

計算科学演習B

OpenMP 発展

深沢圭一郎

(京都大学学術情報メディアセンター)



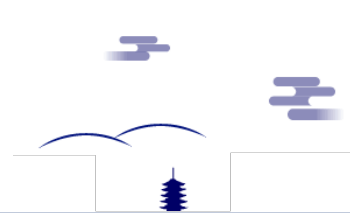
OpenMP発展の前に……

問い： 計算の速度は何で決まる？

- CPU(コア)のクロック？
- キャッシュの容量？
- メモリの性能？

**正解： 計算の種類に依存する
(一概にいけない)**

大規模数値シミュレーション、特に離散化シミュレーションでは
メモリの性能(キャッシュの構成や性能も含む)に影響されやすい



メモリの性能: バンド幅とレイテンシ

バンド幅: 単位時間あたりにどれだけのデータ量を転送できるか

レイテンシ: データを要求してから最初のデータが届くまでの時間

→ ネットワークと同様

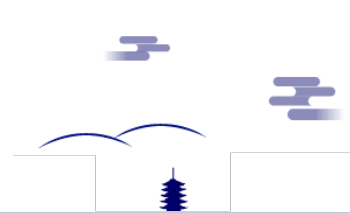


共有メモリ型並列計算機

複数のプロセッサとメモリモジュール

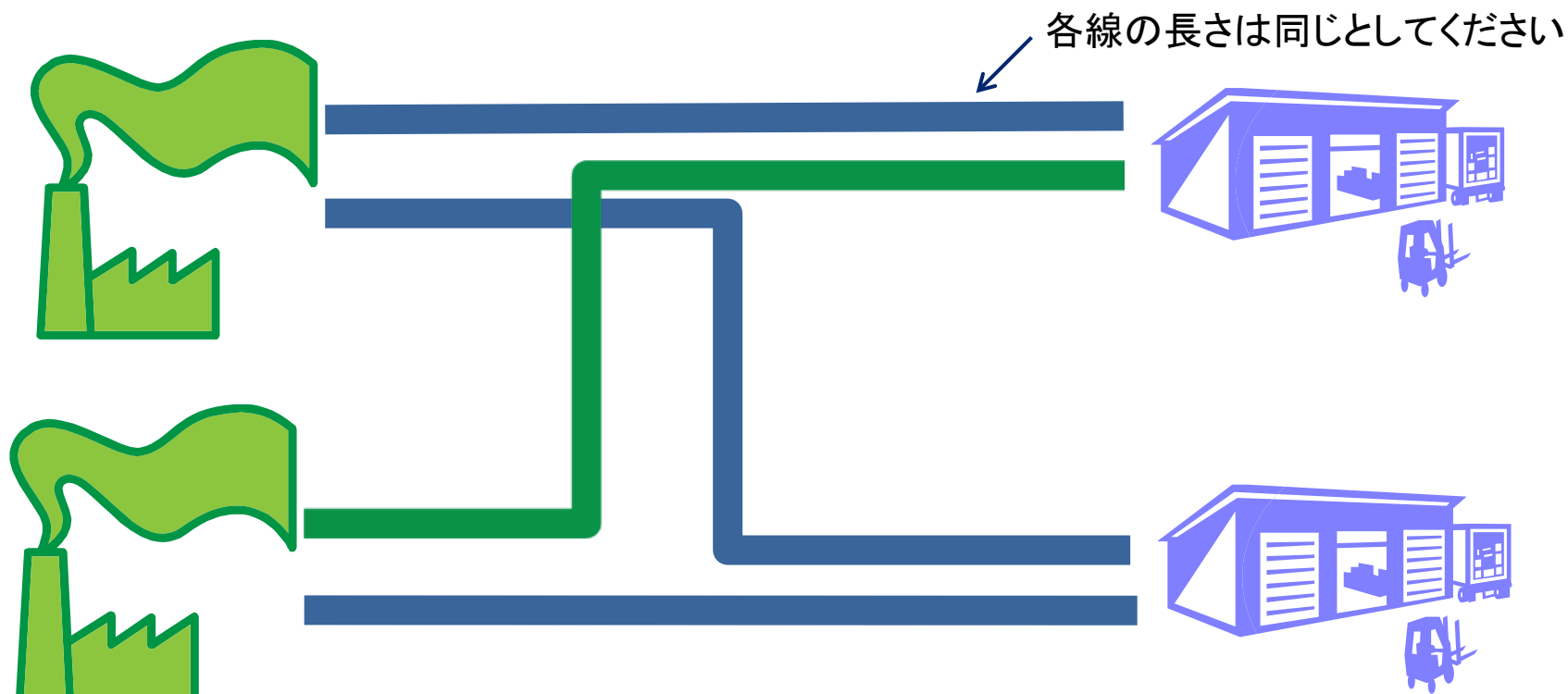


いろいろなプロセッサとメモリの構成がある



共有メモリ型並列計算機

複数のプロセッサとメモリモジュール



UMA(Uniform memory access)

CPUからみてどのメモリモジュール上の
データも均等にアクセスできる



NUMAアーキテクチャ

Non-uniform memory access (NUMA)



京大サブシステムBとCでも採用

多くのPCクラスタ型計算機がこの作りになっている

CPUからみた各メモリモジュールに対するアクセスが
不均一



京大センターのシステム

3種類の演算サーバがある

サブシステムA (Camphor 2)

- Cray XC40 (Xeon Phi KNL)

サブシステムB (Laurel2)

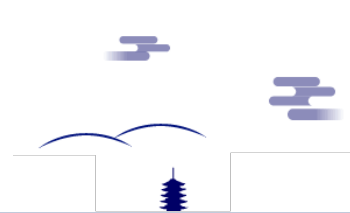
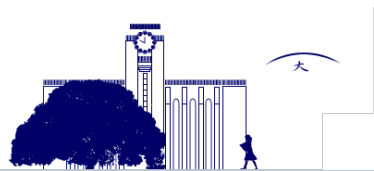
- Cray CS400 (Broadwell Xeon)

サブシステムC (Cinnamon2)

- Cray CS400 (Haswell Xeon)



NUMA



CS400(サブシステムB)

ノード構成

Intel Broadwell Xeon E5系 (2.1 GHz, 18 core) $\times 2 = 1.21$ TFlops

Memory: DDR4-2400, 128 GB, 153.6 GB/sec (メモリチャンネル $\times 4$)

ノード間ネットワーク

Intel OmniPath (12.1 GB/sec)

トータル性能

850ノード = 1.03 PFlops

OS

Red Hat Linux EL 7



NUMAアーキテクチャ

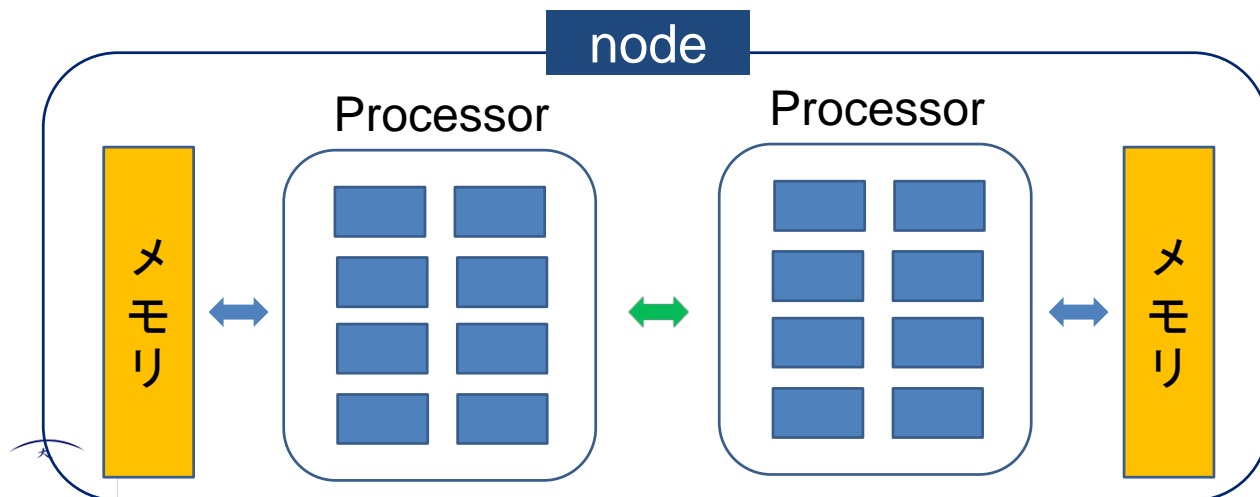
サブシステムBとCで採用

(Aでもあるが特殊なので今回は考えない)

コアからみた物理メモリがノード内でも不均一



メモリインテンシブな計算において、演算を行うコアの近くのメモリに配置する必要がある



メモリアンテンシブな計算(1)

例えば、こんな計算を考えてみる

```
integer, parameter :: n=10**7
double precision :: a(n), b(n)
!$OMP PARALLEL DO
do i=1,n
    b(i)=a(i)*2.0
enddo
```

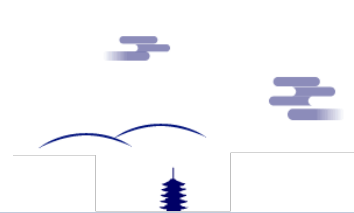
```
#define n 10000000
double a[n], b[n];
#pragma omp parallel for
for(i=0;i<n;i++) {
    b[i]=a[i]*2.0;
}
```

ロード/ストア:2回、演算:1回

Byte per flop: 16 ($8 \times 2 \div 1$)

1コアの理論性能: 33.6 GFlops

1ソケットのメモリバンド幅: 76.8 GB/s



メモリーインテンシブな計算(2)

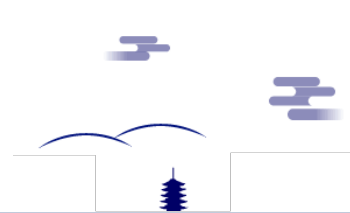
旧サブシステムB上での計算結果

スレッド数(コア数)	計算時間 (msec)
1	2130
2	1311
4	927
16	931

現サブシステムB上での計算結果

スレッド数(コア数)	計算時間 (msec)
1	29.4
2	15.4
4	13.0
8	11.1
16	8.6
36	14.2

計算は単純で並列化可能であるが、
よい台数効果、特に2ソケット利用では
良い性能が得られていない。



ファーストタッチ(1)

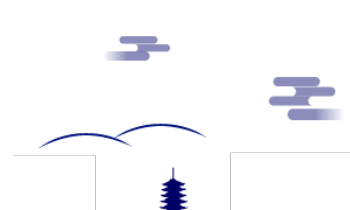
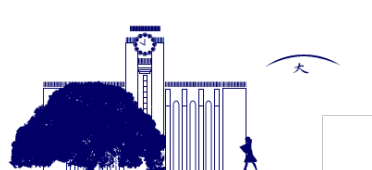
スレッドのあるコアに近いメモリにそのスレッドが使用するデータを配置する。

配列は宣言時には、物理メモリ上には確保されない。

配列にデータを格納した時点で物理的なメモリ上の場所が確定する。

データを使うスレッドにより当該データを初期化する

 **ファーストタッチ**



ファーストタッチ(2)

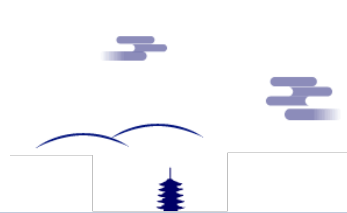
```
integer, parameter :: n=10**7
!$OMP PARALLEL DO
do i=1,n
    a(i)=1d0
    b(i)=0d0
enddo
!$OMP END PARALLEL DO
```

```
!$OMP PARALLEL DO
do i=1,n
    b(i)=a(i)*2.0
enddo
```

```
#define n 10000000
#pragma omp parallel for
for (i=0;i<n;i++) {
    a[i]=1;
    b[i]=0;
}
```

この部分がファーストタッチ

```
#pragma omp parallel for
for (i=0;i<n;i++) {
    b[i]=a[i]*2.0;
}
```



ファーストタッチの効果

旧サブシステムB上での計算結果

スレッド数(コア数)	計算時間(msec)FT無	計算時間(msec)FT有
1	2130	2058
2	1311	1062
4	927	555
16	931	331

現サブシステムB上での計算結果

スレッド数(コア数)	計算時間(msec)FT無	計算時間(msec)FT有
1	29.4	8.8
2	15.4	4.9
4	13.0	3.1
8	11.1	2.9
16	8.6	2.6
36	14.2	1.4

疎行列・ベクトル積ベンチマーク

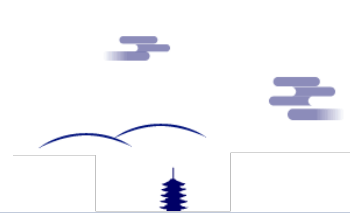
この計算は有限要素解析等の離散化解法において頻繁に用いられる

行列データは一回の行列・ベクトル積演算で一度しか使われない

データサイズが大きい場合、キャッシュの有効利用が期待できない
メモリスループットにより計算速度が決まる

- 行列中の非ゼロ要素位置によってはそれ以下の性能となる場合があるが、今回は考えない。

CS400(システムB)上でメモリアクセスチューニングの効果を検証してみる



通常のプログラム(オリジナルコード)

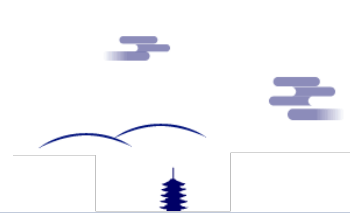
```
!$OMP DO PRIVATE(i,j)
```

```
do i=1,n  
  do j=ihead(i),ihead(i+1)-1  
    bvec(i)=bvec(i)+a(j)*x(icol(j))  
  enddo  
enddo
```

```
#pragma omp for private(i,j)
```

```
for (i=0;i<n;i++) {  
  for (j=ihead[i];j<ihead[i+1];j++) {  
    bvec[i]=bvec[i]+a[j]*x[icol[j]];  
  }  
}
```

行列ベクトル積の並列化は
一般的に簡単とされているが...



通常のプログラム(オリジナルコード)

```
!$OMP DO PRIVATE(i,j)
```

```
do i=1,n
```

```
  do j=ihead(i),ihead(i+1)-1
```

```
    bvec(i)=bvec(i)+a(j)*x(icol(j))
```

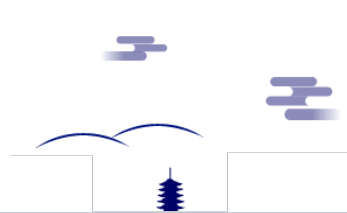
```
  enddo
```

```
enddo
```

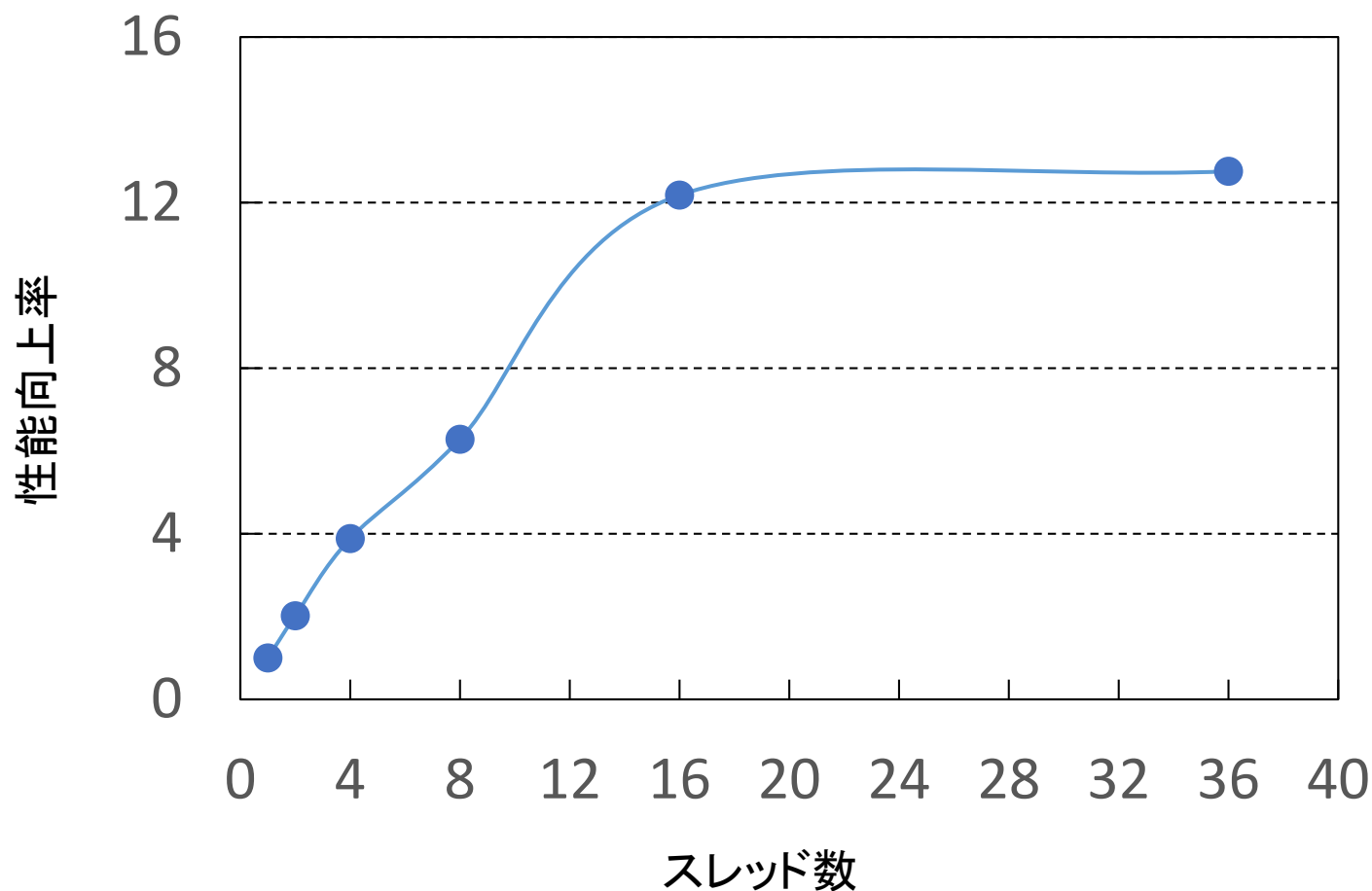
間接参照される配列xについては、配列要素に対する複数のスレッドによるアクセスが避けがたい。

※実際はそんなコード書いては駄目です

配列a, bvecについてはファーストタッチの効果が期待できる。



1コア時の経過時間を基準とした台数効果



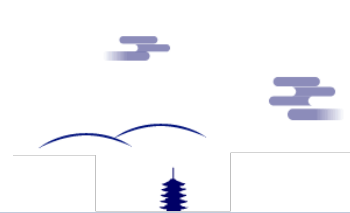
並列処理による効果があまり得られていない

メモリアクセスキューニング

ファーストタッチ

(再掲ですが)プログラムにおいて配列を宣言した時点では、物理的なメモリ上に領域は確保されない

各スレッドが計算において扱う配列の領域を事前にタッチ(ダミーの値を格納→初期化)する



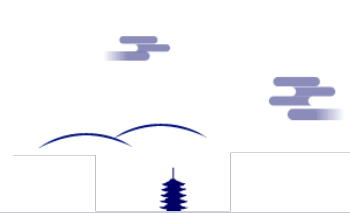
ファーストタッチ

```
!$OMP DO
do i=1,n
    bvec(i)=0d0
    ihead(i)=0
    x(i)=0d0
enddo
!$OMP DO
do j=1,nzero
    a(j)=0d0
    icol(j)=0
enddo
```

疎行列・ベクトル積のデータ
を読み込む配列に対して、
左のプログラムを実行する



配列要素のうち、各スレッド
が初期化した部分は当該の
スレッドのあるコアの近くの
メモリに配置される



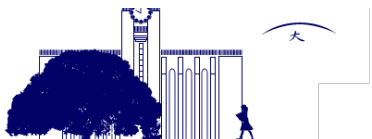
ファーストタッチ

```
#pragma omp for
for(i=0;i<n;i++) {
    bvec[i]=0.0;
    ihead[i]=0;
    x[i]=0.0;
}
#pragma omp for
for(j=0;j<nzero;j++){
    a[j]=0.0;
    icol[j]=0;
}
```

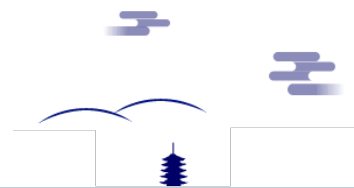
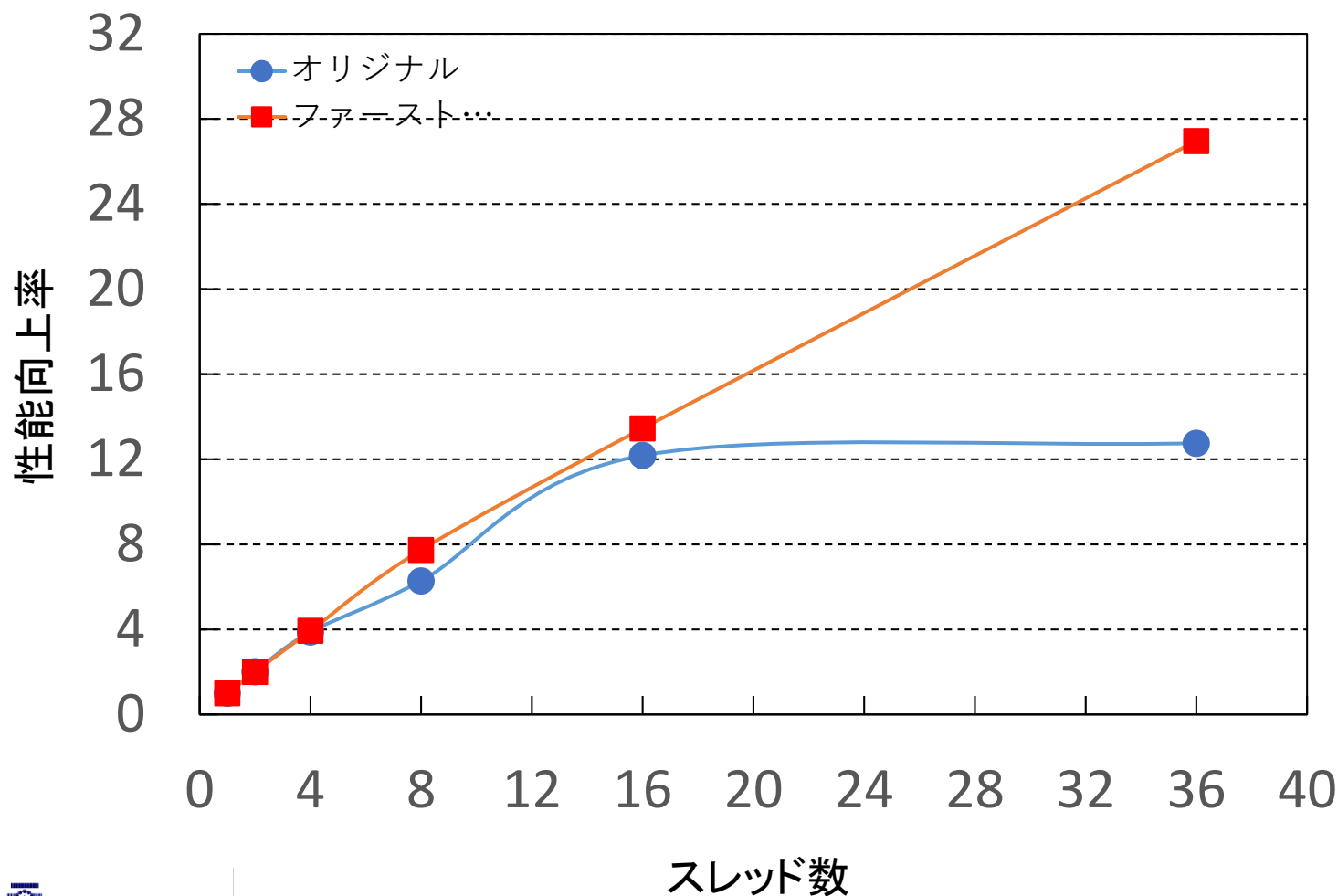
疎行列・ベクトル積のデータを読み込む配列に対して、左のプログラムを実行する



配列要素のうち、各スレッドが初期化した部分は当該のスレッドのあるコアの近くのメモリに配置される



1コア時の経過時間を基準とした台数効果



NUMAチューニングに関するまとめ

ノード内に複数のプロセッサ(ソケット)がある場合、メモリアクセスが多いプログラムでは、データを適切に配置する必要がある。



NUMAメモリアーキテクチャでは、各スレッドの近くの物理メモリに当該スレッドが使用するデータを置くことが重要。

→ファーストタッチにより、データ配置を行う。numactlのようなコマンドによっても可能。

ファーストタッチをしない通常の疎行列ベクトル積のコードでは、スレッド並列化をしても、十分な性能が得られない。

具体的な応用プログラムでは、複数スレッドが同一の配列要素、変数にアクセスする場合があるが、出来る範囲でファーストタッチを行う方がよい。

より高度な並列処理(MPI的なプログラム)

omp_get_num_threads関数: 総使用スレッド数を得る
omp_get_thread_num関数: スレッドIDを得る
→これらを利用してomp doを使わず計算領域を自分で分割。

```
use omp_lib
integer :: omp_get_thread_num, omp_get_num_threads
!$OMP PARALLEL
  numprocs=omp_get_num_threads()
  myid=omp_get_thread_num()
!
  if (myid.eq.0) then
    . . .
  elseif (myid.eq.1) then
    . . .
  elseif (myid.eq.2) then
    . . .
!$OMP END PARALLEL
```



より高度な並列処理(MPI的なプログラム)

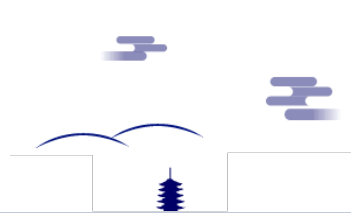
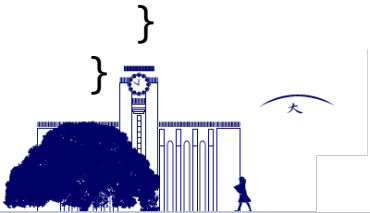
omp_get_num_threads関数 総使用スレッド数を得る

omp_get_thread_num関数 スレッドIDを得る

→これらを利用してomp forを使わず計算領域を自分で分割。

```
#include <omp.h>
int nthreads, myid, xstart, xend;

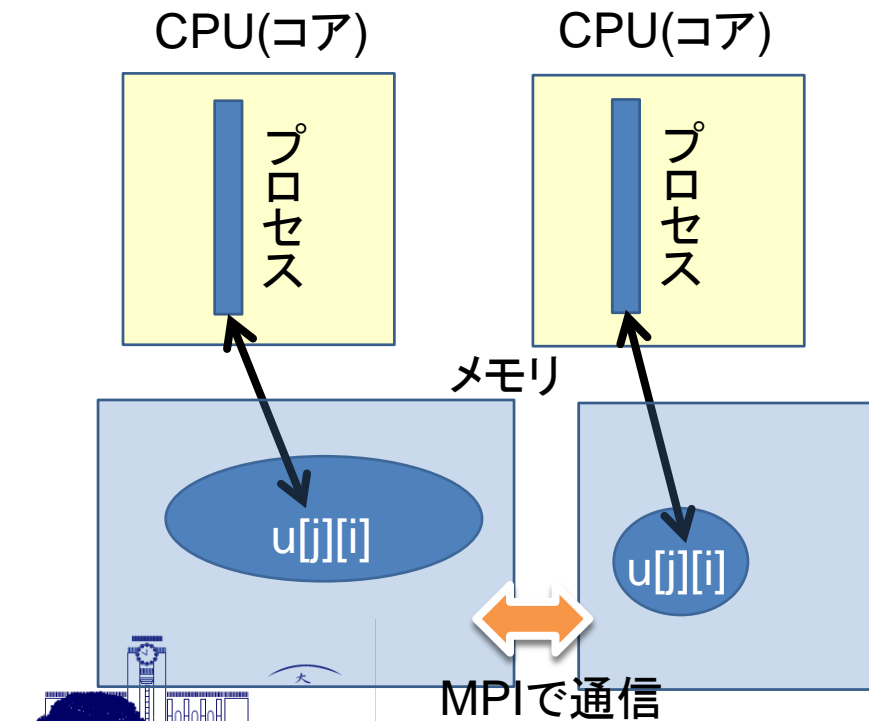
#pragma omp parallel private(nthreads, myid, xstart, xend){
    nthreads=omp_get_num_threads();
    myid=omp_get_thread_num();
    xstart=NX/nthreads*myid+1
    xend=NX/nthreads*(myid+1)
    for(x=xstart;x<xend;x++) {
        for(y=0;y<NY+1;y++) {
            a[x][y]=...;
        }
    }
}
```



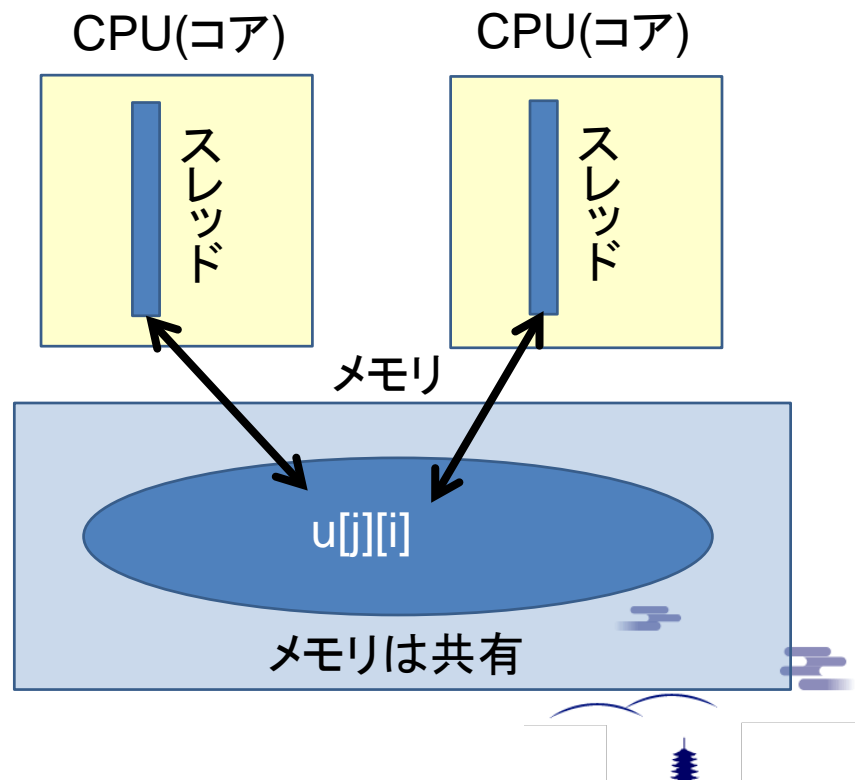
MPI的に書く場合の注意

MPIとOpenMPの違いを考えて、MPI的なプログラムを書く
→OpenMPでは**配列が共有で疎通信は要らない**

プロセス並列



スレッド並列



プロセス/スレッド併用型並列処理

MPIのライブラリの実装に依存する部分がある

MPIによるプログラムがあるとして

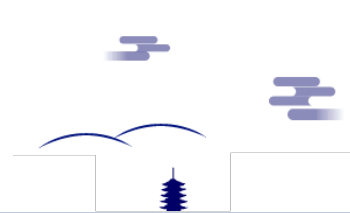
MPI+自動並列→もっとも簡単

MPI+OpenMP(複合Parallel work-sharing構文)

- doループやforループを複合Parallel work-sharing構文で並列化
→割と簡単で安全

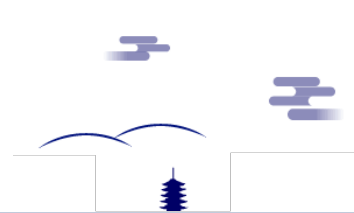
MPI+OpenMP (OpenMPのスレッド並列処理をフル活用)

- 以下の方法が多くのシステム(京大スパコンを含む)で利用可能なやり方
MPIライブラリの通信関数はマスタスレッドからのみ呼び出す
送受信するデータはSharedのデータであること



MPI/OpenMP併用並列処理 (1)

```
integer, parameter :: n=10000
integer :: i
real(kind=8) :: a(0:n-1),b(0:n-1)
do i=0,n-1
    a(i)=dble(i)
enddo
do i=0,n-1
    b(i)=dsin(a(i))
enddo
```

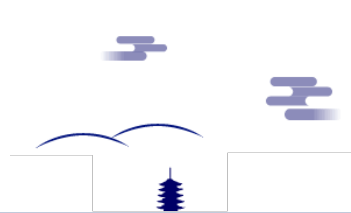


MPI/OpenMP併用並列処理 (2)

```
include "mpif.h"
integer, parameter :: n=10000
integer :: i,myid,ieer,nump,,it
integer , allocatable::ir(:),id(:)
real(kind=8) :: a(0:n-1),b(0:n-1)
    call MPI_INIT(ieer)
    call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ieer)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,nump,ieer)
    allocate(ir(0:nump-1),id(0:nump-1))
    it=n/nump
    do i=it*(myid),min(n-1, it*(myid+1)-1)
        a(i)=dble(i)
    enddo
    do i=it*(myid),min(n-1, it*(myid+1)-1)
        b(i)=dsin(a(i))
    enddo
```


```
do i=0,nump-1
    id(i)=it*i
enddo
do i=0,nump-2
    ir(i)=it
enddo
ir(nump-1)=n - it*(nump-1)
call MPI_ALLGATHERV(b(it*(myid)), &
& ir(myid), MPI_DOUBLE_PRECISION, &
& b(0), ir(0), id(0), &
& MPI_DOUBLE_PRECISION, &
& MPI_COMM_WORLD, ieer)

call MPI_FINALIZE(ieer)
```



MPI/OpenMP併用並列処理 (3)

```
include "mpif.h"
integer, parameter :: n=10000
integer :: i,myid,ieer,nump,,it
integer , allocatable::ir(:),id(:)
double precision :: a(0:n-1),b(0:n-1)
call MPI_INIT(ieer)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ieer)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nump,ieer)
allocate(ir(0:nump-1),id(0:nump-1))
it=n/nump
!$OMP PARALLEL DO
do i=it*(myid),min(n-1, it*(myid+1)-1)
a(i)=dble(i)
enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
do i=it*(myid),min(n-1, it*(myid+1)-1)
b(i)=dsin(a(i))
enddo
!$OMP END PARALLEL DO
```



```
do i=0,nump-1
id(i)=it*i
enddo

do i=0,nump-2
ir(i)=it
Enddo
ir(nump-1)=n - it*(nump-1)

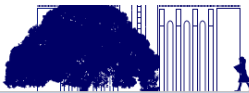
call MPI_ALLGATHERV(b(it*(myid)), &
& ir(myid), MPI_DOUBLE_PRECISION, &
& b(0), ir(0), id(0), &
& MPI_DOUBLE_PRECISION, &
& MPI_COMM_WORLD, ieer)

call MPI_FINALIZE(ieer)
```

MPI/OpenMP併用並列処理 (4)

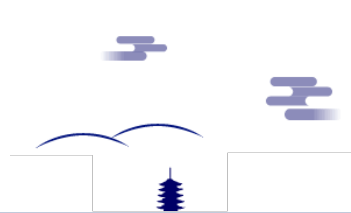
```
include "mpif.h"
integer, parameter :: n=10000
integer :: i,myid,ieer,nump,,it
integer , allocatable::ir(:),id(:)
double precision :: a(0:n-1),b(0:n-1)
    call MPI_INIT(ieer)
    call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ieer)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,nump,ieer)
    allocate(ir(0:nump-1),id(0:nump-1))
    it=n/nump
!$OMP PARALLEL PRIVATE(i)
!$OMP DO
    do i=it*(myid),min(n-1, it*(myid+1)-1)
        a(i)=float(I)
    enddo
!$OMP DO
    do i=it*(myid),min(n-1, it*(myid+1)-1)
        b(i)=dsin(a(i))
    enddo
```

```
!$OMP MASTER
    do i=0,nump-1
        id(i)=it*i
    enddo
    do i=0,nump-2
        ir(i)=it
    enddo
    ir(nump-1)=n - it*(nump-1)
    call MPI_ALLGATHERV(b(it*(myid)), &
        & ir(myid), MPI_DOUBLE_PRECISION, &
        & b(0), ir(0), id(0), &
        & MPI_DOUBLE_PRECISION, &
        & MPI_COMM_WORLD, ieer)
!$OMP END MASTER
!$OMP BARRIER
!$OMP END PARALLEL
call MPI_FINALIZE(ieer)
```



MPI/OpenMP併用並列処理 (1)

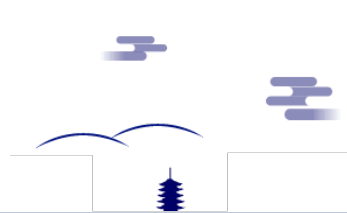
```
#define n 10000
#include <math.h>
int i;
double a[n],b[n];
for(i=0;i<n;i++) {
    a[i]=(double)i;
}
for(i=0;i<n;i++) {
    b[i]=sin(a[i]);
}
```



MPI/OpenMP併用並列処理 (2)

```
#include <mpif.h>
#include <math.h>
#define n 10000
int main(int argc, char **argv) {
    int i,myid,ierr,nump,*ir,*id,it, tail;
    double a[n],b[n];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&nump;;
    it=n/nump;
    tail=it*(myid+1);
    if (tail>n) tail=n;
    for(i=it*(myid); i<tail;i++) a[i]=(double)i;
    for(i=it*(myid); i<tail;i++) b[i]=sin(a[i]);
```

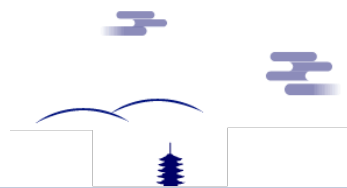
```
    for (i=0;i<nump;i++) id[i]=it*i;
    for (i=0;i<nump-1;i++) ir[i]=it;
    ir[nump-1]=n - it*(nump-1);
    MPI_Allgatherv(&b[it*[myid]],
        ir[myid], MPI_DOUBLE, b,
        ir, id, MPI_DOUBLE,
        MPI_COMM_WORLD);
    MPI_Finalize();
}
```



MPI/OpenMP并用並列処理 (3)

```
#include <mpif.h>
#include <math.h>
#define: n 10000
int main(int argc, char **argv) {
    int: i,myid,ieer,nump,*ir,*id,it, tail;
    double a[n],b[n];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&nump;;
    it=n/nump;
    tail=it*(myid+1);
    if (tail>n) tail=n;
    #pragma omp parallel for
        for(i=it*(myid); i<tail;i++) a[i]=(double)i;
    #pragma omp parallel for
        for(i=it*(myid); i<tail;i++) b[i]=sin(a[i]);
```

```
for (i=0;i<nump;i++) id[i]=it*i;
for (i=0;i<nump-1;i++) ir[i]=it;
ir[nump-1]=n - it*(nump-1);
MPI_Allgatherv(&b[it*[myid]],
    ir[myid], MPI_DOUBLE, b, ir, id,
    MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Finalize();
}
```



MPI/OpenMP并用並列処理 (4)

```
#include <mpif.h>
#include <math.h>
#define: n 10000
int main(int argc, char **argv) {
    int: i,myid,ieer,nump,*ir,*id,it, tail;
    double a[n],b[n];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&nump;
    it=n/nump;
    tail=it*(myid+1);
    if (tail>n) tail=n;
    #pragma omp parallel {
    #pragma omp for
        for(i=it*(myid); i<tail;i++) a[i]=(double)i;
    #pragma omp for
        for(i=it*(myid); i<tail;i++) b[i]=sin(a[i]);
```

```
#pragma omp master {
    for (i=0;i<nump;i++) id[i]=it*i;
    for (i=0;i<nump-1;i++) ir[i]=it;
    ir[nump-1]=n - it*(nump-1);
    MPI_Allgatherv(&b[it*[myid]],
        ir[myid], MPI_DOUBLE, b, ir,
        id, MPI_DOUBLE, MPI_COMM_WORLD);
    } // end master
#pragma omp barrier
} // end parallel
MPI_Finalize();
}
```

