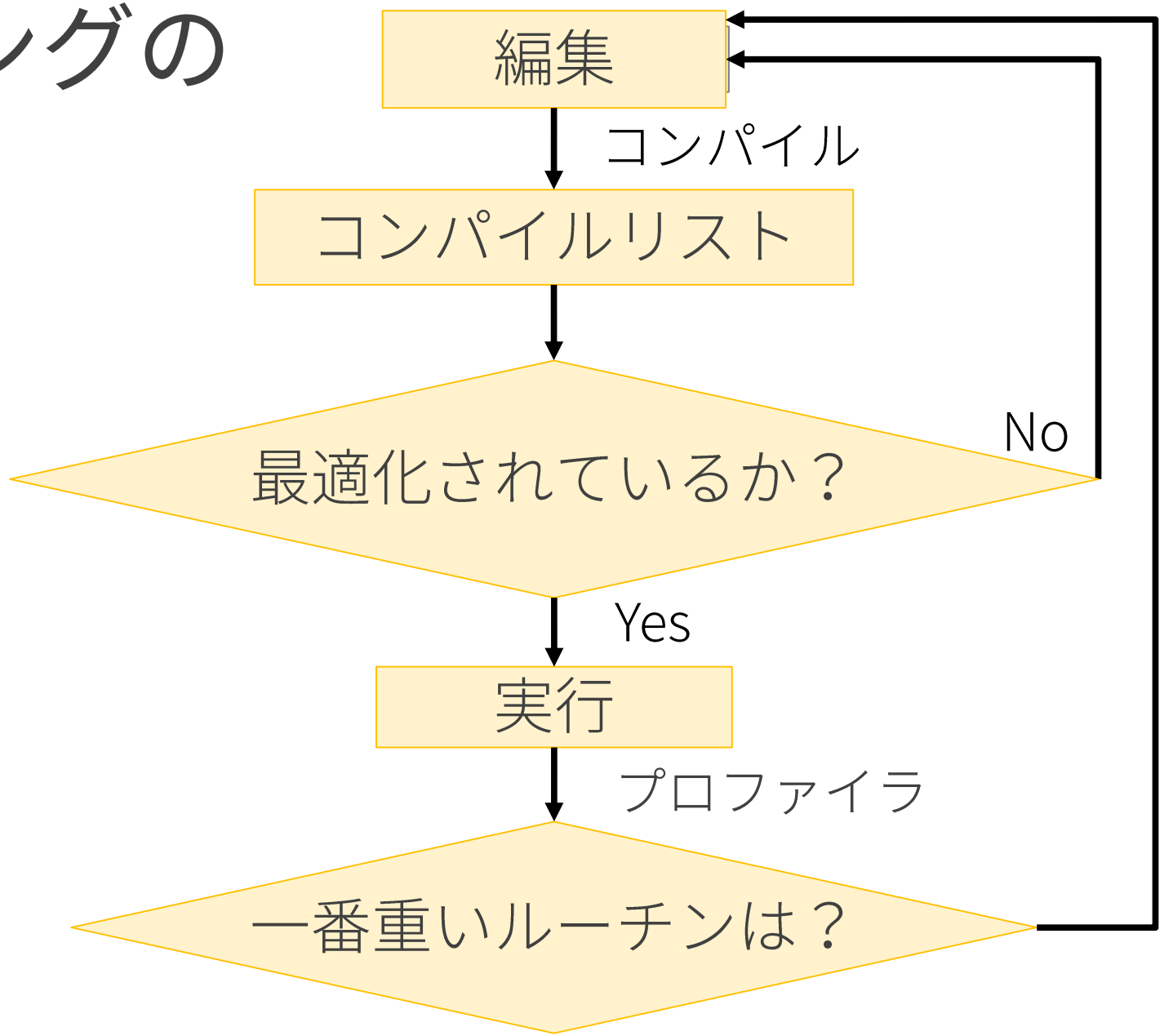
注意

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
- スーパーコンピュータ「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: -qopt-report-phase={all, vec, par,openmp}

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の中身

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（なるべくCPUの近くにデータを置く）

分岐処理の回避例

例１

```
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
```

例２

```
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
```

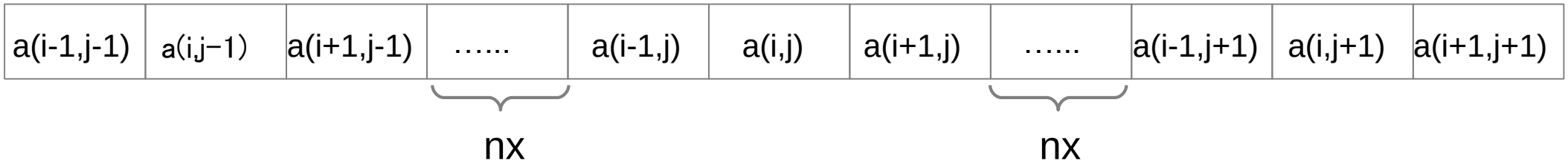


```
do i=1,nx
 c(i) = sign(1.0,a(i))             &
       *max(0.0,                   &
            min(abs(a(i)),         &
            sign(1.0,a(i))*b(i)) &
           )
enddo
```

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、

nx　　nx

```
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 $\rho^{n+1} = \mathrm{f}(\rho^n, V_x^n, V_y^n)$

次のステップに進むためには、自分自身（ρ）の他に
速度場（V_x, V_y）が必要

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

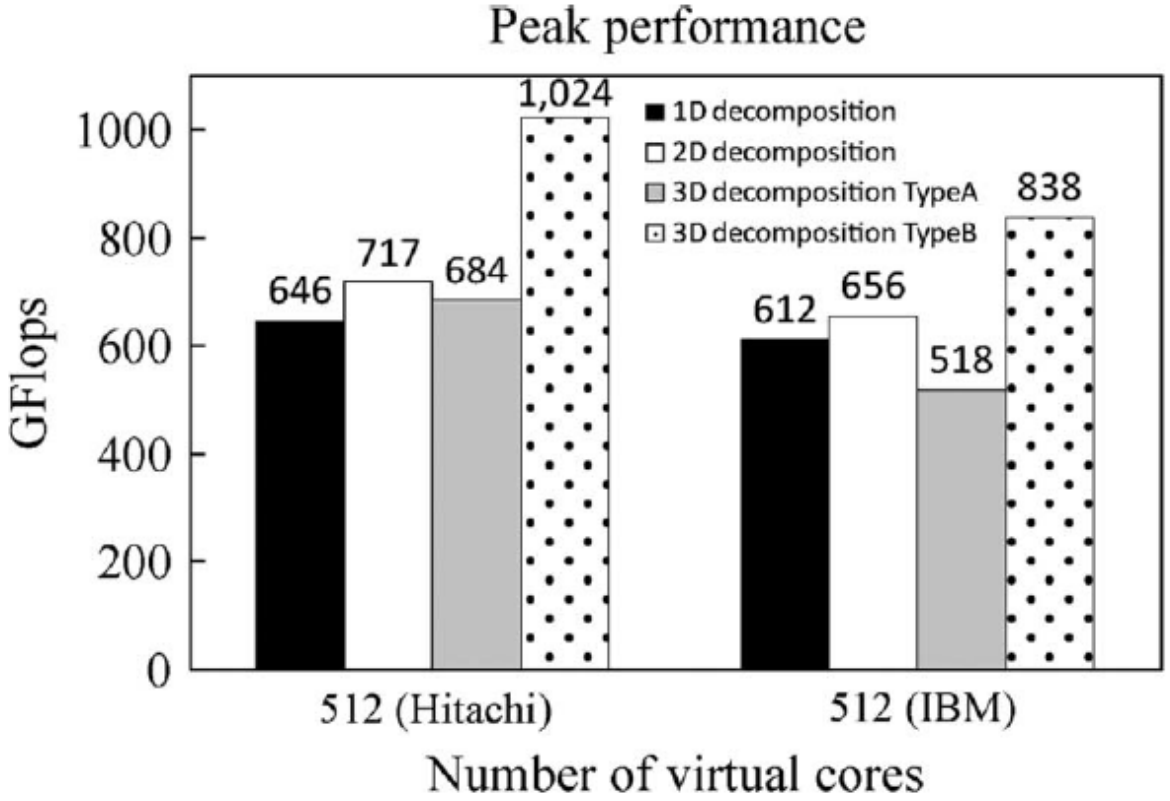
と変数を個別に用意する代わりに、

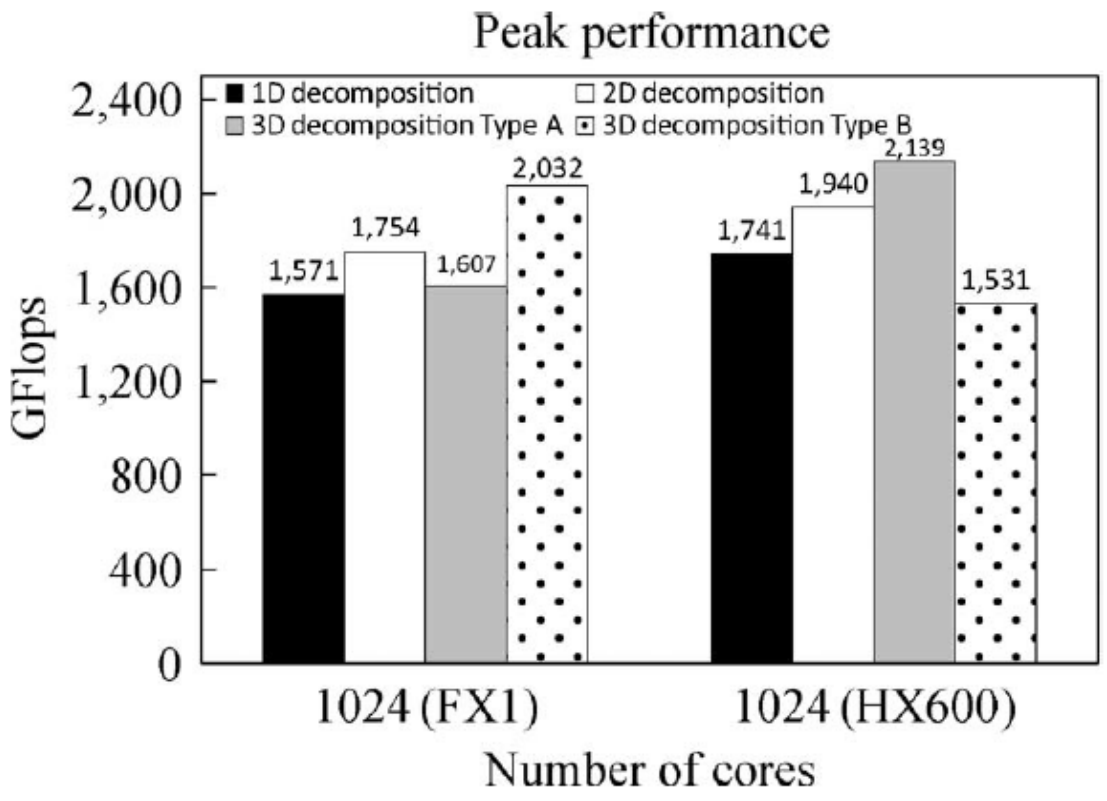
```
dimension f(8,nx,ny) ! 1:rho, 2:p, 3-5:v, 6-8:B
```

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)





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

近年のキャッシュ重視型のスパコンにおいて効果的

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;
   }
}
```

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
```

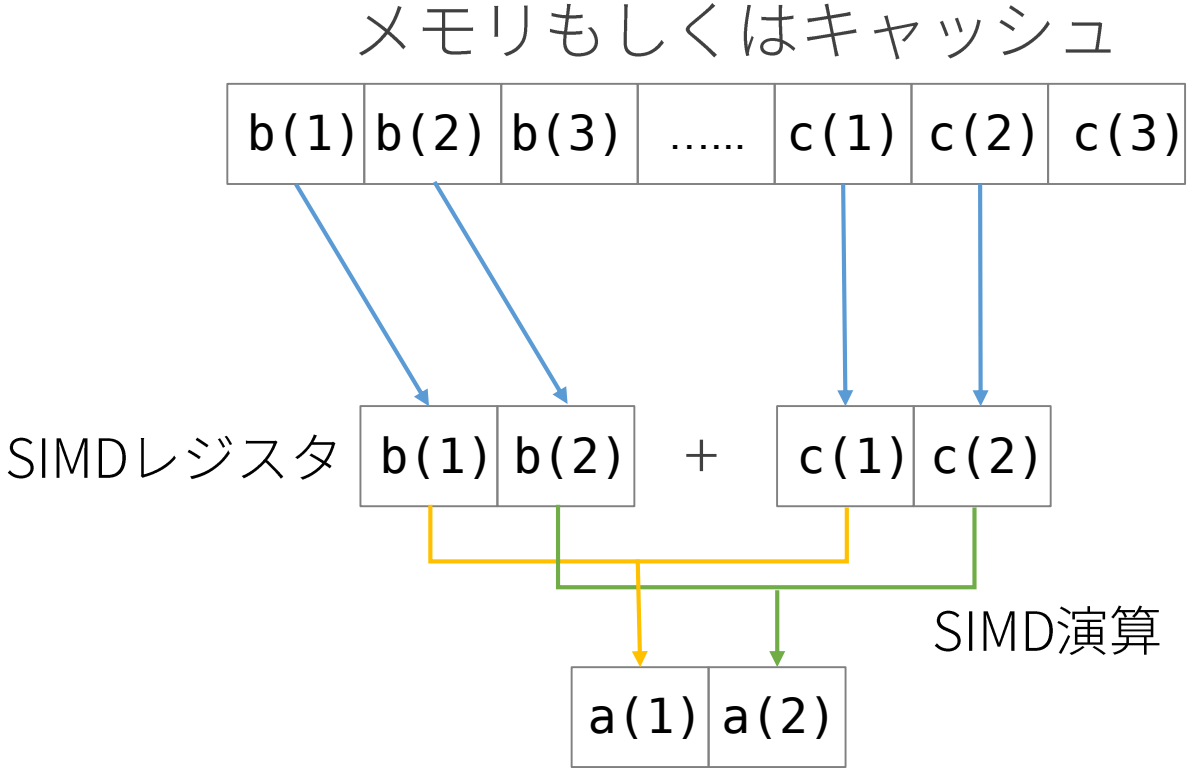
のように、ループ内で繰り返し呼び出す場合、呼び出しのオーバーヘッドが大きい。関数内の手続きが短い場合は、内容をその場所に展開する→インライン展開

- 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

SIMD: Single Instruction Multiple Data

- 同じ計算を複数のデータに対して一括して処理する
- 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2～16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
- コンパイルオプションで最適化
  - gcc/gfortran: -m{avx, sse4}
  - icc/ifort: -x{avx,sse4}

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



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
```

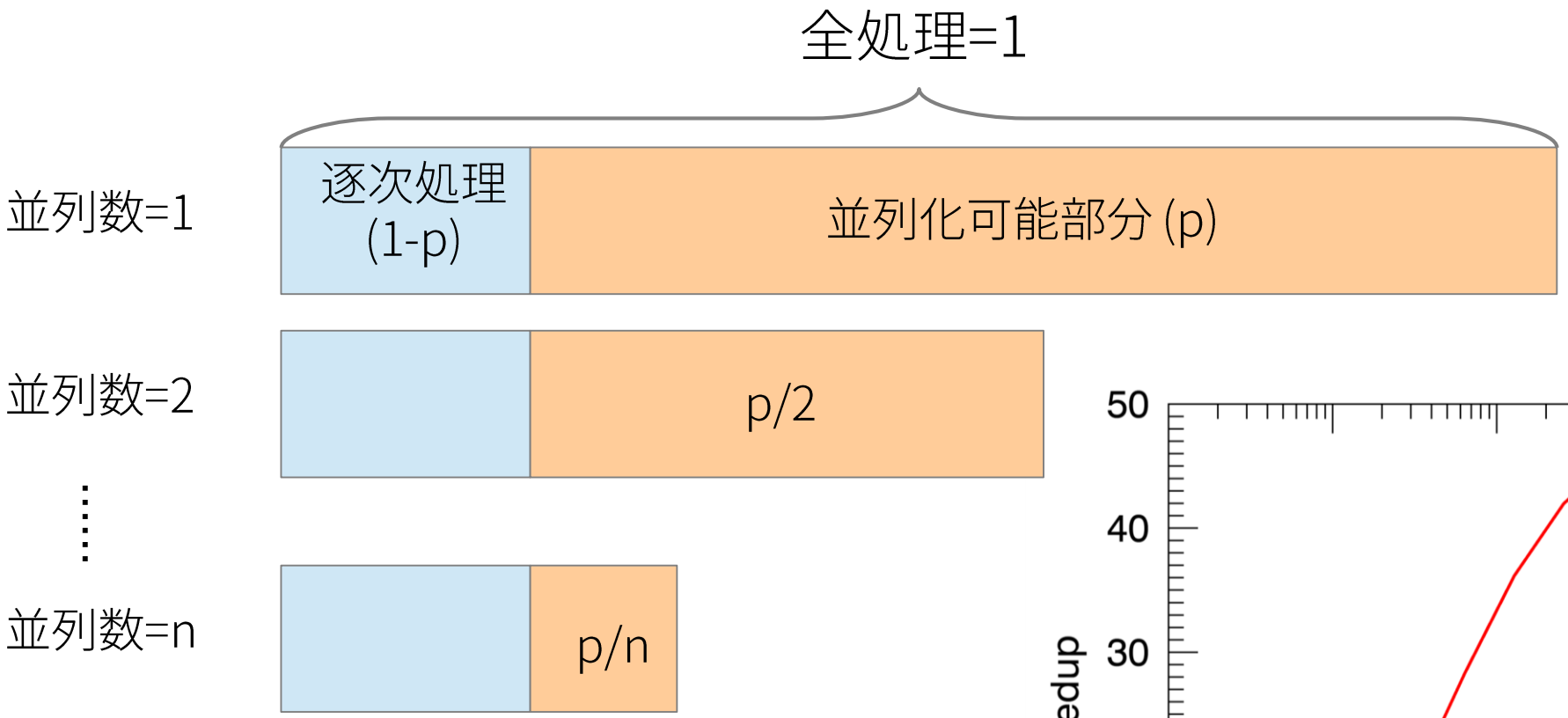
例２: ループ番号によって処理が異なる場合

```
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
```

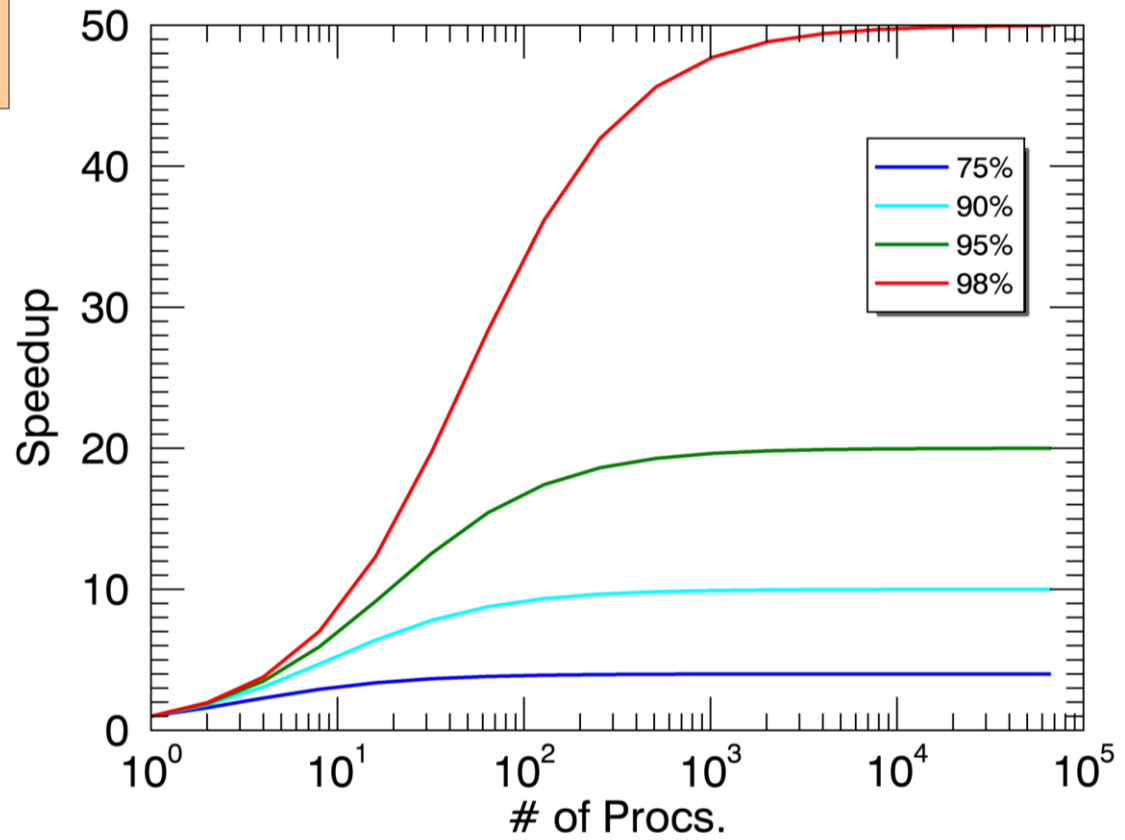
OpenMPによるコードの並列化

アムダールの法則



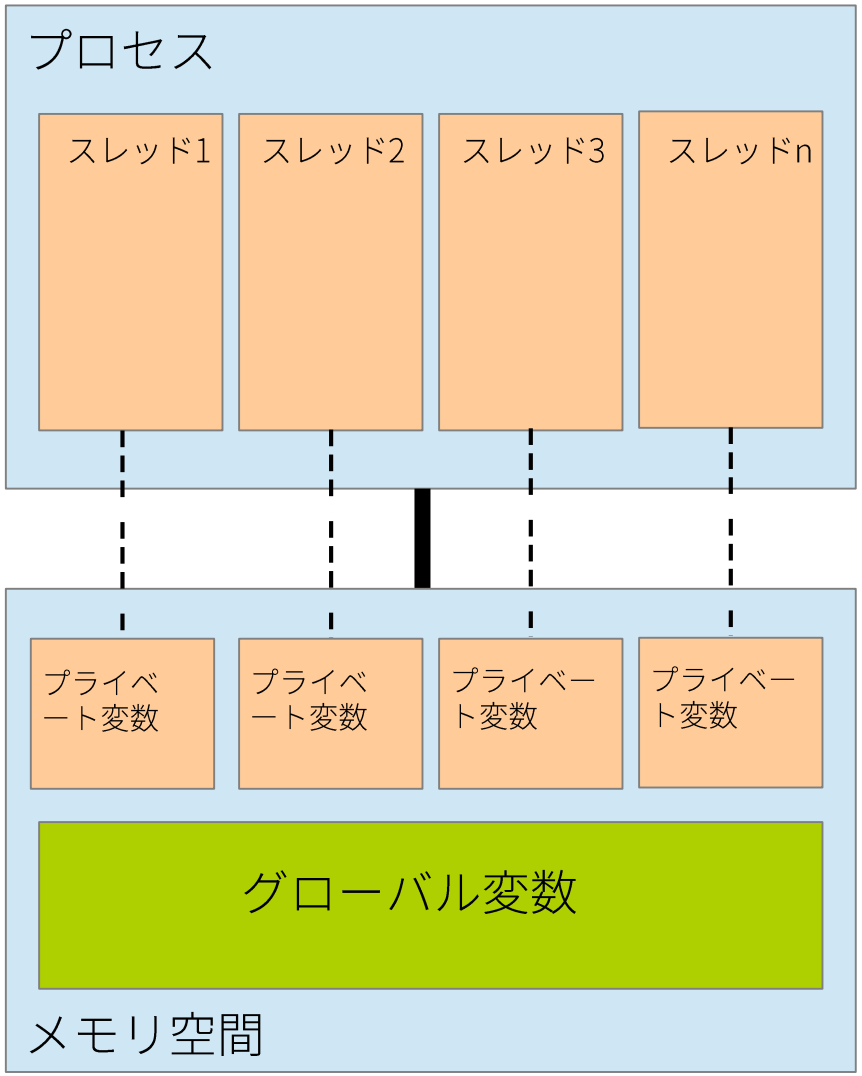
$$\text{性能向上率} = \frac{1}{(1-p)+\frac{p}{n}}$$

少なくも並列化率p>0.99である必要あり

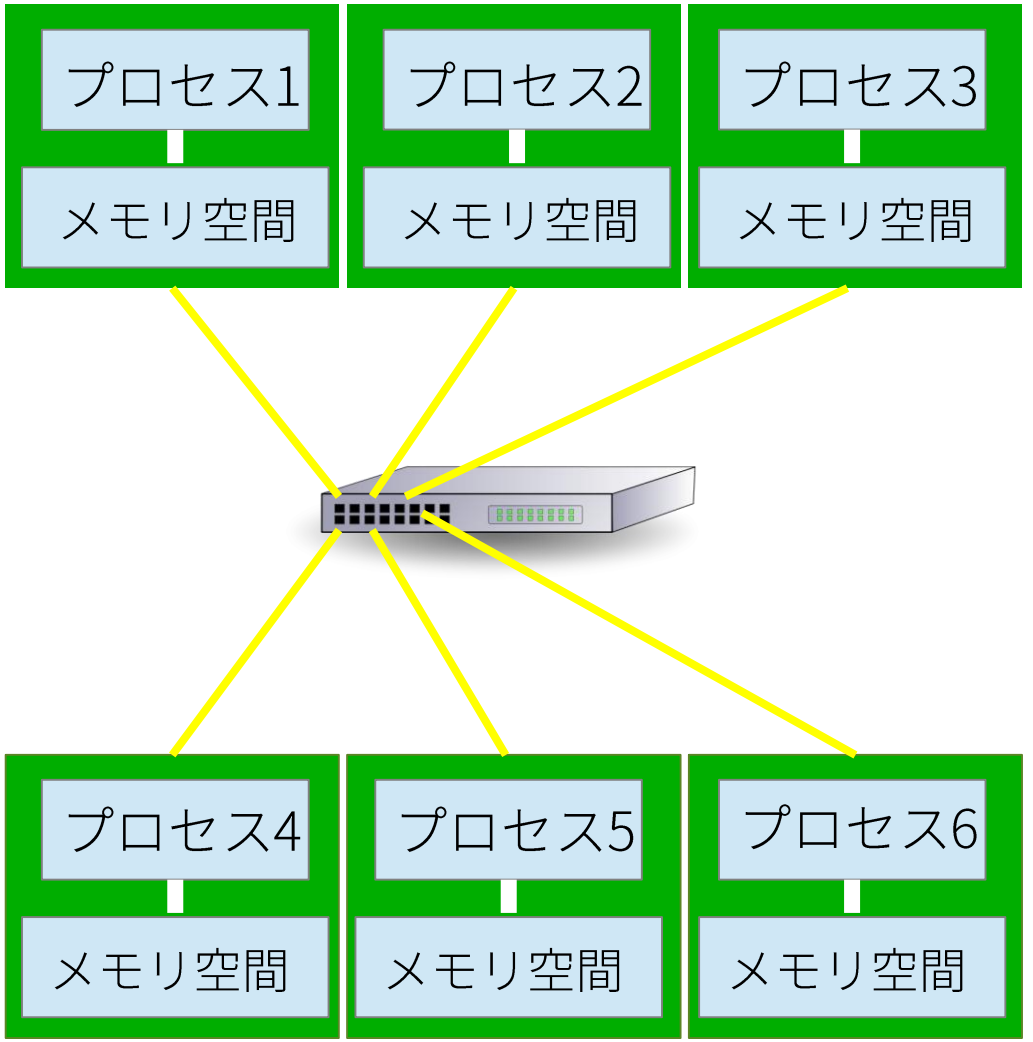


スレッド並列とプロセス並列

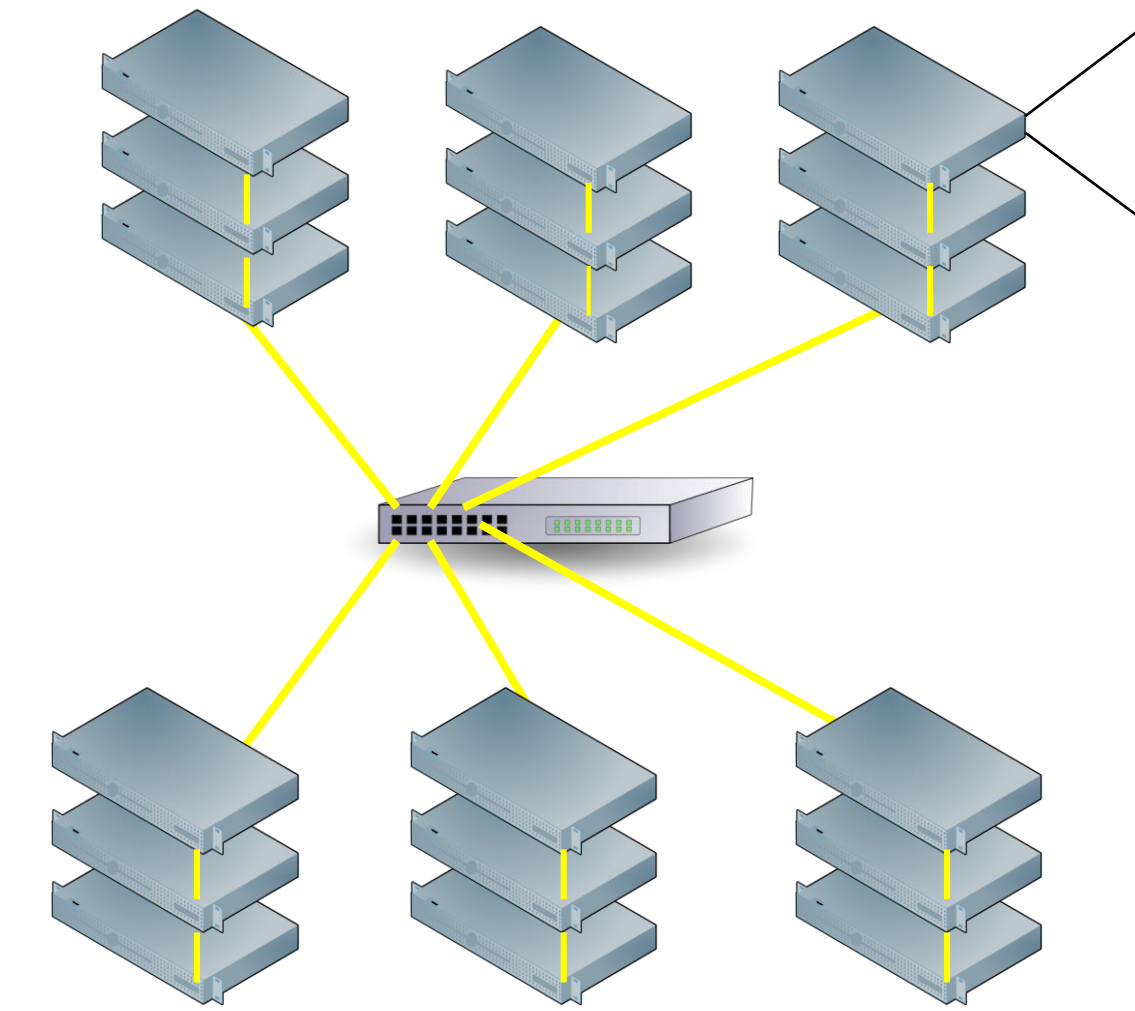
スレッド並列

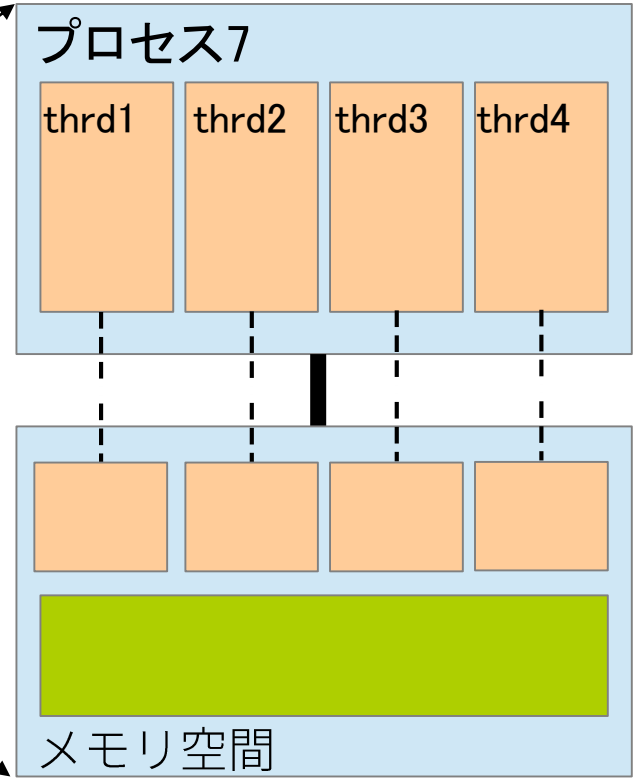


プロセス並列



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: -qopenmp
- 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
- 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
  - tcsh: setenv OMP_NUM_THREADS 8
  - bash: export 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）

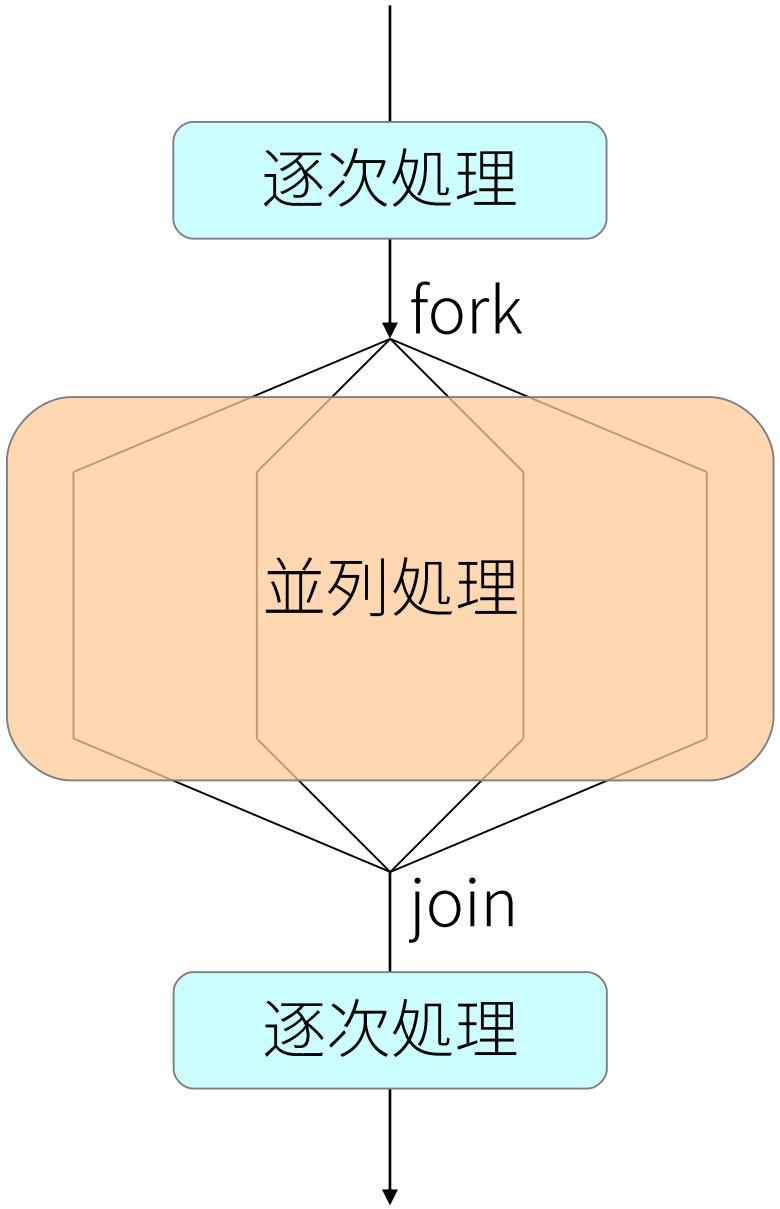
Fortran:

```
!$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
```

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

```
#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];
  }
 }
```

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

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

スレッド間でtmpを上書きしてしまうので正しい結果が得られない

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

Cの場合はループ中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の一回だけ

途中で逐次処理が入る場合はSINGLEで対処

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。

```
#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];
 }
}
```

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];
 }
}
```

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

2019年8月現在

- アメリカがTOP1に返り咲き
- 電力消費は1 - 20MWatt
- ＡＩ専用スパコン@産総研が8位
- ほとんどがNVIDIA GPUによる加速演算器つき

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**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
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
7	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
8	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
9	**SuperMUC-NG** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	
10	**Lassen** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100 , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	288,288	18,200.0	23,047.2	

The GREEN 500

- 最新のGPU（NVIDIA）がTOP10のほとんどを占める。
- PEZY Computing社によるシステムがランクから消えた（いろいろあって、、、）
- 日本からは東工大のTSUBAME3.0
- 15GFLOPS/Wに到達

Rank	TOP500 Rank	System	Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
1	469	**DGX SaturnV Volta** - NVIDIA DGX-1 Volta36, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla V100 , Nvidia NVIDIA Corporation United States	22,440	1,070.0	97	15.113
2	1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	10,096	14.719
3	8	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	1,649	14.423
4	393	**MareNostrum P9 CTE** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100 , IBM Barcelona Supercomputing Center Spain	18,360	1,145.0	81	14.131
5	25	**TSUBAME3.0** - SGI ICE XA, IP139-SXM2, Xeon E5-2680v4 14C 2.4GHz, Intel Omni-Path, NVIDIA Tesla P100 SXM2 , HPE GSIC Center, Tokyo Institute of Technology Japan	135,828	8,125.0	792	13.704
6	11	**PANGEA III** - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100 , IBM Total Exploration Production France	291,024	17,860.0	1,367	13.065
7	2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	7,438	12.723

スーパーコンピュータ「富岳」

- 次世代フラッグシップスーパーコンピュータ
- 2021年より運用開始予定
- CPU: Arm, 48cores, 512bitSIMD, 2.7TFLOPS
- Memory: HBM2, <1GB/core (32GiB/CPU)
- B/F=0.37 : GPU並みのメモリアクセス速度
- ~400 PFLOPS
- 30-40MWatt (~15GFLOPS/W)
- アプリ性能比（対「京」）で30倍
- 予想される CPU数: 400PFLOPS/2.7TFLOPS ~ 1.5×10^5 (~7×10^6cores)

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
- 汎用／専用CPU構成のヘテロジニアスなシステムが多い中、「富岳」の低消費電力CPUが登場
- 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と省電力CPUを搭載した「富岳」の登場
  - →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