第４回　並列計算法

筑波大学 University of Tsukuba

大規模並列計算

”大規模”問題とは？

自由度が多い問題 (計算時間が長く，記憶容量が必要な問題)

分散メモリ並列計算機の利用

より自由度が多い問題 (大規模問題) ：広域・高解像度

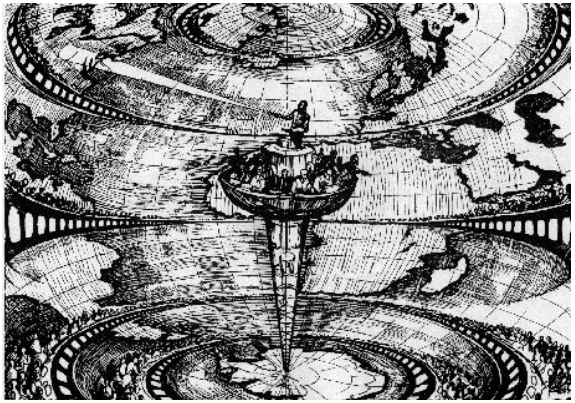
より多くの計算機の利用 (大規模計算機システム)

大規模なデータを大規模なシステムで処理すること

筑波大学 University of Tsukuba

リチャードソンの夢

Lewis Fry Richardson(1881-1953，数学者、気象学者)



気象予報の原型を提案
第１次世界大戦のデータを使用し、
6 時間後を予測
--> 手計算で２ヶ月かかる。

“64000 人の計算者を巨大ホールに集め、指揮者に従い整然と計算を行えば、
実際の天候の変化と同じぐらいの速さで予報が行える。”
Richardson, L.F. : Weather Prediction by Numerical Process, University Press, 1922.

⇒ 並列計算の基本概念

筑波大学 University of Tsukuba

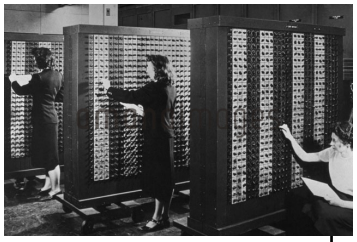
コンピュータの開発

John von Neumann (1903-1957，数学者)

von Neumann architecture ;
CPU(中央演算装置) とアドレス付けされた記憶装置、それらをつなぐ Bus で構成される。
=> 記憶装置に保存されたプログラムを実行する　(Stored-program computer)
– First Draft of a Report on the EDVAC, 1945 --

* 「博士の異常な愛情」の Dr. Strangelove のモデルの一人と言われている。

ENIAC(Electronic Numerical Integrator and Computer), 1946
- 世界初の電子汎用コンピュータ
- 弾道計算を目的にしているが、マンハッタン計画の Neumann がこの計算機に深く関わる
- John W. Mauchly, John P. Eckert により開発
- EDVAC へ引き継がれる
- 真空管を使用



筑波大学 University of Tsukuba

コンピュータの開発

UNIVAC I (UNIVersal Automatic Computer), 1950

- John W. Mauchly, John P. Eckert により開発
- 世界最初の商用コンピュータ
- 真空管の本数は ENIAC の 3 分の 1 程度
- 磁気テープ搭載
- プログラム内蔵方式

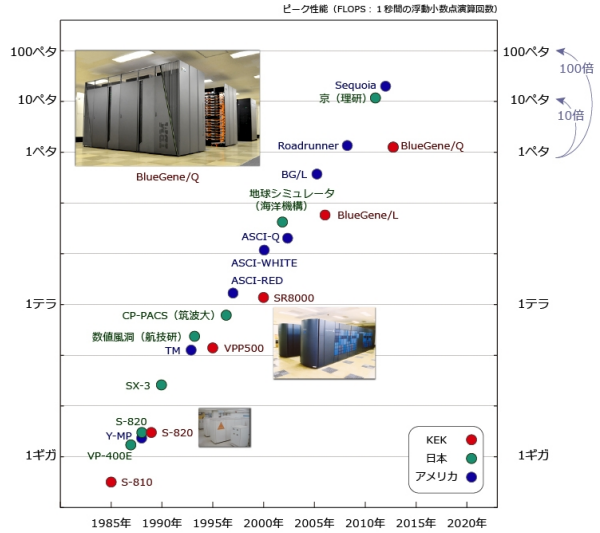
IBM 1401, 1959

- トランジスタ集積回路
- 搭載ソフトウェア
  - FORTRAN II, COBOL ...

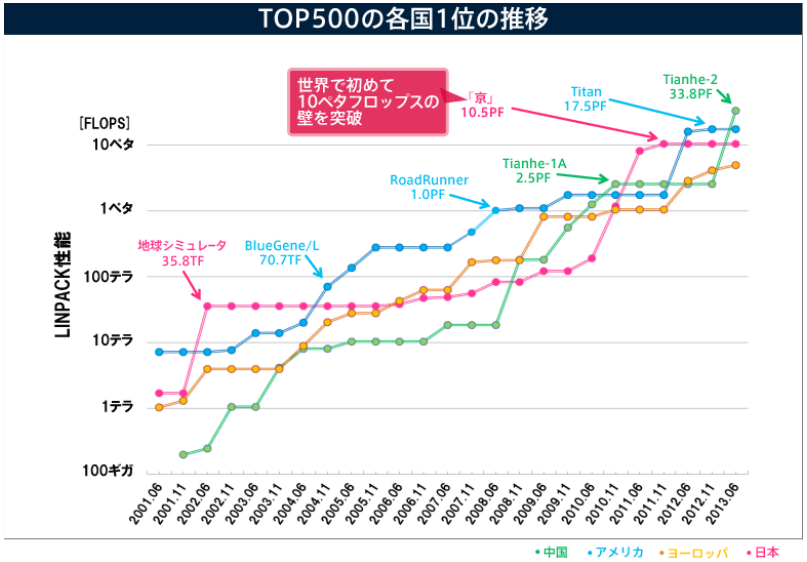
以降、マイクロプロセッサの時代へ

コンピュータの開発

スーパーコンピュータの性能推移



コンピュータの開発

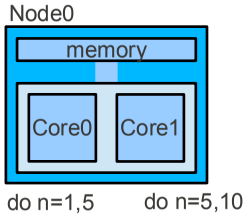


並列計算方法

共有メモリ型並列計算

- 各演算装置 (CPU/Core) は同一のメモリにアクセス
- ループを分割することにより高速化
- プログラミングが容易 (一部のみ並列化可能)
- データ構造の変更が無い
- メモリ容量の削減はされない

```
      nsum = 0
!$OMP parallel do private(n) shared(node,nn) &
!$OMP             reduction(+:nsum)
      do n = 1, node
        nsum = nsum + nn(n)
      enddo
!$OMP end parallel do
```

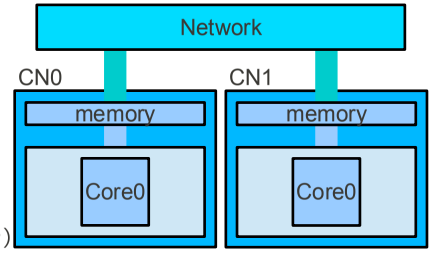


並列計算方法

分散メモリ型並列計算

- 各演算装置 (CPU/Core) は各々のメモリにアクセス
- データを分割することにより高速化
- プログラミングが比較的難しい（全体を並列化）
- データ構造の変更が必要（領域分割）
- メモリ容量が削減される

```
nsum = 0
do n = 1, node
  nsum = nsum + nn(n)
enddo
mpi_allreduce(nsum,nsumr,1,mpi_integer,&
              mpi_sum,mpi_comm_world,ierr)
nsum = nsumr
```



Serial 計算

```
nsum = 0
do n = 1, node
  nsum = nsum + nn(n)
enddo
```

CN0

Memory

node 10 nsum

nn( )	1	2	3	4	5	6	7	8	9	10

Core0

```
nsum = nn(1)+nn(2)+nn(3)+nn(4)+nn(5)
     + nn(6)+nn(7)+nn(8)+nn(9)+nn(10)
```


共有メモリ並列計算

```
    nsum = 0
!$OMP parallel do private(n) shared(node,nn) reduction(+:nsum)
    do n = 1, node
      nsum = nsum + nn(n)
    enddo
!$OMP end parallel do
```

CN0

Memory

node 10 nsum ← reduction(+) nsum + nsum

nn( )	1	2	3	4	5	6	7	8	9	10

nsum nsum *private area

Core0

```
nsum = nn(1)+nn(2)+nn(3)+nn(4)+nn(5)
```

Core1

```
nsum = nn(6)+nn(7)+nn(8)+nn(9)+nn(10)
```


分散メモリ並列計算

```
    nsum = 0
    do n = 1, node
      nsum = nsum + nn(n)
    enddo
    mpi_allreduce(nsum,nsumr,1,mpi_integer,mpi_sum,mpi_comm_world,ierr)
    nsum = nsumr
```

CN0

Memory

node 5 nsum

nn( )	1	2	3	4	5

Core0

```
nsum = nn(1)+nn(2)+nn(3)+nn(4)+nn(5)
```

CN1

Memory

node 5 nsum

nn( )	6	7	8	9	10

Core1

```
nsum = nn(1)+nn(2)+nn(3)+nn(4)+nn(5)
```

nsumr = nsum + nsum
(mpi_allreduce)

What is OpenMP ?

OpenMP is a set of compile directives and callable runtime library routines that extend that Fortran (and other languages) to express shared memory parallelization.

1) http://www.openmp.org/

Hello world (single)

```
program  hello_world
 integer :: mythread
 mythread = 0
 write(6,*) mythread,   'Hello World'
end program hello_world
```

Hello world (OMP)

```
program  hello_world_omp
 integer :: mythread
 mythread = 0
 !$OMP parallel private( mythread )
 !$ mythread = omp_get_thread_num()
  wirte(6,*) mythread, 'Hello World'
 !$OMP end parallel
end program hello_world_omp
```

筑波大学 University of Tsukuba

Why should we use OpenMP ?

Merit

- Easy to parallelize (for simple loop!)
  - MPI programming require the change over the entire your code.
  - MPI programming require the data partitioning. (for the specified parallelization)

Hello world (single)

```
program  hello_world
 integer :: mythread
 mythread = 0
 write(6,*) mythread,   'Hello World'
end program hello_world
```

Hello world (OMP)

```
program  hello_world_omp
 integer :: mythread
 mythread = 0
 !$OMP parallel private( mythread )
 !$ mythread = omp_get_thread_num()
  wirte(6,*) mythread, 'Hello World'
 !$OMP end parallel
end program hello_world_omp
```

筑波大学 University of Tsukuba

Why should we use OpenMP ?

Merit

- Easy to parallelize (for simple loop!)
  - MPI programming require the change over the entire your code.
  - MPI programming require the data partitioning. (for the specified parallelization)
- Easy to construct the OpenMP environment.
  - Can you do the clustering for some computers and setting up the MPI library?

Shared memory parallel computing needs:

- Multi Core/CPU Computer

⇒ If you buy the computer now, it have the dual or quad core CPU(s).

2006.11. Intel started to sell the "first" Quad core CPU (codename:Clovertown)
2007. 9. AMD started to sell the "native" Quad core CPU (codename:Barcelona)

•OpenMP library

⇒ The major compiler (intel, PGI and NAG?) has had the OpenMP option.

筑波大学 University of Tsukuba

Parallerization using OpenMP

The rules of OpenMP

!$OMP	○○○ ×××	: OpenMP directive
!$	○ = × * △	: It is done only OpenMP
!	○×△	: Comments

```
program  hello_world_omp
 integer :: mythread
 mythread = 0
 !$OMP parallel private( mythread )
 !$ mythread = omp_get_thread_num()
  wirte(6,*) mythread, 'Hello World'
 !$OMP end parallel
end program hello_world_omp
```

→ Parallel Construct (start)

→ OMP Function (to get the my thread number)

→ Parallel Construct (end)

筑波大学 University of Tsukuba

OpenMP sample program

```
program  welcome_to_parallel_world_omp
 integer :: mythread
 mythread = 0
 write(6,*) mythread,  'Hello Parallel World'
 !$OMP parallel private( mythread )
 !$ mythread = omp_get_thread_num()
  write(6,*) mythread, 'Parallel Computing'
 !$OMP end parallel
  write(6,*) mythread, 'Good-by'
end program welcome_to_parallel_world_omp
```

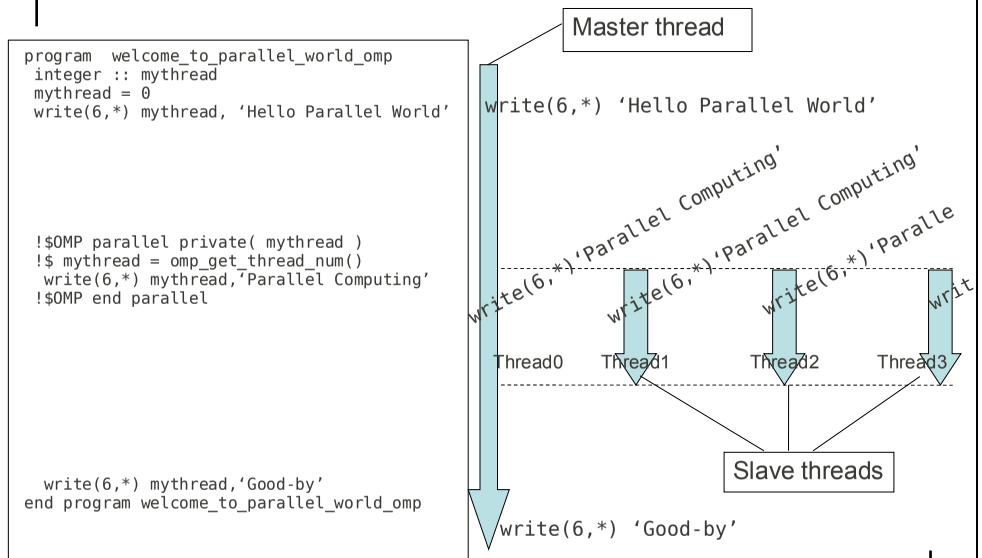
Run with 4 threads

```
0  Hello Parallel World
0  Parallel Computing
1  Parallel Computing
2  Parallel Computing
3  Parallel Computing
0  Good-by
```

筑波大学 University of Tsukuba

Outline of Running on OpenMP

```
program  welcome_to_parallel_world_omp
 integer :: mythread
 mythread = 0
 write(6,*) mythread, 'Hello Parallel World'

 !$OMP parallel private( mythread )
 !$ mythread = omp_get_thread_num()
  write(6,*) mythread,'Parallel Computing'
 !$OMP end parallel

  write(6,*) mythread,'Good-by'
end program welcome_to_parallel_world_omp
```



筑波大学 University of Tsukuba

Parallel and Worksharing Construct

2. Worksharing construct

```
!$OMP parallel
 !$OMP do
   do i = 1, n
     ........
   enddo
 !$OMP enddo
!$OMP end parallel
```

Direction to do worksharing in Parallel Construct

- do directions
- single directions
- workshare directions

3.Unitted Worksharing construct

```
!$OMP parallel do
   do i = 1, n
     ........
   enddo
!$OMP end parallel do
```

Same !

筑波大学 University of Tsukuba

Parallel and Worksharing Construct

2. Worksharing construct

```
!$OMP parallel
 !$OMP do
   do i = 1, n
     ........
   enddo
 !$OMP enddo
   m = n * 2
 !$OMP do
   do j = 1, m
     ........
   enddo
 !$OMP enddo
!$OMP end parallel
```

3.Unitted Worksharing construct

```
!$OMP parallel do
   do i = 1, n
     ........
   enddo
!$OMP end parallel do
   m = n * 2
!$OMP parallel do
   do j = 1, m
     ........
   enddo
!$OMP end parallel do
```

筑波大学 University of Tsukuba

OMP Running of Do Loop

```
program  plus
 integer :: n, i
 integer :: a(10), b(10)

 do n = 1, 10
    a(n) = n
 enddo



!$OMP parallel do private(n,i) shared(a, b)

 do n = 1, 10
    i = n * 2
    b(n) = a(n) + i
 enddo

!$OMP end parallel do


end program plus
```

```
do n = 1, 10
  . . .
enddo
```

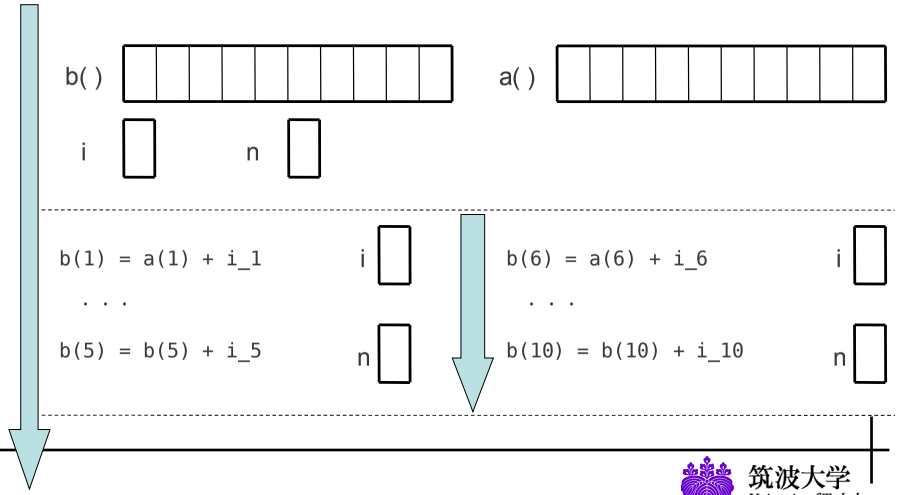
```
do n = 1, 5
  . . .
enddo
```

```
do n = 6, 10
  . . .
enddo
```


Attribute of variables

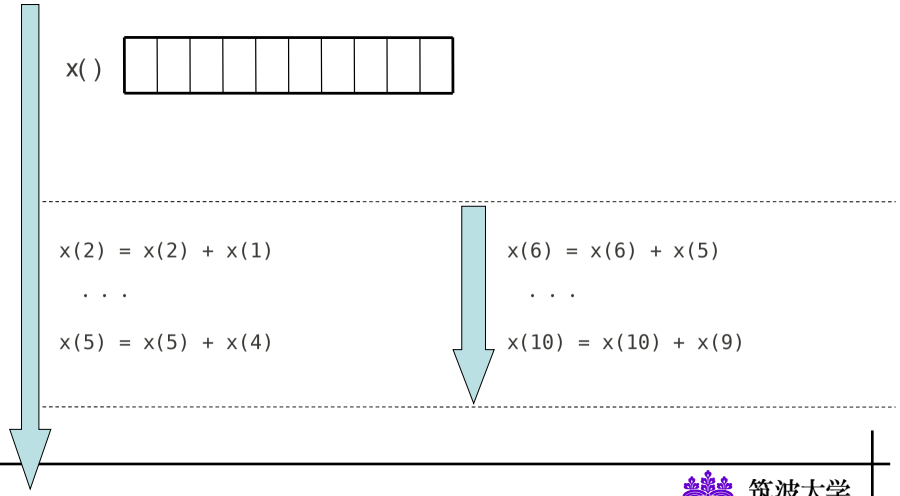
1. Shared and Private variables

Shared ・・・ Variables it can be accessed form each threads
Private ・・・ Independent variables in each threads



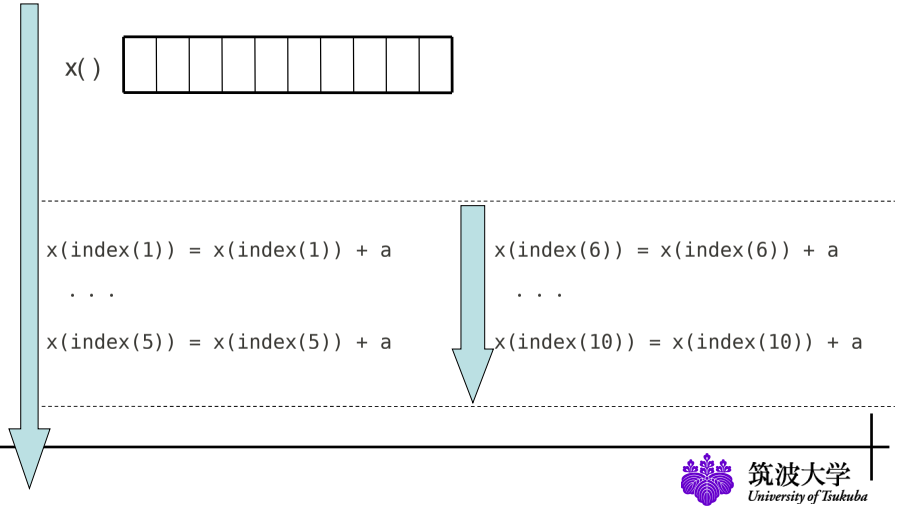
You can not parallelize easily 1

```
do n = 2, 10
 x(n) = x(n) + x(n-1)
enddo
```



You can not parallerize (easily) 2

```
do n = 1, 10
 x(index(n)) = x(index(n)) + a
enddo
```



You can not parallerize (easily) 3

Example: Calculation of the sum

```
do i = 1, num
 a(i) = i
enddo

s = 0
do i=1, num
  s = s + a(i)
enddo
```

Variables “s” is ,
- Private ?
- Shared ?

You can not parallerize (easily) 3

Example: Calculation of the sum

```
do i = 1, num
 a(i) = i
enddo

s = 0
do i=1, num
  s = s + a(i)
enddo
```

Case 1. If “s” is private

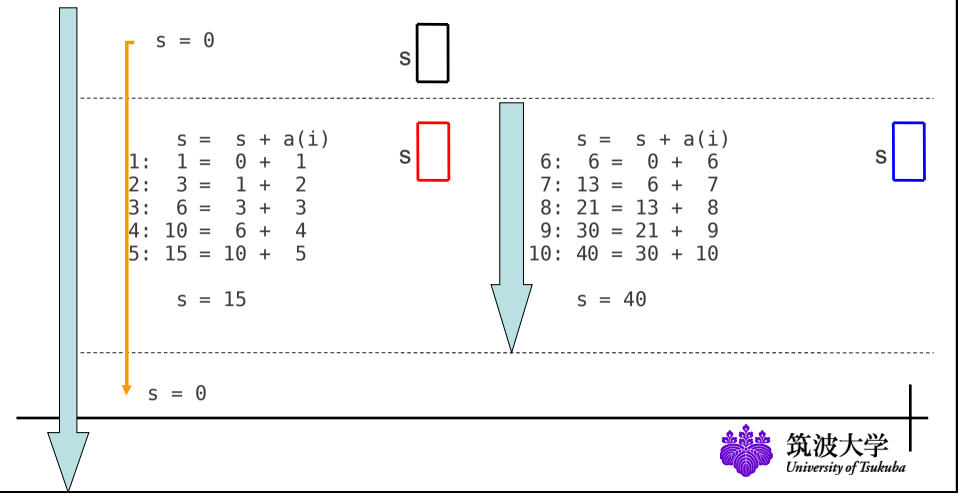
```
do i = 1, num
 a(i) = i
enddo

 s = 0
!$OMP parallel do private(s,i) shared(a,num)
 do i=1, num
   s = s + a(i)
 enddo
!$OMP end parallel do
```

Results

s = 0

You can not parallerize (easily) 3

```
 s = 0
!$OMP parallel do private(s,i) shared(a,num)
 do i=1, num
   s = s + a(i)
 enddo
```



You can not parallerize (easily) 3

Example: Calculation of the sum

```
do i = 1, num
 a(i) = i
enddo

s = 0
do i=1, num
  s = s + a(i)
enddo
```

Case2. If “s” is shared.

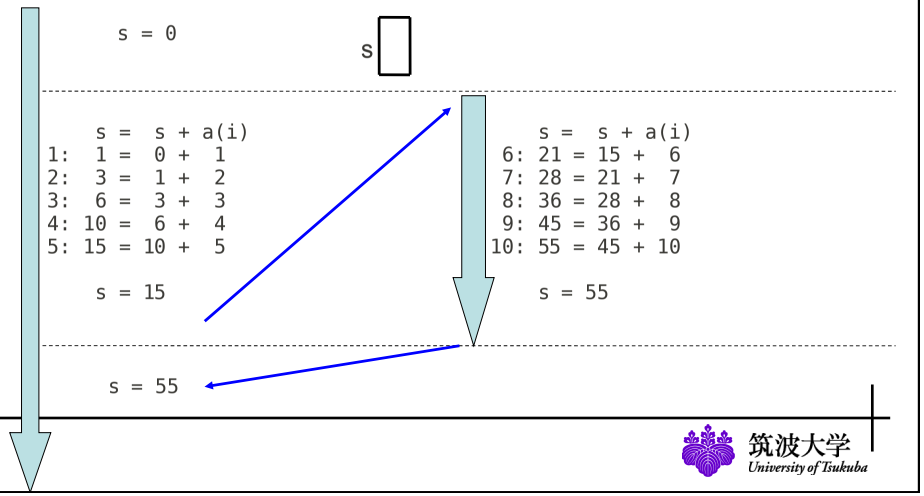
```
do i = 1, num
 a(i) = i
enddo

 s = 0
!$OMP parallel do private(i) shared(a,num,s)
 do i=1, num
   s = s + a(i)
 enddo
!$OMP end parallel do
```

Results

s = 55 (num=10)

You can not parallerize (easily) 3

```
 s = 0
!$OMP parallel do private(i) shared(a,num,s)
 do i=1, num
   s = s + a(i)
 enddo
```



You can not parallerize (easily) 3

Example: Calculation of the sum

```
do i = 1, num
 a(i) = i
enddo

s = 0
do i=1, num
  s = s + a(i)
enddo
```

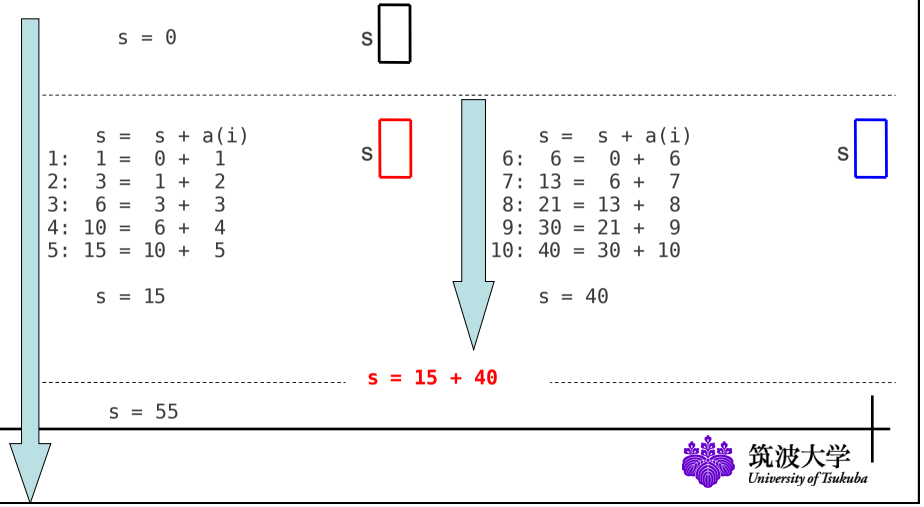
```
do i = 1, num
 a(i) = i
enddo

s = 0
!$OMP parallel do private(i) shared(a,num) &
!$                reduction(+:s)
 do i=1, num
   s = s + a(i)
 enddo
!$OMP end parallel do
```

Result

s = 55

You can not parallerize (easily) 3

```
 s = 0
!$OMP parallel do private(i) shared(a,num) reduction (+:s)
 do i=1, num
   s = s + a(i)
 enddo
```



Basic Styles

```
 s = 0
!$OMP parallel do private(i) shared(a,num) reductio (+:s)
 do i=1, num
   s = s + a(i)
 enddo
!$OMP end parallel do
```

!$OMP parallel do ・・・ United worksharing direction
!$ private(variable1， variable2，) ・・・ Private variables direction
!$ shared(variable1， variable2，) ・・・ Shared variables direction
!$ reduction (Operation ： variable1， variable2，) ・・・ Variable direction

Compile & Run the OpenMP program

If you use Intel Fortran Compiler, you can complile with “-openmp” option.

```
$ ifort –openmp *****.f90
```

* You can see the OMP option in compiler manual

You should define the number of threads, as follows;

* bash

```
$ export OMP_NUM_THREADS=4
```

* csh

```
$ setenv OMP_NUM_THREADS 4
```

* Windows

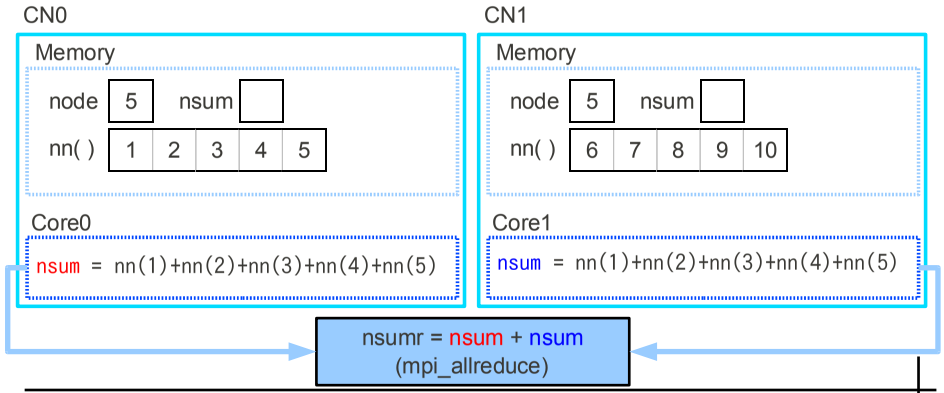
```
$ set OMP_NUM_THREADS=4
```

Finally, you can run the exe file

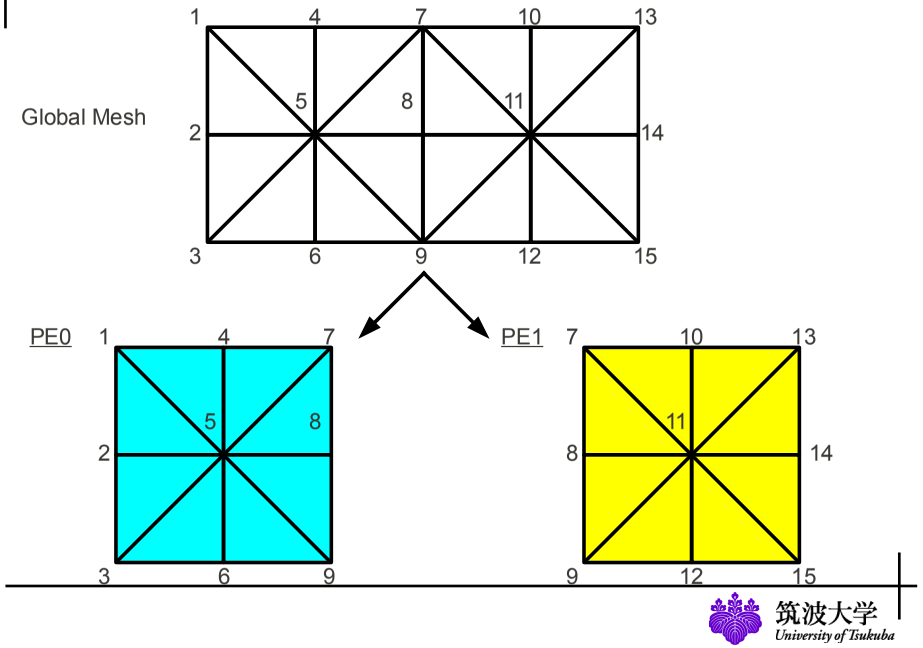
```
$  ./a.out
```


分散メモリ並列計算

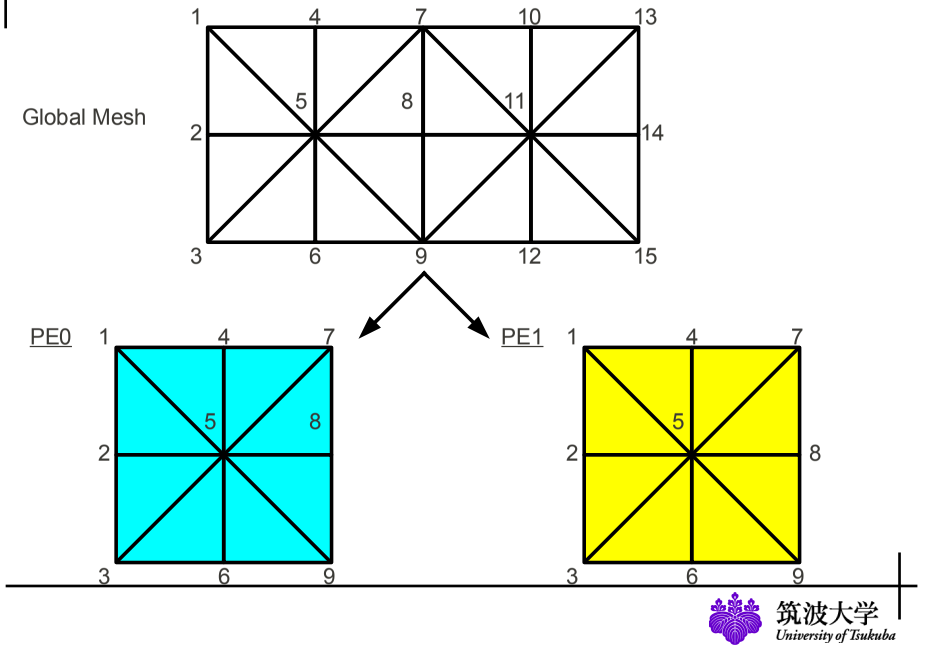
```
nsum = 0
do n = 1, node
  nsum = nsum + nn(n)
enddo
mpi_allreduce(nsum,nsumr,1,mpi_integer,mpi_sum,mpi_comm_world,ierr)
nsum = nsumr
```



領域分割による並列計算

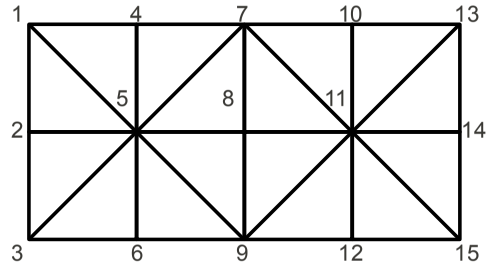


領域分割による並列計算



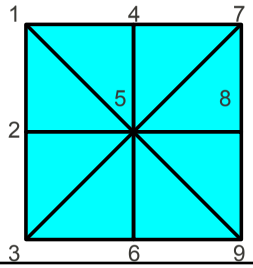
隣接小領域間通信

Global Mesh

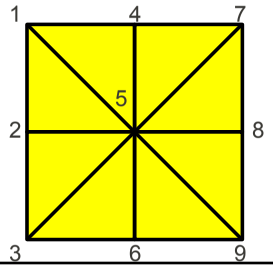



```
rr(:) = 0.0d0
do m = 1, nelem
  do i = 1, 3
    do j = 1, 3
      rr(nc(i,m)) = rr(nc(i,m)) &
        + ea(i,j,m) * xx(nc(i,m))
    enddo
  enddo
enddo
```

PE0



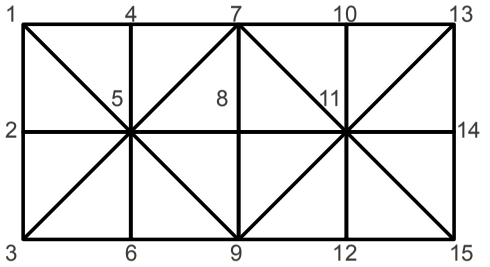
PE1



筑波大学 University of Tsukuba

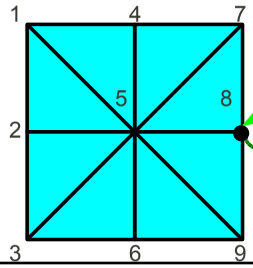
隣接小領域間通信

Global Mesh

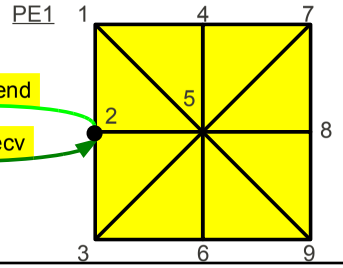



```
rr(:) = 0.0d0
do m = 1, nelem
  do i = 1, 3
    do j = 1, 3
      rr(nc(i,m)) = rr(nc(i,m)) &
        + ea(i,j,m) * xx(nc(i,m))
    enddo
  enddo
enddo
```

PE0



PE1



筑波大学 University of Tsukuba

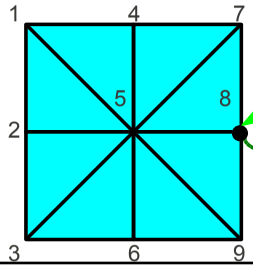
通信に必要なデータ

```
isb                        : 1
nid(1:isb)                 : 1
nsb(1:isb)                 : 3
nsub(1:maxval(nsb),1:isb): 7    8    9
```

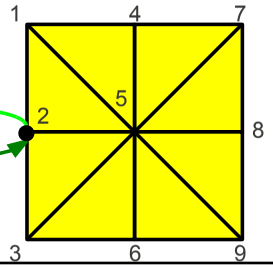
隣の領域がいくつあるか
そのランク ID
領域境界の節点数
その節点番号

```
isb                        : 1
nid(1:isb)                 : 0
nsb(1:isb)                 : 3
nsub(1:maxval(nsb),1:isb): 1    2    3
```

PE0



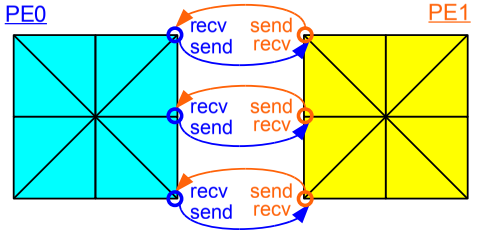
PE1



筑波大学 University of Tsukuba

数値解析で必要な通信

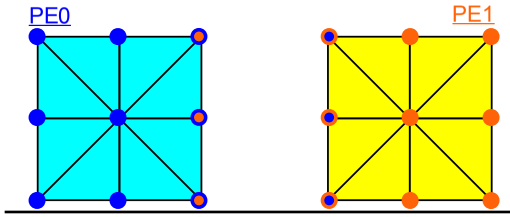
隣接小領域間通信 (Send&Recv)



要素毎の節点ベクトルを全体系ベクトルに重ね合わせる.

- 右辺ベクトルの作成
- 行列 - ベクトル積

全小領域間通信 (Allreduce)



領域全体で評価すべき値の計算

- 解析領域の大きさ (面積・体積)
- 節点ベクトルのノルム計算

筑波大学 University of Tsukuba

MPI の基礎：初期化

```
include "mpif.h"

call mpi_init(ierr)
call mpi_comm_size(MPI_COMM_WORLD,numnod,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)
```

! mpi の初期化
! MPI_COMM_WORLD の MPI プロセス総数を取得
! MPI_COMM_WORLD 内での自身のランクを取得

MPI_COMM_WORLD? ・・・デフォルトのコミュニケーター

MPI の基礎： Send & Recv

Non Blocking Communication : MPI_ISEND, MPI_IRECV and MPI_WAIT

```
!Buffer Setting
   do i = 1, isb
     do j = 1, nsb(i)
       buf_s(j,i) = dat(nsub(j,i))
     enddo
   enddo

!Send <=> Recv
   do i = 1, isb
     icount = nsb(i)
     ncomID = naid(i)
     call MPI_ISEND( buf_s(1,i), icount, MPI_DOUBLE_PRECISION, &
                     ncomID, 0, MPI_COMM_WORLD, irqs(i), ierr)
     call MPI_IRECV( buf_r(1,i), icount, MPI_DOUBLE_PRECISION, &
                     ncomID, 0, MPI_COMM_WORLD, irqr(i), ierr)
   enddo

!Synchronization
   call MPI_WAITALL( isb, irqs(1), istatus, ierr)
   call MPI_WAITALL( isb, irqr(1), istatus, ierr)
```

```
isb                          : 1
nid(1:isb)                   : 1
nsb(1:isb)                   : 3
nsub(1:maxval(nsb),1:isb): 7    8    9
```

```
isb                          : 1
nid(1:isb)                   : 0
nsb(1:isb)                   : 3
nsub(1:maxval(nsb),1:isb): 1    2    3
```


MPI の基礎： Send & Recv

Non Blocking Communication : MPI_ISEND, MPI_IRECV and MPI_WAIT

```
!Buffer Setting
   do i = 1, isb
     do j = 1, nsb(i)
       buf_s(j,i) = dat(nsub(j,i))
     enddo
   enddo

!Send <=> Recv
   do i = 1, isb
     icount = nsb(i)
     ncomID = naid(i)
     call MPI_ISEND( buf_s(1,i), icount, MPI_DOUBLE_PRECISION, &
                     ncomID, 0, MPI_COMM_WORLD, irqs(i), ierr)
!Synchronization
   call MPI_WAITALL( isb, irqs(1), istatus, ierr)
   call MPI_WAITALL( isb, irqr(1), istatus, ierr)
```

送信データの先頭アドレス
送信データの個数
送信データの型
エラーコード
コミュニケーター名
送信リクエスト
tag(とりあえず Send と Recv で同じ値を)
宛先のコミュニケーター内のランク

MPI の基礎： Send & Recv

Non Blocking (MPI_ISEND/IRECV) と Blocking(MPI_SEND/RECV) の違い

```
!Send <=> Recv
   do i = 1, isb
     icount = nsb(i)
     ncomID = naid(i)
     call MPI_ISEND( buf_s(1,i), icount, MPI_DOUBLE_PRECISION, &
                     ncomID, 0, MPI_COMM_WORLD, irqs(i), ierr)
     call MPI_IRECV( buf_r(1,i), icount, MPI_DOUBLE_PRECISION, &
                     ncomID, 0, MPI_COMM_WORLD, irqr(i), ierr)
   enddo
!Synchronization
   call MPI_WAITALL( isb, irqs(1), istatus, ierr)
   call MPI_WAITALL( isb, irqr(1), istatus, ierr)
!
!    do i = 1, isb
!      icount = nsb(i)
!      ncomID = naid(i)
!      call MPI_SEND( buf_s(1,i), icount, MPI_DOUBLE_PRECISION, &
!                     ncomID, 0, MPI_COMM_WORLD, ista_s, ierr)
!      call MPI_RECV( buf_r(1,i), icount, MPI_DOUBLE_PRECISION, &
!                     ncomID, 0, MPI_COMM_WORLD, ista_r, ierr)
!    enddo
```

MPI の基礎： Send & Recv

Non Blocking (MPI_ISEND/IRECV) と Blocking(MPI_SEND/RECV) の違い

Blocking

MPI_SEND がコールされ，そのデータが相手に MPI_RECV されて送受信が完了．これが正常に終了するまで，次の作業には進めない．

Non-Blocking

MPI_ISEND がコールされる．ただし，送受信が完了していなくても，別の別の処理を開始することが可能．送受信を完了するには MPI_WAIT をコールする必要がある．

```
!
!   do i = 1, isb
!     icount = nsb(i)
!     ncomID = naid(i)
!     call MPI_SEND( buf_s(1,i), icount, MPI_DOUBLE_PRECISION, &
!                    ncomID, 0, MPI_COMM_WORLD, ista_s, ierr)
!     call MPI_RECV( buf_r(1,i), icount, MPI_DOUBLE_PRECISION, &
!                    ncomID, 0, MPI_COMM_WORLD, ista_r, ierr)
!   enddo
```

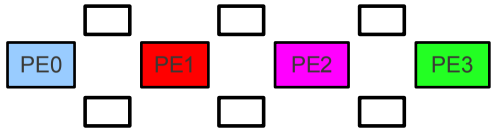
筑波大学 University of Tsukuba

MPI の基礎： Send & Recv

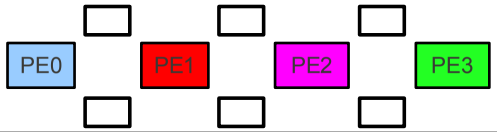
Non Blocking (MPI_ISEND/IRECV) と Blocking(MPI_SEND/RECV) の違い

```
do i = 1, isb
  icount = nsb(i)
  ncomID = naid(i)
  call MPI_SEND( buf_s(1,i), icount, MPI_DOUBLE_PRECISION, &
                 ncomID, 0, MPI_COMM_WORLD, ista_s, ierr)
  call MPI_RECV( buf_r(1,i), icount, MPI_DOUBLE_PRECISION, &
                 ncomID, 0, MPI_COMM_WORLD, ista_r, ierr)
enddo
```

Blocking



Non-Blocking

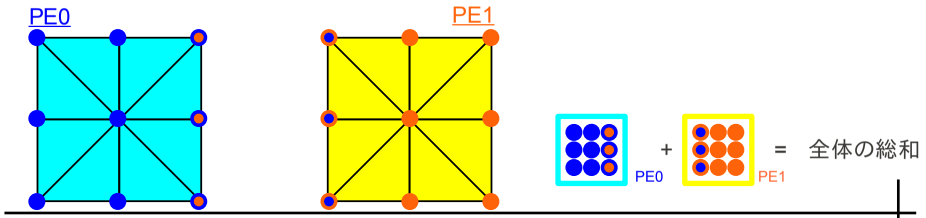


筑波大学 University of Tsukuba

MPI の基礎： Allreduce

```
dots = 0.0d0
do n = 1, node                          ← 節点重複数
  dots = dots + r(n)*r(n)/ndb(n)
enddo

call MPI_ALLREDUCE( dots, dotr, 1, MPI_DOUBLE_PRECISION, &
                    MPI_SUM, MPI_COMM_WORLD, ierr )
```

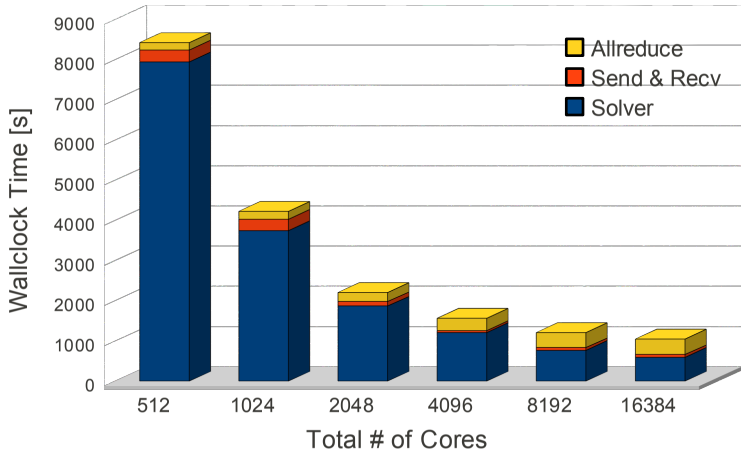
全小領域間通信 (Allreduce)



筑波大学 University of Tsukuba

大規模並列計算での問題点 1

MPI プロセス数 (小領域数) の増加による Allreduce 通信時間の増加

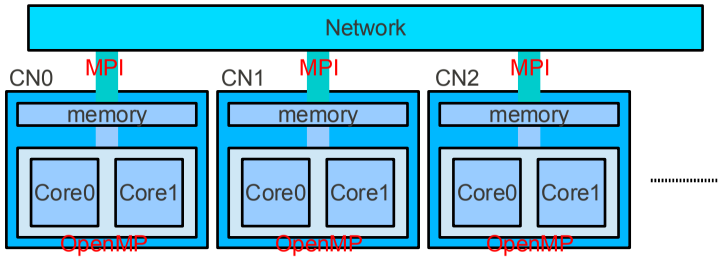


筑波大学 University of Tsukuba

大規模並列計算での問題点 1

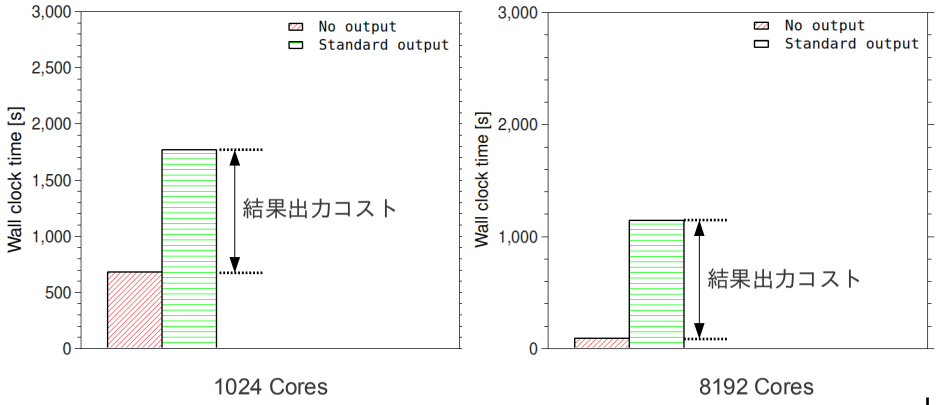
Allreduce 通信を減らすためには．．．

MPI/OpenMP Hybrid 並列計算



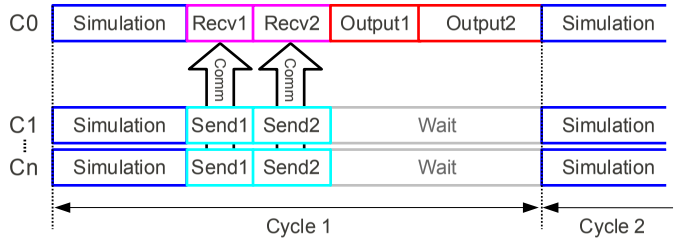
大規模並列計算での問題点 2

大規模 (高解像度) メッシュを利用すると，
結果出力ファイルも巨大なものとなり，そのコストが増大する．



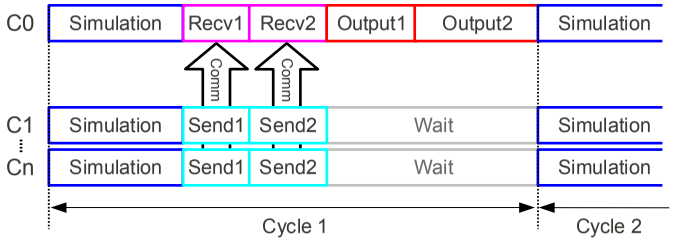
出力アルゴリズム

Standard outputting

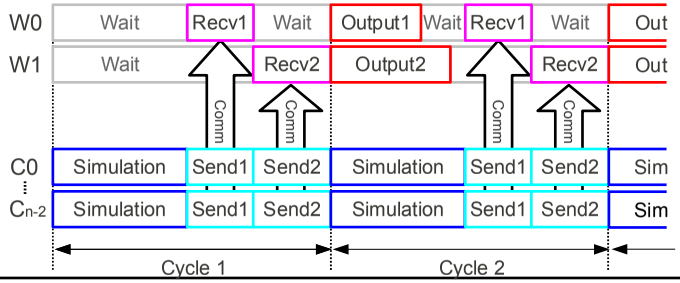


出力アルゴリズム

Standard outputting

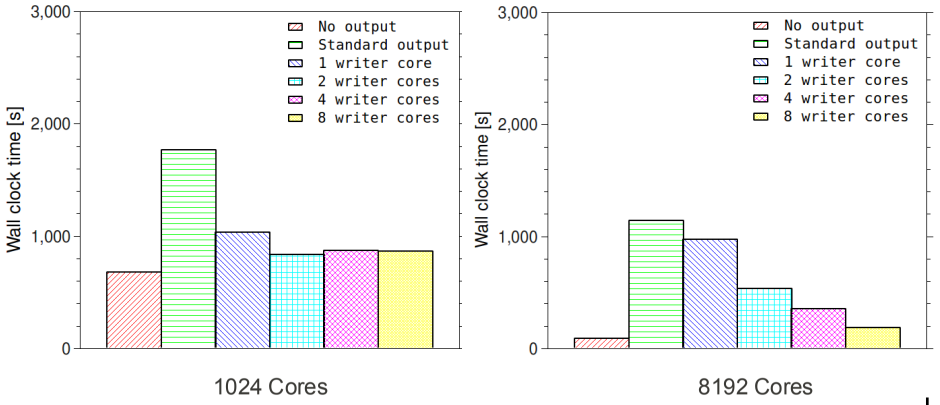


Writer core outputting



Writer Cores の性能

0.125 day simulation of Katrina Hurricane on SL15x04 grid
Output every 15 minutes,
4 files (Elevation, Velocity, Atmospheric pressure, Wind Stress)
Explicit method.



MPI の基礎：コミュニケータの分割

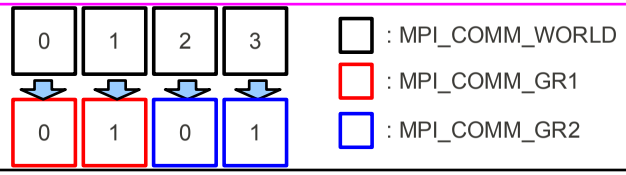
```
call mpi_init(ierr)
call mpi_comm_dup(MPI_COMM_WORLD, MPI_COMM_GBL)
call mpi_comm_size(MPI_COMM_GBL,numnod_gbl,ierr)
call mpi_comm_rank(MPI_COMM_GBL,myrank_gbl,ierr)
call mpi_comm_group(MPI_COM_GBL,GROUP_GBL,ierr)
n1 = numnod_gbl/2
n2 = numnod_gbl-n1
do i = 1, n1
  ilist(i) = i-1
enddo
call mpi_group_incl(GROUP_GBL,n1,ilist(1),GROUP1,ierr)
call mpi_comm_create(MPI_COMM_GBL,GROUP1,MPI_COMM_GR1,ierr)
do i = 1, n2
  ilist(i) = n1+i-1
enddo
call mpi_group_incl(GROUP_GBL,n2,ilist(1),GROUP2,ierr)
call mpi_comm_create(MPI_COMM_GBL,GROUP2,MPI_COMM_GR2,ierr)
```

コミュニケータを複製

全体のグループを作成

GROUP_GBL から GROUP1 を作成

GROUP1 からなる コミュニケータを作成



結果出力方法

POSIX I/O

Ascii output

```
do i = 1, nnode
  write(60,*)  uu(i)
enddo
```

Binary output (formatted)

```
inquire(iolength=length) a
open(60,access='direct', recl=length)
irec=0
do i = 1, nnode
  irec=irec+1
  write(60,rec=irec)  uu(i)
enddo
```

Binary output (unformatted)

```
open(60,format='unformatted')
write(60)  (uu(i), i=1,nnode)
```

Slow

Fast

ただし， endian 問題があり，異なる環境 (CPU， OS 等) により
互換性が失われる場合がある．

結果出力方法

NetCDF (Network Common Data Form)

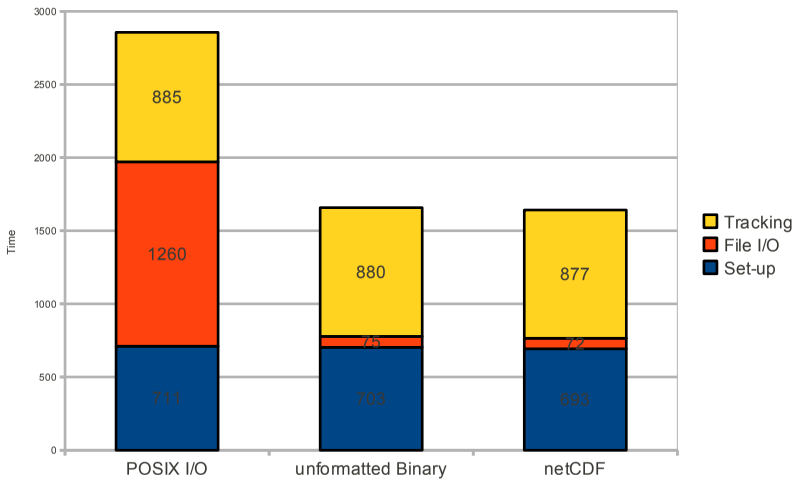
- University Corporation Atmospheric Research (UCAR) が開発.
- Library と幾つかの tool のパッケージ
- C/C++/Fortran， その他の言語 (Perl, Python, Ruby, Matlab 等) でも利用可能

```
use netCDF
ist=nf90_create('fort.60',nf90_64bit_offset,ncid)
ist=nf90_def_dim(ncid,'Total_Num_of_Nodes',nnode,id_nnode)
ist=nf90_def_var(ncid,'Velocity_u',nf90_double,id_nnode,id_varU)
ist=nf90_enddef(ncid)

ist=nf90_put_var(ncid,id_varU,uu,start=(/1/),count=(/nnode/))
ist=nf90_close(ncid)
```

```
$ gfortran -c -I$NETCDF_INC wirte_nc.f90
$ gfortran -o write_nc.exe write_nc.o -L$NETCDF_LIB -lnetcdf
```

出力コストの比較

1000 万 particles, 20snaps, 16threads with OpenMP



筑波大学 University of Tsukuba

大規模並列計算での軽微な問題点

Problems on 32,768 (=2^{15}) MPI process

Range of 32bit Integer number (-2^{31}~2^{31})

- Metis 4.0 requires 4*(# of MPI)^2 for memory allocation.

Limit of maximum total number of sub-directories. *1

- ext3 file system has the limit. (max: 31,998)

GlobalDir/PE00000/mesh.data, bc.data, comm.data

⋮

/PE31997/mesh.data, bc.data, comm.data

Memory allocation related to # of MPI

```
isb                          : 1
nid(1:isb)                   : 1
nsb(1:isb)                   : 3
nsub(1:maxval(nsb),1:isb): 7    8    9
```

Sometimes nsub is allocated as,
allocate(nsub(1:nnode/nproc,1:nproc))

*1 The newest ext4 file system don't have such a limitation.

筑波大学 University of Tsukuba