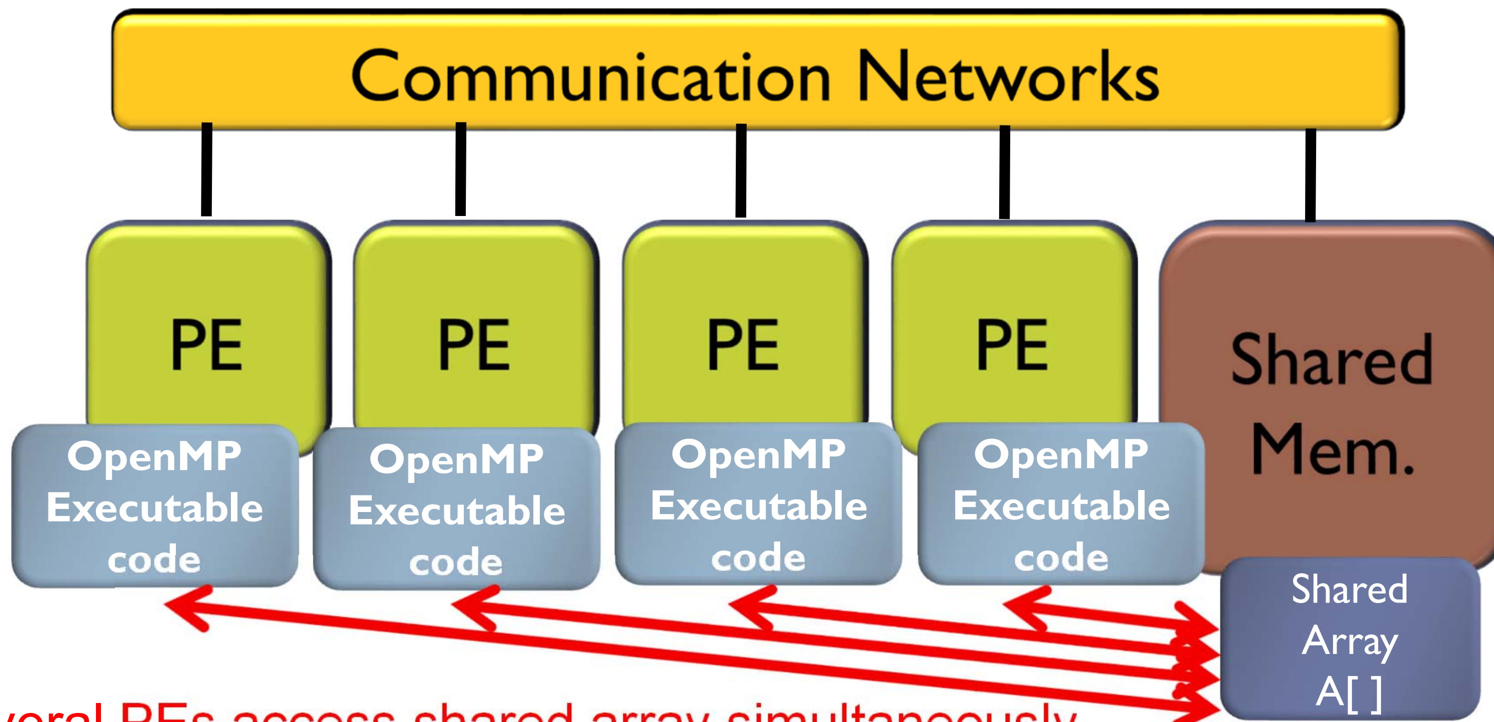
Overview of OpenMP

東京大学情報基盤中心　准教授　片桐孝洋
Takahiro Katagiri, Associate Professor,
Information Technology Center, The University of Tokyo

台大数学科学中心 科学計算冬季学校

Target Machine of OpenMP

- OpenMP is designed for shared memory parallel machine.



Several PEs access shared array simultaneously.
⇒ Parallel result may not equal to sequential result,
if we do not take care of controls in parallel processing.

What is OpenMP?

- OpenMP (OpenMP C and C++ Application Program Interface Version 1.0) is a standard specification for programs on shared memory parallel machines in the follows:
  1. Directives,
  2. Libraries, and
  3. Environmental Variables.
- Programmer specifies directives to parallelize own codes. It is not compiler with automatic parallelization.
- Programming with OpenMP is easier than that with MPI, since there is no cost of data distribution. But there is a limitation of scaling up. (See parallelism inside node.)

OpenMP and Multicore Parallel Machines (1/2)

- OpenMP is a programming model for thread parallelization.
- OpenMP is well suited for current multicore parallel machines.
  - Experimental Performance： good for parallel execution less than 8 threads.
  - Highly programming effort is needed to obtain high parallel efficiency if we use more than 8 threads.
    1. Low performance of data transfer from main memory to cache.
    2. There is no parallelism in target program.
- OpenMP cannot parallelize programs with internode communications.
  - MPI is used to implement internode parallelization.
  - Only thread parallelization is supported for automatic parallelization compilers.
    - It supports internode parallelization for HPF. In research level, XcalableMP(Tsukuba U. and RIKEN AICS) can parallelize sequential program to parallel program with internode communication. However, XcalableMP is not widely supported for CPUs.

OpenMP and Multicore Parallel Machines (2/2)

- Typical Number of Threads
  - 16 Threads / Node
    - The Fujitsu PRIMEHPC FX10 （Sparc64 IVfx）
  - 32 - 128 Threads / Node
    - Fujitsu FX100 (Sparc64 VIfx)
    - HITACHI SR16000 (IBM Power7)
      - 32 Physical cores, 64 - 128 Theoretical Cores (with SMT)
  - 60 - 240 Threads / Node
    - Intel Xeon Phi (Intel MIC(Many Integrated Core) , Knights Conner)
      - 60 Physical Cores, 120 – 240 Theoretical Cores (with HT)
- OpenMP execution with 100 threads or more is pervasive.
  - To establish high performance, much effort of programming is required.

Basics of OpenMP Directives

- In C Language:
  - Comments with
    #pragma omp
- In Fortran Language:
  - Comments with
    !$omp

How to compile program with OpenMP

- Add option for OpenMP to compiler for sequential.
  - e.g.) Fujitsu Fotran90 Compiler
    frt –Kfast,openmp foo.f
  - e.g.） Fujitsu C Compiler
    fcc –Kfast,openmp foo.c
- Loops without OpenMP directives are sequential.
- Some compilers support automatic parallelization of threads in addition to parallelization with OpenMP. However, this depends on vendors.
  - Lines with OpenMP directives are parallelized with OpenMP threads, and lines without OpenMP directives are parallelized with automatic parallelization of threads by compiler.
  - e.g.) Fujitsu Fortran90 Compiler

    frt –Kfast,parallel,openmp foo.f

How to execute executable files of OpenMP

- Specify the file name in command line.
- Number of processes can be specified with environmental variable OMP_NUM_THREADS
- e.g.) In case that executable file is "*a.out*".

```
$ export OMP_NUM_THREADS=16
$ ./a.out
```

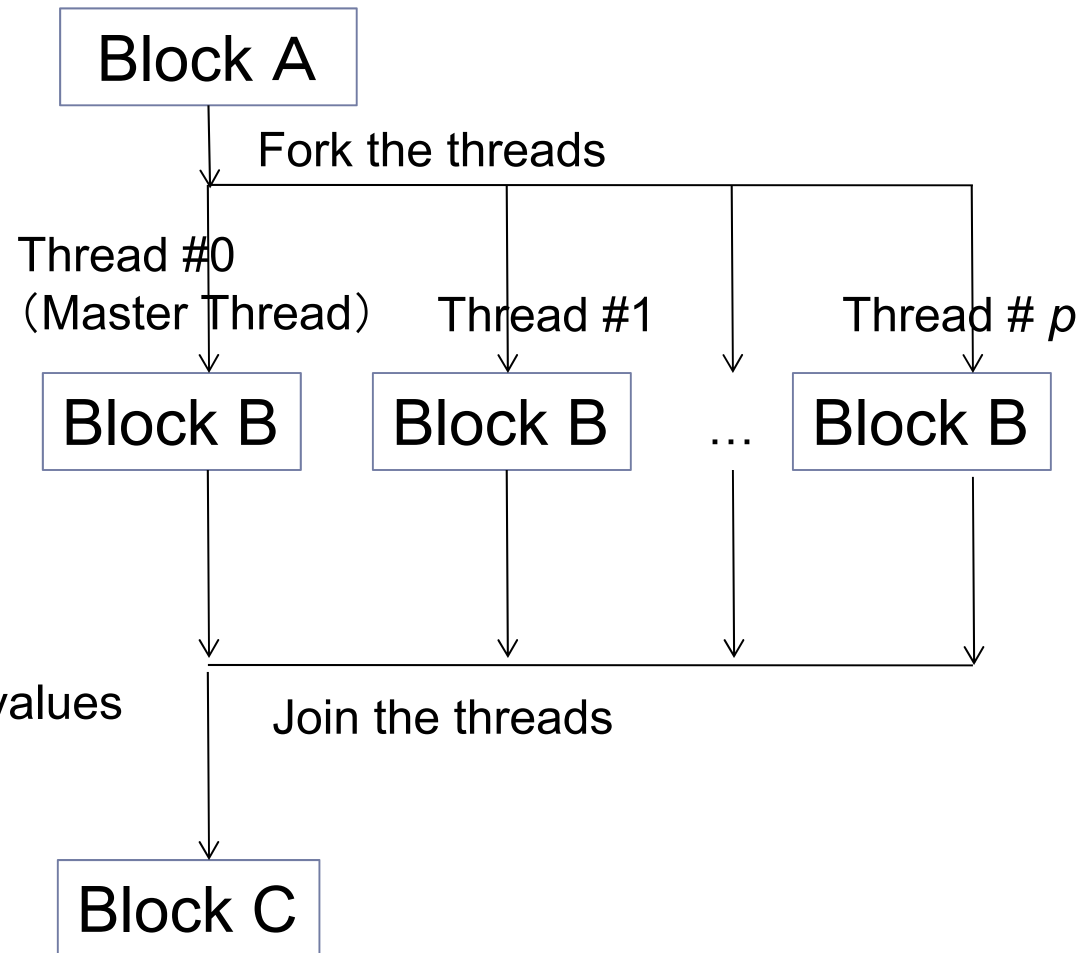
- Note
  - Execution speeds between sequential comping and OpenMP compiling with OMP_NUM_THREADS=1 may different.
    (Execution with OpenMP compiling is slower.)
    - The main reason is additional processes for OpenMP parallelization (overheads).
    - With highly thread execution, the overheads become remarkable.
    - It is possible to improve performance by implementation of codes.

Execution model of OpenMP

Model of OpenMP（C Language)

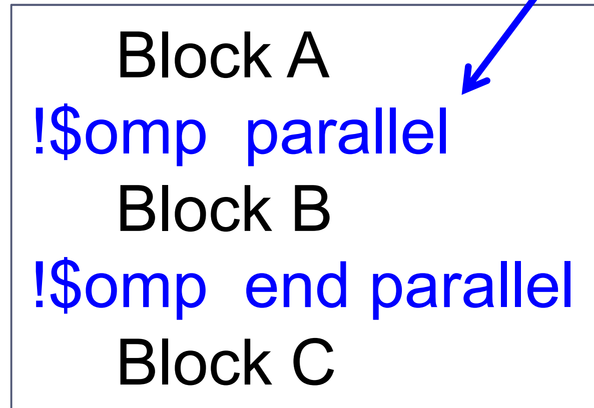
OpenMP Directive

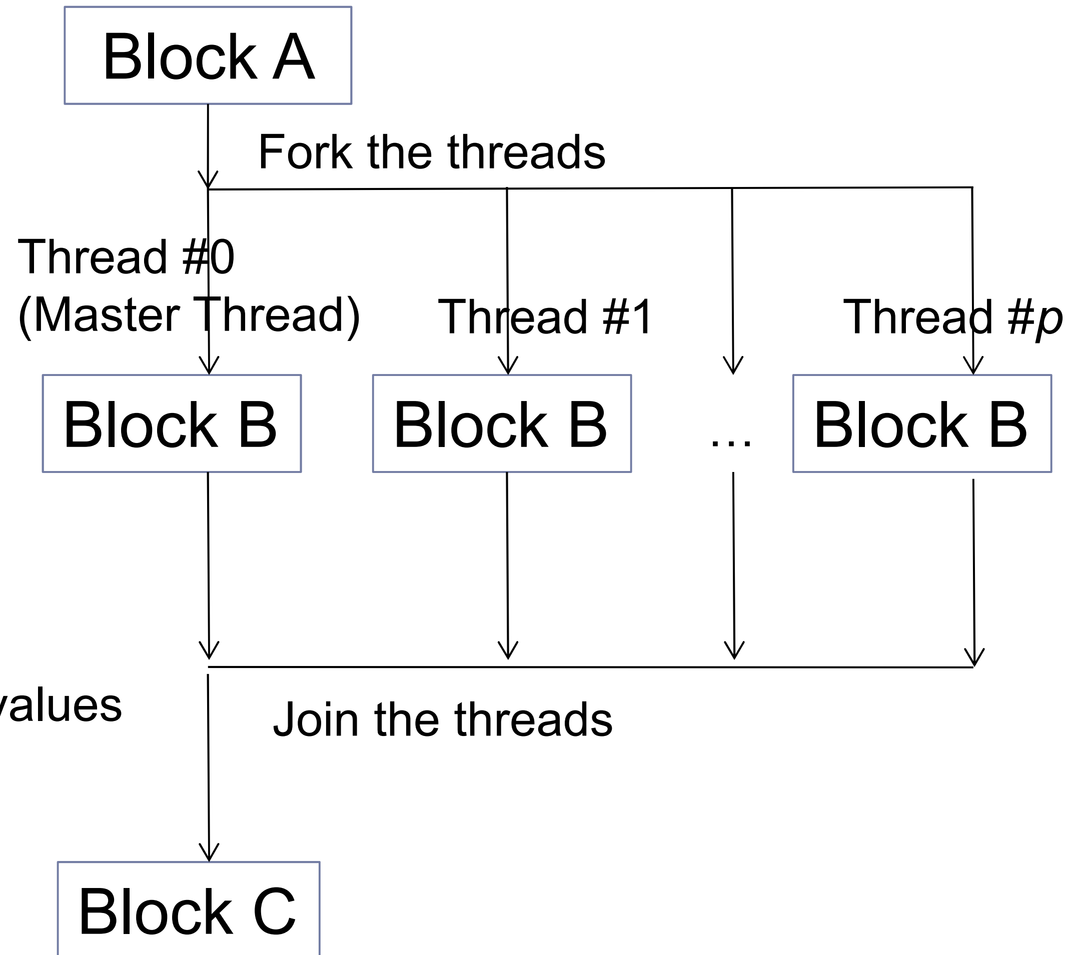
```
    Block A
#pragma  omp  parallel
{
    Block B
}
    Block C
```



* The number of thread p is specified with environmental values OMP_NUM_THREADS

Model of OpenMP (Fortran Language)





* The number of thread *p* is specified with environmental values
OMP_NUM_THREADS

Work Sharing Construct

- The process of parallel description by OpenMP with multiple threads, such as **Block B** for the **parallel** construct, is called *Parallel Region*.
- OpenMP construct that specify parallel region, and execute parallel between threads is Work Sharing Construct.
- The work sharing construct provides the followings:
  1. Described in parallel region:
     - **for** construct. (**do** construct)
     - **sections** construct.
     - **single** construct. (**master** construct), etc.
  2. With **parallel** construct:
     - **parallel for** construct. (**parallel do** construct.)
     - **parallel sections** construct, etc.

Typical Constructs

for construct (**do** construct)

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

* In Fortran Language :

```
!$omp parallel do
 …
!$omp end parallel do
```

Upper Process

Fork the threads

Thread #0

```
for (i=0; i<25; i++){
  a[i] = a[i] * b[i];
}
```

Thread #1

```
for (i=25; i<50; i++){
  a[i] = a[i] * b[i];
}
```

Thread #2

```
for (i=50; i<75; i++){
  a[i] = a[i] * b[i];
}
```

Thread #3

```
for (i=75; i<100; i++){
  a[i] = a[i] * b[i];
}
```

Join the threads

Lower Process

Users must verify
collect results
in parallel compared
to results in sequential.

Cases that cannot specify **for** construct

```
for (i=0; i<100; i++) {
  a[i] = a[i] +1;
  b[i] = a[i-1]+a[i+1];
}
```

- The results differ from sequential
  (See a case that a[i-1] is not updated, and read it in a thread.)

```
for (i=0; i<100; i++) {
  a[i] = a[ ind[i] ];
}
```

- It depends on contents of ind[i] that whether it can parallelize or not.
- If all a[ind[i]] are not updated in parallel, the loop can be parallelized.

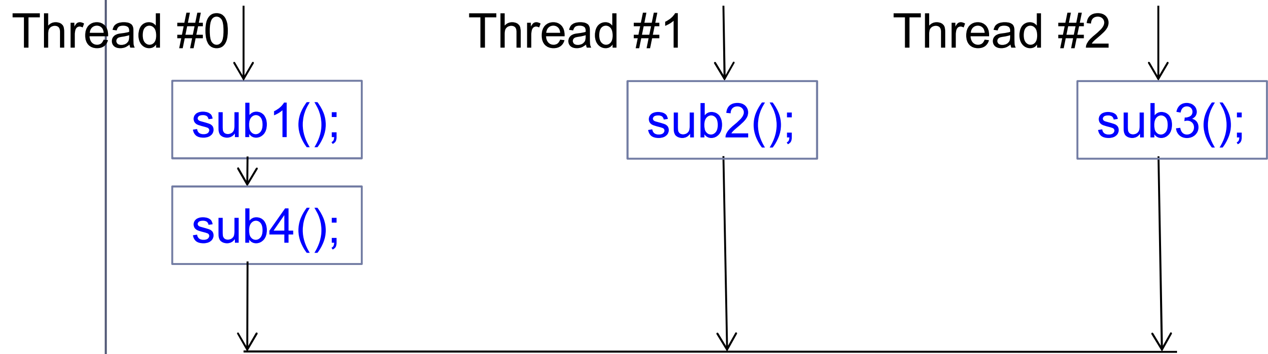
sections construct

```
#pragma omp parallel sections
{
#pragma omp section
    sub1();
#pragma omp section
    sub2();
#pragma omp section
    sub3();
#pragma omp section
    sub4();
}
```

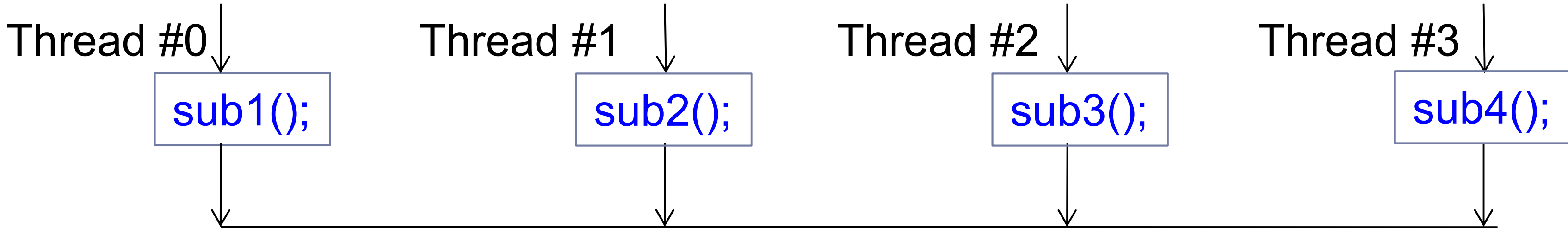
In Fortran Language:

```
!$omp parallel sections
…
!$omp end parallel sections
```

●Case that number of threads is 3.



●Case that number of threads is 4.

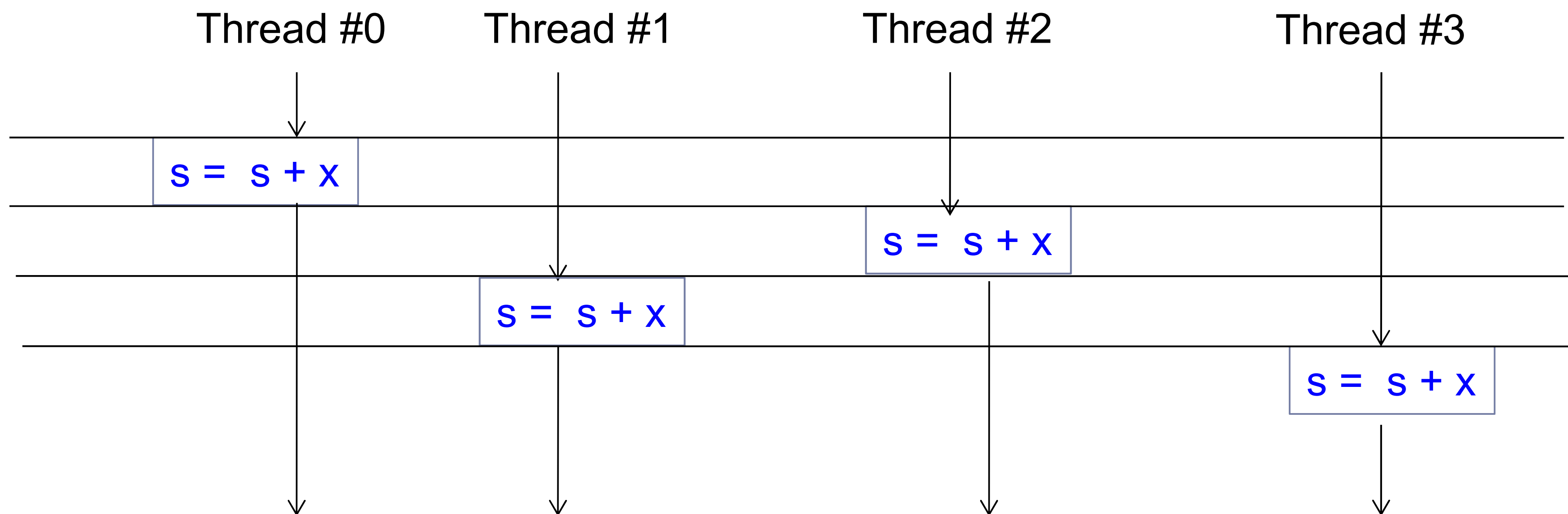


critical construct

In Fortran Language:

```
!$omp critical
...
!$omp end critical
```

```
#pragma omp critical
{
  s =  s +  x;
}
```



private clause

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

The variable c is allocated in each thread. (This indicates different variables in each threads.)

If we use value of c that is defined before the loop, **firstprivate(c)** is needed to specify.

Upper process

Fork the threads

Thread #0

```
for (i=0; i<25; i++){
  a[i] = a[i] + c0*b[i];
}
```

Thread #1

```
for (i=25; i<50; i++){
  a[i] = a[i] + c1*b[i];
}
```

Thread #2

```
for (i=50; i<75; i++){
  a[i] = a[i] + c2*b[i];
}
```

Thread #3

```
for (i=75; i<100; i++){
  a[i] = a[i] + c3* b[i];
}
```

Join the threads

Lower process

A Note of **private** clause (C Language)

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

- Loop induction variable j is allocated as different variable in each thread.
- If private(j) is not specified, j is summed up simultaneously between all threads, then we obtain different result from sequential result.

A Note of **private** clause (Fortran Language)

```
!$omp parallel do private( j )
do i=1, 100
   do j=1,  100
      a( i ) = a( i ) + amat( i , j ) * b( j )
   enddo
enddo
!$omp end parallel do
```

- Loop induction variable j is allocated as different variable in each thread.
- If private(j) is not specified, j is summed up simultaneously between all threads, then we obtain different result from sequential result.

reduction clause (C Language)

- In case to obtain a result to sum results of parallel execution in each thread, such as dot product.
  - Without **reduction** clause, ddot is defined as a shared variable, then parallel summations perform in each thread. This causes wrong answer to result in sequential.

```
#pragma omp parallel for reduction(+, ddot )
for (i=1; i<=100; i++) {
    ddot += a[ i ] * b[ i ]
}
```

ddot can only specify "scalar" variable.
It is not allowed to specify array.

reduction clause (Fortran Language)

- In case to obtain a result to sum results of parallel execution in each thread, such as dot product.
  - Without **reduction** clause, ddot is defined as a shared variable, then parallel summations perform in each thread. This causes wrong answer to result in sequential.

```
!$omp parallel do reduction(+, ddot )
do i=1, 100
    ddot = ddot + a(i) * b(i)
enddo
!$omp end parallel do
```

ddot can only specify "scalar" variable. It is not allowed to specify array.

A note of **reduction** clause

- **reduction** clause performs exclusively hence performance goes to down.
  - In our experience, it causes heavy speed down in case of more than 8 threads.
- The following implementation that allocates array of ddot for summation may be fast. (This depends on size of loop, and hardware architecture)

```
!$omp parallel do private ( i )
do j=0, p-1
  do i=istart( j ), iend( j )
     ddot_t( j ) = ddot_t( j ) + a(i) * b(i)
  enddo
enddo
!$omp end parallel do
ddot = 0.0d0
do j=0, p-1
    ddot = ddot + ddot_t( j )
enddo
```

Making loop length according to threads : Maximum p threads.

Loop lengths are set with respect to each PE in advance.

Local array ddot_t() for computation of ddot is allocated in advance. (initialized by 0)

Sequential summation.

Other OpenMP Functions

Obtaining Maximum Number of Threads

- To obtain maximum number of threads, we use omp_get_num_threads().
- Type is integer (Fortran Language), int (C Language).

● e.g.) Fortran90

```
use omp_lib
Integer nthreads

nthreads = omp_get_num_threads()
```

● e.g.) C Language

```
#include <omp.h>
int nthreads;

nthreads = omp_get_num_threads();
```

Obtaining own identification number of threads

- To obtain own identification number of threads, we use omp_get_thread_num().
- Type is integer (Fortran Language), int (C Language).

● e.g.) Fortran90

```
use omp_lib
Integer myid

myid = omp_get_thread_num()
```

● e.g.) C Language

```
#include <omp.h>
int myid;

myid = omp_get_thread_num();
```

Time Measurement Function

- To obtain elapse time, we use omp_get_wtime().
- Type is double precision (Fortran Language), double (C Language).

● e.g.) Fortran90

```
use omp_lib
double precision dts, dte

dts = omp_get_wtime()
    対象の処理
dte = omp_get_wtime()
print *, “Elapse time [sec.] =”,dte-dts
```

● e.g.) C Language

```
#include <omp.h>
double dts, dte;

dts = omp_get_wtime();
    対象の処理
dte = omp_get_wtime();
printf(“Elapse time [sec.] = %lf ¥n”,
  dte-dts);
```

Other Constructs

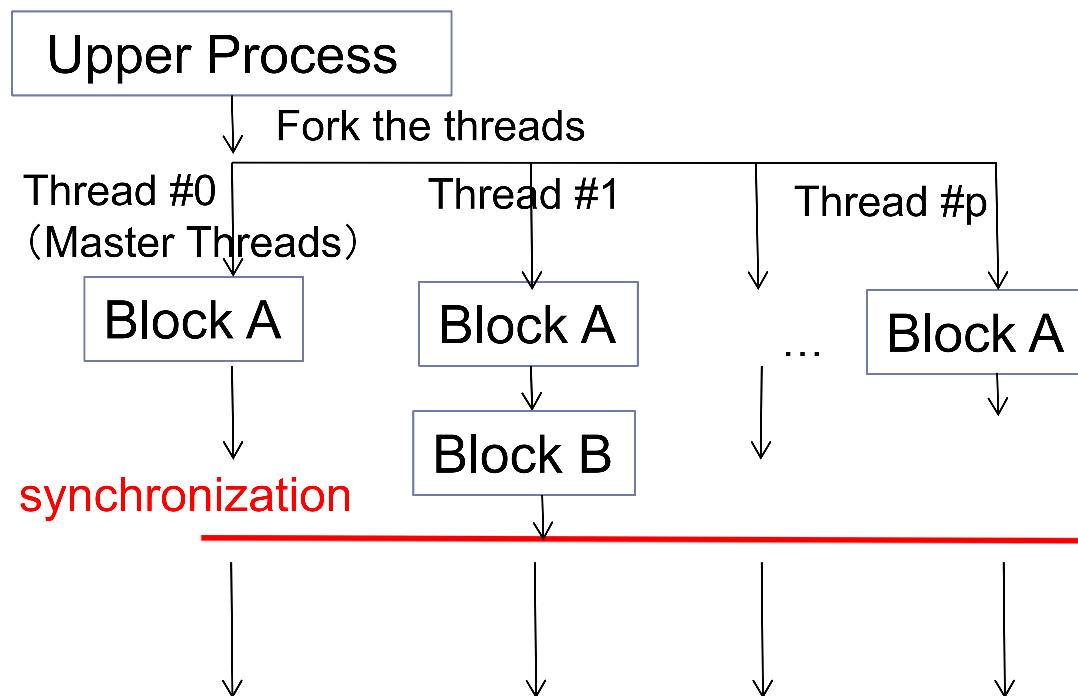
single construct

In Fortran language:

```
!$omp single
…
!$omp end single
```

- A block specified by **single** construct is allocated to a thread.
- It is not predictable which thread should be allocated.
- Except for using **nowait** construct, a synchronization is inside.

```
#pragma omp parallel for
{
    Block A
#pragma omp single
{   Block B }
…
}
```



master construct

- Using **master** construct is as same as **single** construct.
- Difference is: it is allocated for master thread that process specified by **master** construct, for example, **Block B** in the previous figure.
- There is no synchronization after finishing the region.
  - Due to that, the execution speeds up in some cases.

flush construct

- Keep consistency with contents in physical memory
- Variables specified by **flush** construct are consistent in the location. The other variables are not consistent from contents in memory.
  - Computed results are store in registers. The results do not be stored in memory.
  - Hence results are different every execution if we do not specify **flush** construct.
  - The following constructs are automatically include **flush** construct.
    - **barrier** construct, enter and out of **critical**, out of **parallel**.
    - Out of **for, sections**, and **single** constructs are implicitly flushed.
- Using flush construct makes performance down. Try to avoid using it.

```
#pragma omp flush (Lists of variables)
```

If lists of variable are omitted, all variables are specified.

threadprivate construct

- Declare private variables in each thread, but the variables can be accessed in global.
- It is good for declaration of global variables which have different values in each thread.
  - For example, define different values of start and end of loops in each thread.

```
#include <omp.h>
int myid, nthreds, istart, iend;
#pragma omp threadprivate (istart,
iend)
…
void kernel() {
   int i;
   for (i=istart; i<iend; i++) {
      for (j=0; j<n; j++) {
         a[ i ] = a[ i ] + amat[ i ][ j ] * b[ j ];
      }
   }
}
…
```

```
…
void main() {
…
#pragma omp parallel private (myid, nthreds,
istart, iend)   {
  nthreds = omp_num_threds();
  myid = omp_get_thread_num();
  istart = myid * (n/nthreads);
  iend = (myid+1)*(n/nthreads);
  if (myid == (nthreads-1)) {
     nend = n;
  }
  kernel();
}
```

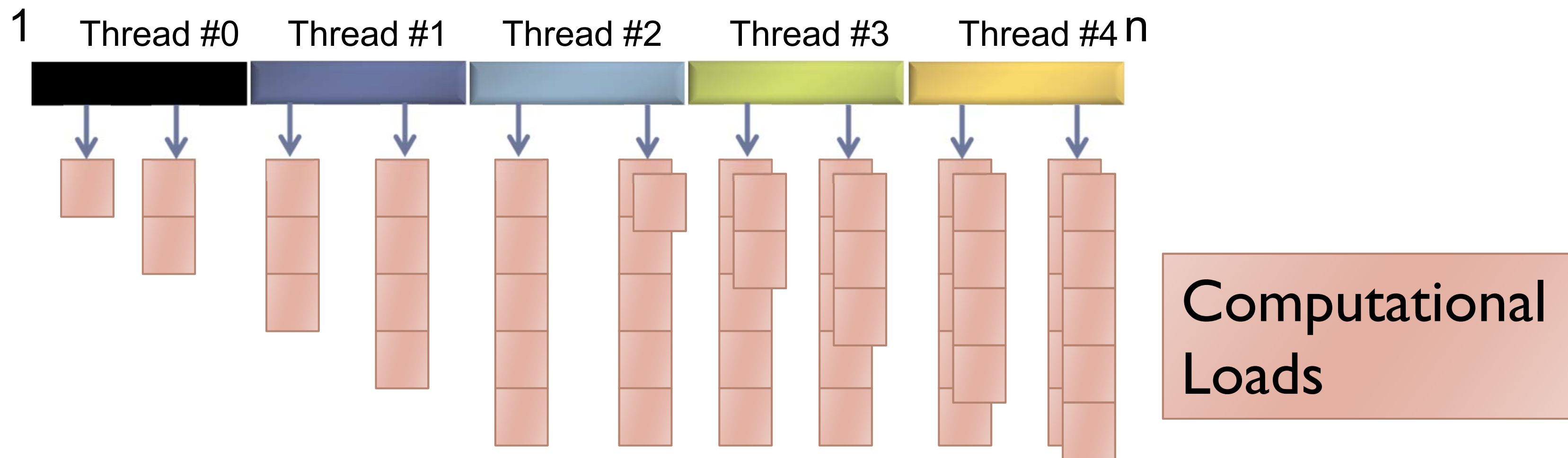
Global variables which allocate in each thread are specified in **parallel** construct.

Scheduling

What is scheduling? (1/2)

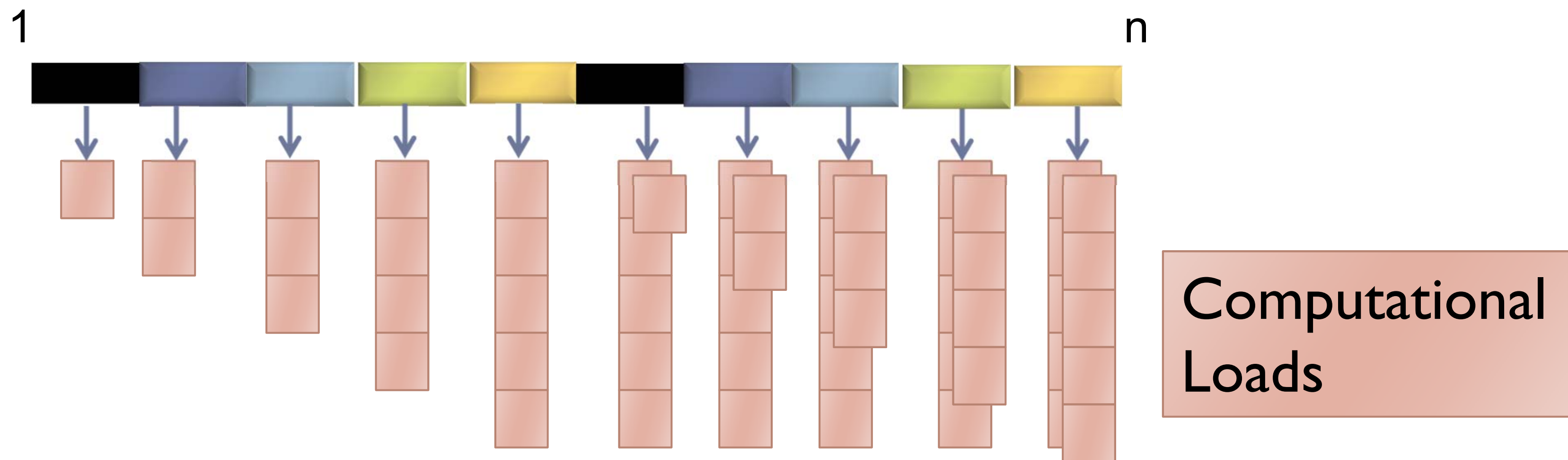
- In **parallel do** construct (**parallel for** construct), it divides length of target loop, such as from 1 to n, as continual manner, and it allocates the divided lengths to all threads.

- If computational loads allocated in each iteration are not balanced, parallel efficiency goes poor.



What is scheduling? (2/2)

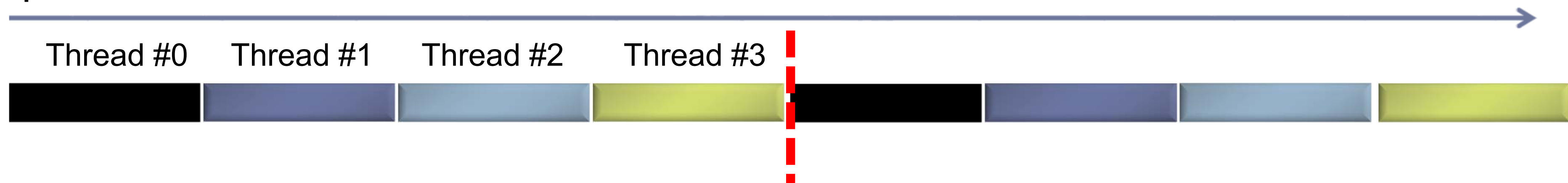
- To improve load balancing, sizes of allocated loop are shorten, and allocate them with cyclic manner.



- Optimal size of the allocated loop (we call this ***chunk size***) depends on computer hardware and target process.
- There is a clause to do the above allocation in OpenMP.

Loop scheduling and Its clause (1/3)

- schedule (static, n)
  - Divide loop with chunk size, and allocate them cyclic manner from thread #0, such as (thread #0, thread #1, …). This is called round-robin allocation. We can specify chunk size to n.
  - Without **schedule** clause (default), static is specified with loop length / number of threads as its chunk size.



Loop scheduling and Its clause (2/3)

- schedule(dynamic, n)
  - Loop length is divided by chunk size, and the allocation is performed by thread that finishes execution, in first come, first served manner.
  - We can specify chunk size to n.

Loop scheduling and Its clause (3/3)

- schedule(guided, n)
  - First, loop length is divided by chunk size, and the allocation is performed by thread that finishes execution in first come, first served manner. The divided loop length is getting smaller according to loop count. We can specify first chunk size to n.
  - If specified chunk size is 1, chunk size is specified with remainder loop length / number of threads.
  - Chunk size is exponentially reduced toward to 1.
  - If we specify k > 1 to chunk size, the chunk size is exponentially reduced toward to k. The last size of chunk may be smaller than k.
  - If chunk size is not specified, the chunk size is set to 1.

How to use loop **schedule** clause?

- Fortran90 Language

Because iteration number of j-loop is determined by indirect accesses, it is not clear that whether computational loads of i-loop is equal or not until run-time. To do load balancing, we use **schedule** clause with **dynamic**.

```
!$omp parallel do private( j, k ) schedule(dynamic,10)
do i=1, n
   do j=indj(i), indj (i+1)-1
      y( i ) = amat( j ) * x( indx( j ) )
   enddo
enddo
!$omp end parallel do
```

- C Language

```
#pragma omp parallel for private( j, k ) schedule(dynamic,10)
for (i=0; i<n; i++) {
   for ( j=indj(i); j<indj (i+1); j++) {
      y[ i ] = amat[ j ] * x[ indx[ j ]];
   }
}
```

A Note of **schedule** clause in Programming

- Chunk size of dynamic and guided affects performance.
  - If we specify too small chunk size, we obtain nice load balancing, but system overhead is increase.
  - On the other hand, if we specify too big chunk size, we obtain bad load balancing, but system overhead is reduced.
  - Hence there is tread-off.
  - The tuning of chunk size at run-time is required; hence cost of tuning is increasing.
- High performance implementation with **static** clause only. (in some case)
  - There is no system overhead for **static** clause while there is system overhead for **dynamic** clause.
  - Implementation with **static** clause with the best loop length in advance is the best in some cases. However, cost of programming is increase.

An Example of Load Balancing with only **static** clause

- Apply to sparse matrix-vector product.

```
!$omp parallel do private(S, J_PTR,I)
    DO K=1, NUM_SMP
      DO I=KBORDER(K-1)+1, KBORDER(K)
        S=0.0D0
        DO J_PTR=IRP(I), IRP(I+1)-1
          S=S + VAL( J_PTR ) * X(ICOL( J_PTR))
        END DO
        Y(I)=S
      END DO
    END DO
!$omp end parallel do
```

Loop for number of threads:
To know loop length for each thread.

Loop length for each thread by execution in advance:
Non-uniform length for each thread is specified.

Problem that can be load balancing with continues loop in each thread before execute-time is adaptable of this implementation.
:It cannot use a case that load varies dynamically.

Notes to Programming with OpenMP (General Matters)

A Note of Programming with OpenMP

- Main work of parallelization with OpenMP is:

- To parallelize program with **parallel** construct to simple for loop.

  - Parallelizing complex loops with OpenMP lacks merit of OpenMP, since it requires high cost of programming.

- To establish the above, the parallelization with **parallel** construct needs to understand:

  Correct use of **private** clause

  to avoid bugs.

A Note of **private** clause (1/2)

- Variables are treated as shared in default except for declaring variables with **private** clause.
  - The default variables do not allocated in each thread.

● e.g.) Shared variables for loop induction variables.

```
!$omp parallel do
do i=1, 100
   do j=1,  100
      tmp = b(i) + c(i)
      a( i ) = a( i ) + tmp
   enddo
enddo
!$omp end parallel do
```

Only the variable i is allocated in a private without declaration.

The variable j is allocated in a shared without **private** clause.
← Addition is done with fast come fast served manner.
← A bug occurs at run-time.

The variable tmp is allocated in a shared without **private** clause.
← Substitution is done with fast come fast served manner.
← A bug occurs at run-time.

A Note of **private** clause (2/2)

- If you make a function for target process and increase arguments to the function to reduce number of variables should be described in **private** clause, you may obtain no effect of thread parallelization due to high overhead of the function calling.
  - e.g.) Too many arguments of function.

```
!$omp parallel do
do i=1, 100
   call foo(i,arg1,arg2,arg3,
    arg4,arg5, ….., arg100)
enddo
!$omp end parallel do
```

Arguments of function are treated as private, hence number of variables for **private** clause can be reduced.
← But, the overhead of calling is increase.
← Speedup factor is limited due to high calling overhead of function when thread execution.
*A solution:
Using global variables to reduce arguments.

Summary of Notes of **private** clause

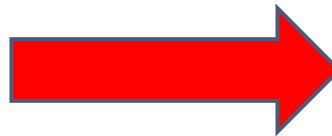
- In OpenMP, all variables without declaration are shared variables.
- Global variables in C language, and variables declared by common in Fortran are also shared variables.
  - To make private variables, declaration with "Threadprivate" is needed.
- If a case to parallelize the outer loop with **parallel** construct:
  - Variables declared in calling functions (or procedures) inside a loop are private.
  - In C language, explicit declarations inside a loop are private.
    - e.g.) int a;

A Note of Nested Loops for **parallel** construct (1/2))

- We can separate parallel construct with **do**.
- If target is one loop, there is compiler to generate code with lower performance to non-separated code. One of the reasons is: the compiler makes a code with fork in separated part in every iteration. However, there is a case that totally opposite case. Hence we need to check both performance.

```
!$omp parallel
!$omp do private(j,tmp)
do i=1, 100
   do j=1,  100
      tmp = b( j ) + c( j )
      a( i ) = a( i ) + tmp
   enddo
enddo
!$omp end do
!$omp end parallel
```

If target is one loop, we can specify it with **parallel do**.

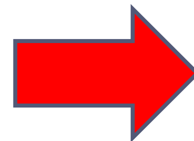


```
!$omp parallel do private(j,tmp)
do i=1, 100
   do j=1,  100
      tmp = b( j ) + c( j )
      a( i ) = a( i ) + tmp
   enddo
enddo
!$omp end parallel do
```

A Note of Nested Loops for **parallel** construct (2/2)

- We can separate parallel construct with **do**.
- If target of parallelization is the inner loop, separated is faster.
  - If the outer loop can be parallelized, then the best target to be parallelized is the outer loop.
  - If there is a data dependency for the outer loop, then it cannot be parallelized for the outer loop.

```
do i=1,  n
!$omp parallel do
  do j=1,  n
     <A parallelizable
       statements. >
  enddo
!$omp end parallel do
enddo
```



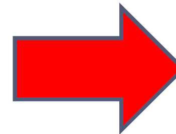
```
!$omp parallel
do i=1,  n
!$omp do
  do j=1,  n
     <A parallelizable
       statements. >
  enddo
!$omp end do
enddo
!$omp end parallel
```

An Example of Braking Data Dependency

- e.g.) Summation to arrays with indirect accesses.
  - Programmer may judge correct execution according to pattern of indirect accesses and timings of threads execution.
    - Theoretically it is wrong.
  - OpenMP system does not provide any consistency of data.
    - To keep consistency of data, we need mutual exclusion by **critical** construct or others.

- e.g.) A wrong code.

```
!$omp parallel do private( j )
do i=1, n
   j = indx( i )
   a( j ) = a( j ) + 1
enddo
!$omp end parallel do
```



```
!$omp parallel do private( j )
do i=1, n
   j = indx( i )
!$omp critical
   a( j ) = a( j ) + 1
!$omp end critical
enddo
!$omp end parallel do
```

Speed down by **critical** construct

- If we use **critical** construct, performance will be down basically. In particular, it is remarkable when it runs with high number of threads.
  - If CPU provides hardware support for **atomic** construct, implementation with **atomic** construct is faster in some cases. However in this case, performance also goes down if we use more threads.
- To establish high performance, modification of algorithm is needed basically.
- There are the following three strategies.

1. Removing **critical** construct by limiting accesses within thread.
   - Algorithm is modified to refer local region of allocated data in each thread for indirect accesses in theoretical.
2. Minimizing access between threads.
   - Reducing number of threads to enter parallel region of **critical** construct at same time. Check data access pattern for indirect accesses in advance, then change data for indirect access to do that.
3. Separate the part to access inter threads, then it remakes sequential code.
   - e.g.) **reduction** clause for dot products.

Drawbacks of Parallelization with OpenMP (1/2)

- OpenMP is basically designed to parallelize simple loops.
- Parallelization of complex loops from real applications may be difficult to implement directives by OpenMP.

1. Number of variables that should be specified in **private** clause goes big number.
   - Variables specified by inner loops are usually many to parallelize them for the outer loop.
   - Some compilers do not print errors for missing declaration of variables for **private** clause, since duties are owned by user.
   - If you miss the declaration, you see different results to results by sequential execution. This means that debugging gets difficult.
   - A Solution: Verify parallelization from logs of optimization by compiler.

Drawbacks of Parallelization with OpenMP (2/2)

2. It is difficult to obtain high performance if we execute code with high number of threads.
   - Again, performance is down if we use more than 8 threads in experimental knowledge in current CPUs.
     1. Low memory bandwidth to establish low power.
     2. There is no parallelism for target loops. (Length of the loops are short.)
   - To solve the above problem, we need modifications of algorithm and implementation. This means that merits of OpenMP are lost, such as easy implementation.
3. Basically, OpenMP is not suited for complex thread parallelization.
   - Since OpenMP is designed to parallelize simple kernel loops for numerical computation, such as using **parallel for** construct.
   - If you need to implement complex process, it is better to use native thread APIs, such as **Pthread**.

Examples from Real Codes

e.g.) Matrix-Matrix Multiplication with OpenMP Parallelization (C Language)

```
#pragma omp parallel for private (j, k)
    for(i=0; i<n; i++) {
      for(j=0; j<n; j++) {
        for(k=0; k<n; k++) {
          C[i][j] += A[i][k] * B[k][j];
        }
      }
    }
```

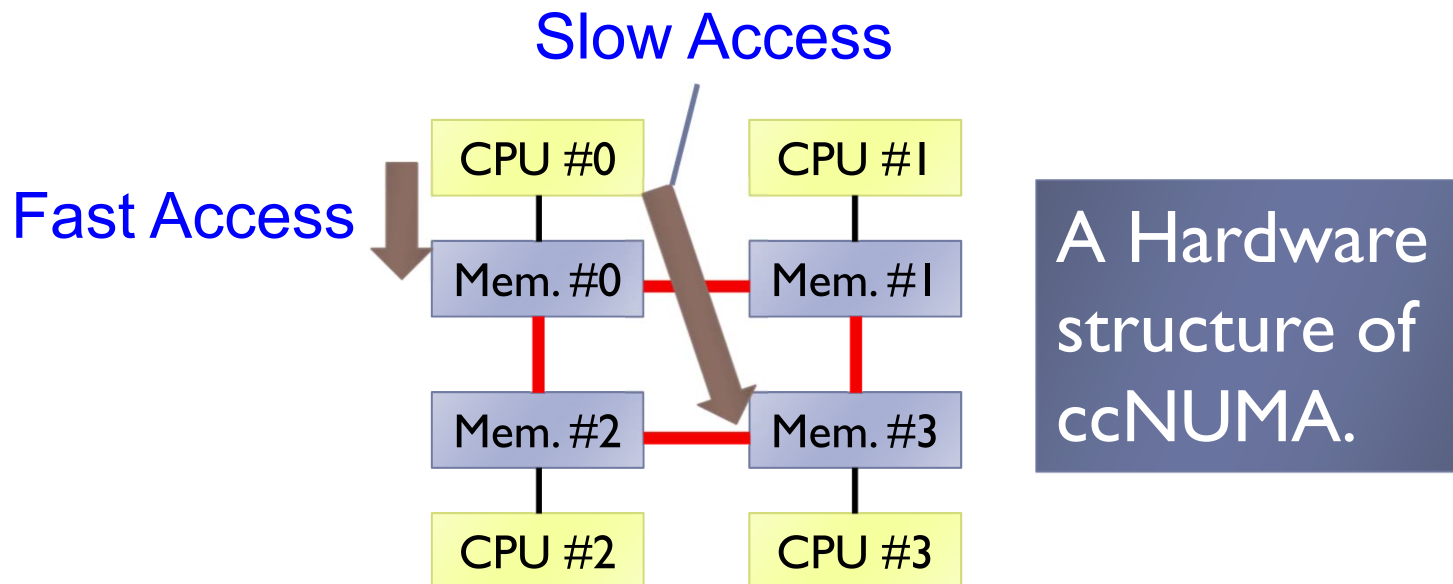
e.g.) Matrix-Matrix Multiplication with OpenMP Parallelization (Fortran Language)

```
!$omp  parallel  do  private (j, k)
do i=1, n
    do j=1, n
        do k=1, n
            C(i, j) = C(i, j) + A(i, k) * B(k, j)
        enddo
    enddo
enddo
!$omp end parallel do
```

A High Performance Implementation: First Touch

What is "First Touch"

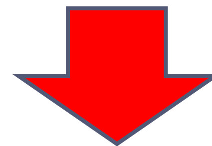
- First Touch is a memory optimization technique for shared parallel machines, which are consist of ccNUMA (Cache Coherent Non-Uniform Memory Access).
- One of important techniques for parallel programing with OpenMP.
- By using nature of memory structure of ccNUMA.



A Hardware structure of ccNUMA.

Why Fast Touch is effective?

- In hardware of ccNUMA, allocated array is assigned to memory that is most near from core which accesses the array at first time.
- By using this nature, initialize the array by using OpenMP at first time in the program with same data access pattern of main computations for the array. After the initialization, the array is assigned to nearest memory for the core to be computed.



- Fast Touch can be implemented with same loop structure for the main computation to initialize array, such as zero or data settings.

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

e.g.) First Touch (C Language)

```
#pragma omp parallel for private( j )
for (i=0; i<100; i++) {
  for (j=0;  j<100; j++) {
    a[ i ] = 0.0;
    amat[ i ][ j ] = 0.0;
}
….

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

Initialization for First Touch. This needs to implement first part of the program.

Main computation with first-touched data.

e.g.) First Touch (Fortran Language)

```
!$omp parallel do private( j )
do i=1, 100
  do j=1,  100
     a( i ) = 0.0d0
     amat( i , j ) =0.0d0
  enddo
enddo
!$omp end parallel do
….
!$omp parallel do private( j )
do i=1, 100
  do j=1,  100
     a( i ) = a( i ) + amat( i , j ) * b( j )
  enddo
enddo
!$omp end parallel do
```

Initialization for First Touch. This needs to implement first part of the program.

Main computation with first-touched data.

Effect of First Touch

- T2K Open Supercomputer (16 cores / node)
- The AMD Quad Core Opteron (Barcelona)
  - 4 sockets, 4 cores per socket, total is 16 cores, ccNUMA.
- A sparse matrix-vector multiplication. This is same implementation of numerical library *Xabclib*.

```
!$omp parallel do private(S,J_PTR,I)
    DO K=1, NUM_SMP
      DO I=KBORDER(K-1)+1,KBORDER(K)
        S=0.0D0
        DO J_PTR=IRP(I),IRP(I+1)-1
          S=S+VAL( J_PTR ) * X(ICOL( J_PTR ))
        END DO
        Y(I)=S
      END DO
    END DO
!$omp end parallel do
```

Indexes of rows of each thread for the sparse matrix.

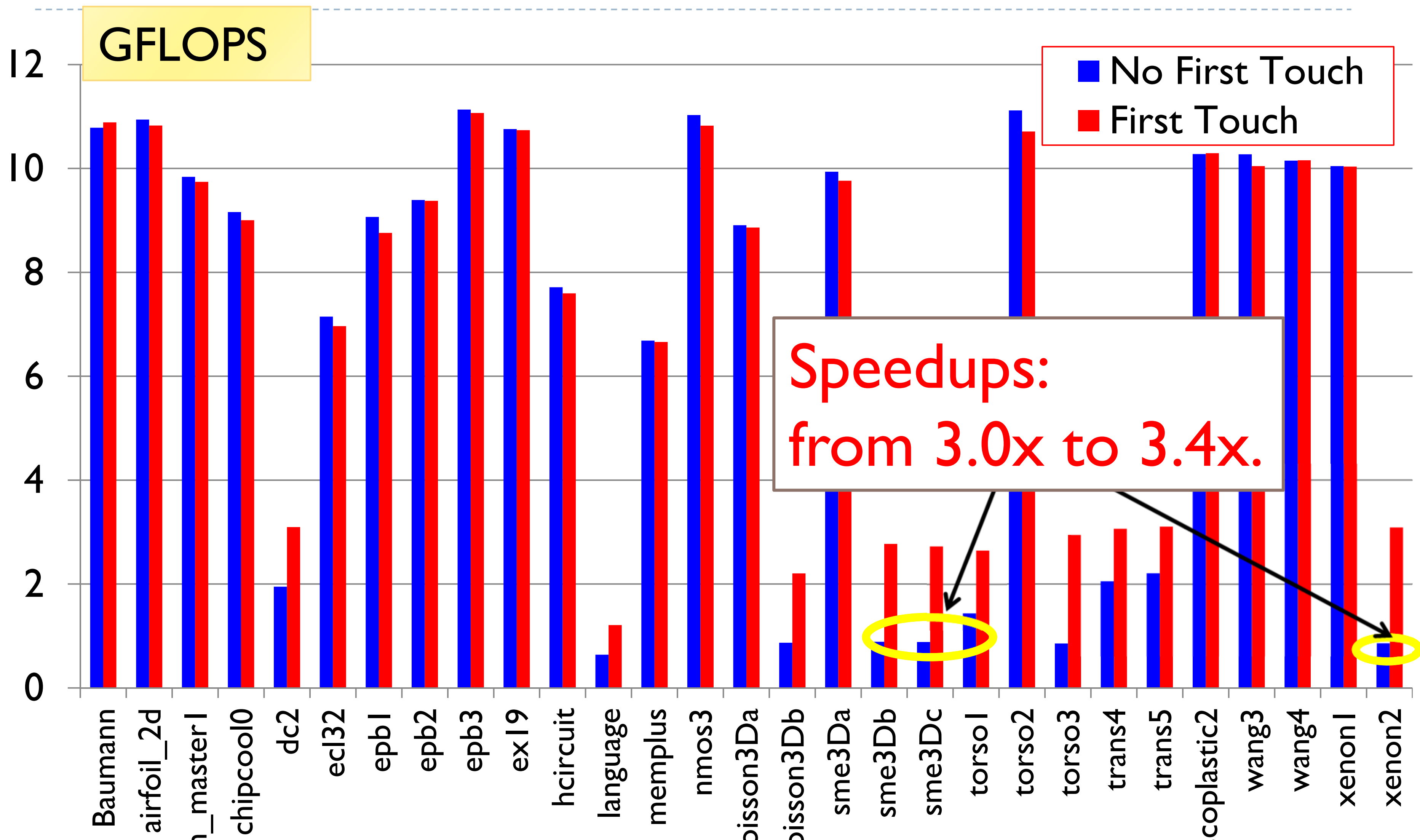
Accesses to non-zero elements to be computed.

Computation for sparse matrix-vector multiplication.

Indexes of Right Hand Side *b*. (Indirect accesses)

Sparse matrix format: CRS (Compressed Row Storage)

Effect of Sparse Matrix-vector Multiplication with First Touch (AMD Quad Core Opteron, 16 Threads)



Matrices that effects well for First Touch

- sme3Da
  - http://www.cise.ufl.edu/research/sparse/matrices/FEMLAB/sme3Da.html
  - Location of non-zero elements is distributed.
  - number of rows:12,504
  - Very small size.
- xenon2
  - http://www.cise.ufl.edu/research/sparse/matrices/Ronis/xenon2.html
  - Almost "tri-diagonal"

←Matrix *A* is optimized, and RHS *b* is on cache memory.

A tri-diagonal matrix. ←By using nature of ccNUMA , matrix *A* and RHS *b* can be optimized for both allocation.

A Note of implementation of First Touch

- There is no gain except for ccNUMA architectures.
  - The FX10 and K-computer are NOT ccNUMA; hence no gain.
- There is no gain expect for "hand made" code; Programmer needs to take care of allocations of arrays and computations by himself or herself.
  - In case of using numerical libraries:
    - Programmer prepares arrays (or matrices).
    - In natural procedure of this, setting arrays at first, then call a numerical library.
    - In the above process, programmer cannot know access patterns of main computation; since library is provided by a binary library.
    - Hence programmer cannot implement initialization with same access pattern of main computations within the library.
    - Since the above reasons, we cannot implement First Touch.

A Quick Note of OpenMP 4.0

OpenMP 4.0

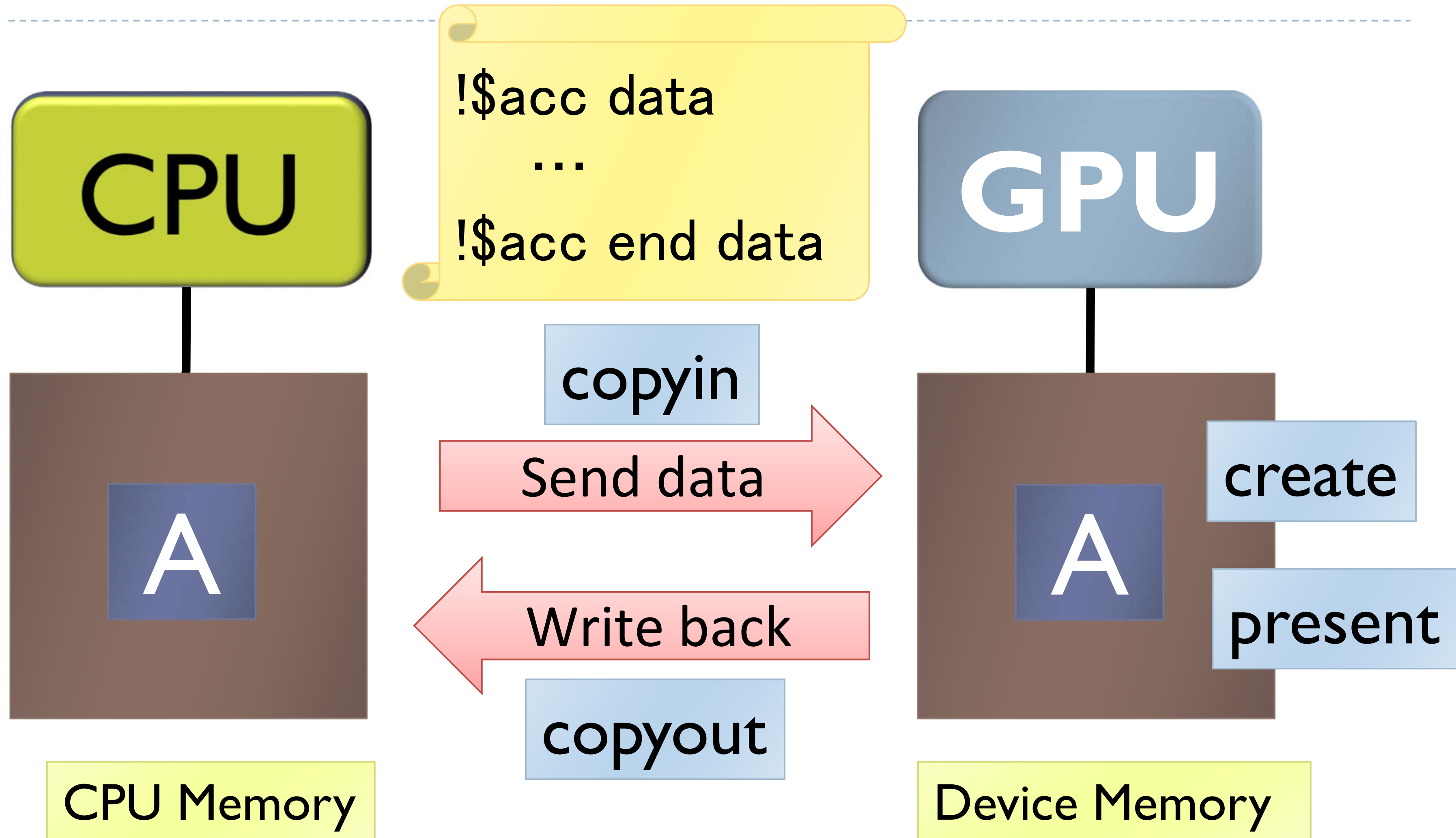
- Specification was opened in July 2013.
  - http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
- Specifying offloading of devices, such as GPUs for computations of OpenMP:
  - **target** construct
- Specifying multi parallel devices:
  - **terms** clause
- Specifying SIMD operations:
  - **simd** construct
- Specifying allocation between threads and cores (NUMA affinity):
  - **proc_bind** clause
- For using GPU, OpenACC is providing same functions.
  (See next slides.)

Towards to OpenACC

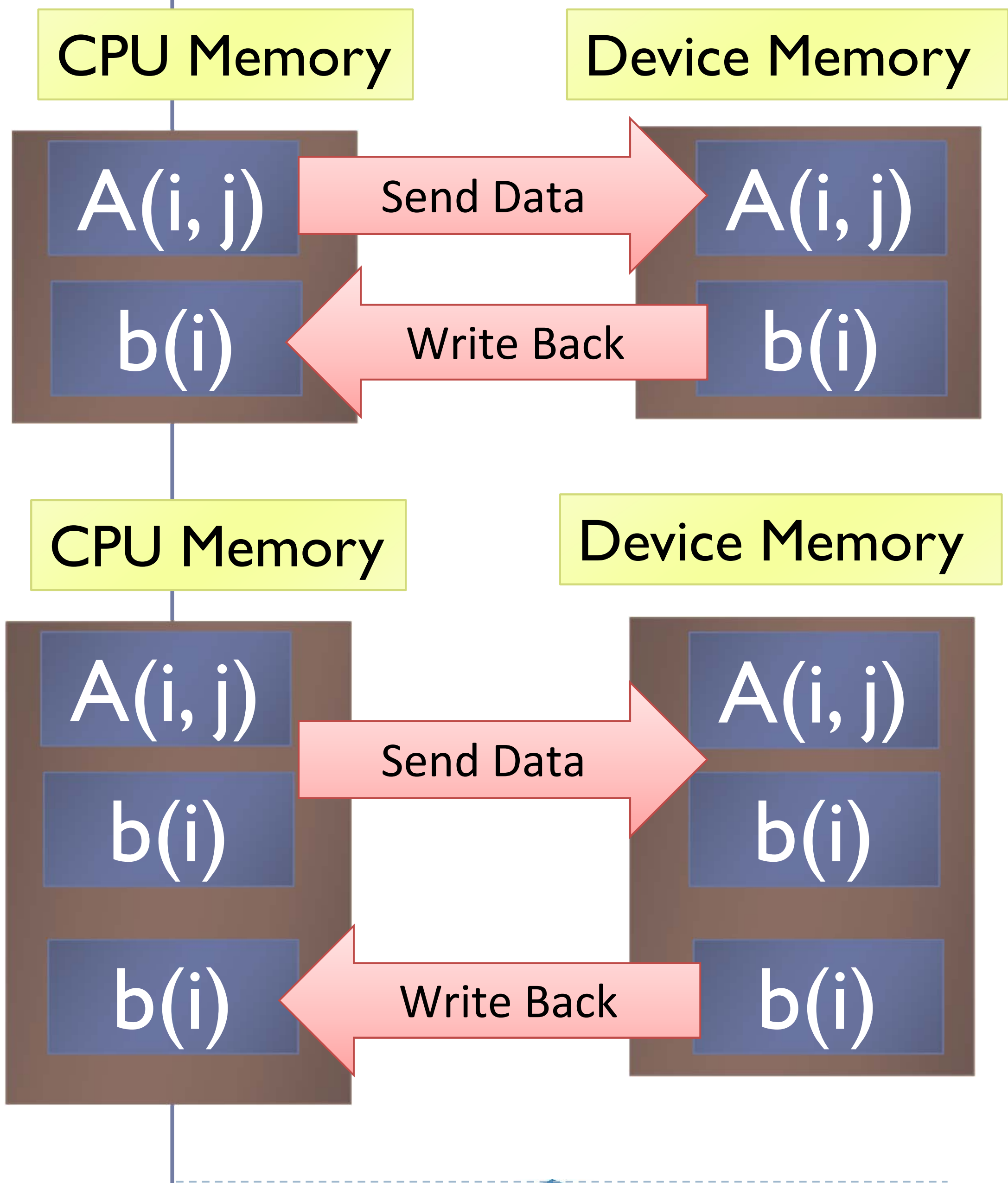
Overview of OpenACC

- OpenACC, which can be treated with GPU with directives like OpenMP, is getting pervasive.
  - I do not predict that which will be widely used between OpenMP 4.0 and OpenACC.
- It is easy translated to OpenACC if you have parallelized with OpenMP.
  - **parallel** construct in OpenMP
    → **kernel** construct or **parallel** construct in OpenACC.
- Note to be implemented in OpenACC:
  - Minimize data movement from CPU to GPU, and from GPU to CPU.
  - To minimize the data movement, we need to use **data** construct for target arrays.

Data flows of **data** construct



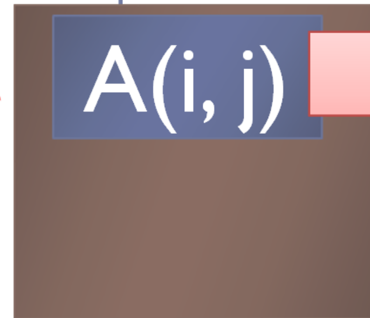
```
  do iter = 1, MAX_ITER
!$acc kernels
    do i=1, n
      do j=1, n
        b(i) = A(i, j) ∗ …
      enddo
    enddo
!$acc end kernels
   …
!$acc kernels
   do i=1, n
      do j=1, n
        b(i) = b(i) + A(i, j) ∗ …
      enddo
    enddo
!$acc end kernels
   …
 enddo
```



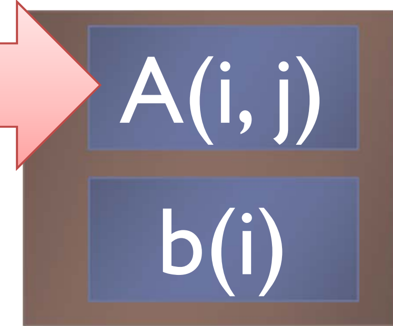
```
!$acc data copyin(A) create(b)
  do iter = 1, MAX_ITER
!$acc data present(A, b)
!$acc kernels
    do i=1, n
       do j=1, n
         b(i) = A(i, j) * …
       enddo
    enddo
!$acc end kernels
!$acc end data
   …
!$acc data present(A, b)
!$acc kernels
   do i=1, n
       do j=1, n
         b(i) = b(i) + A(i, j) * …
       enddo
    enddo
!$acc end kernels
!$acc end data
   …
  enddo
!$acc end data
```

CPU Memory

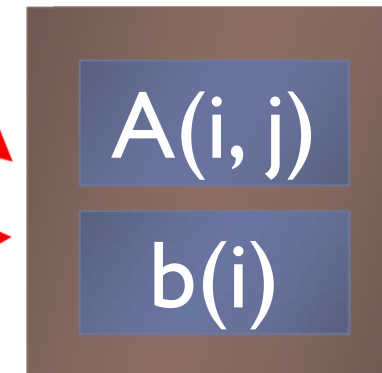
Device Memory



Send Data



Device Memory



Computation with data on device memory. (There is no sending data from CPU, and writing back to CPU memory.)