Fundamentals of OpenMP & MPI Parallel Programming

Jianhua Gao Beijing Normal University

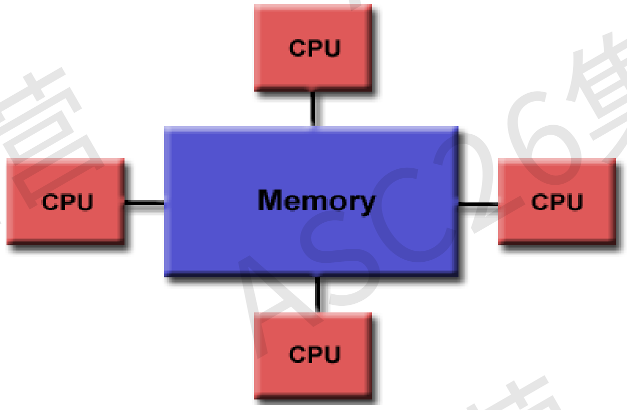
2026/1/27

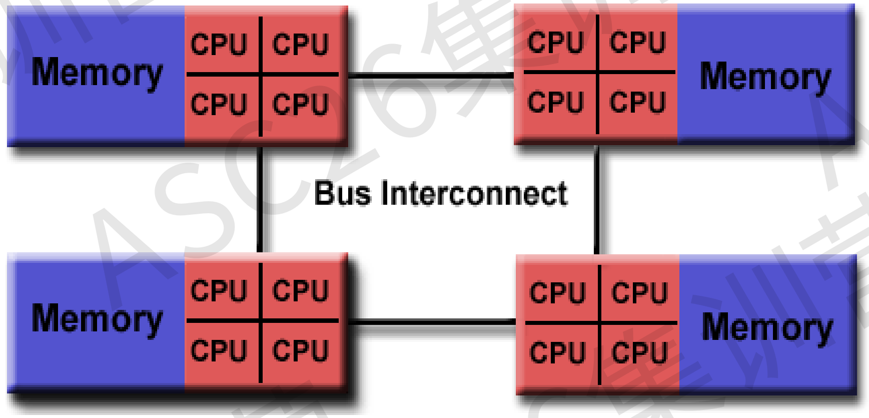
Background

- **Computer system (classified from the perspective of storage models)**
  - Shared memory computer systems
  - Distributed memory computer systems
  - Hybrid distributed-shared memory computer systems

Background

- **Computer system (classified from the perspective of storage models)**
  - Shared memory computer systems
    - UMA (Uniform Memory Access)
    - NUMA (Non-Uniform Memory Access)



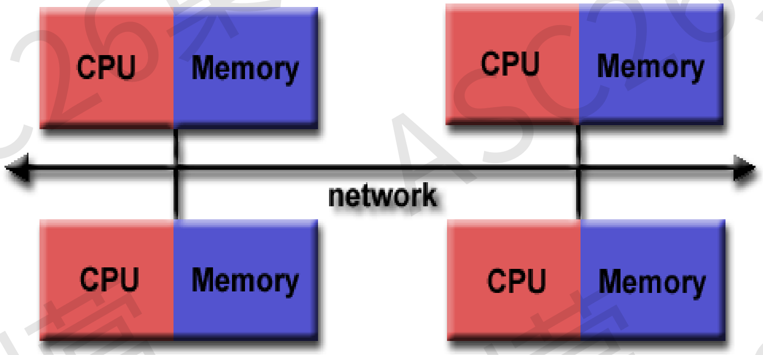


Background

■ **Computer system (classified from the perspective of storage models)**

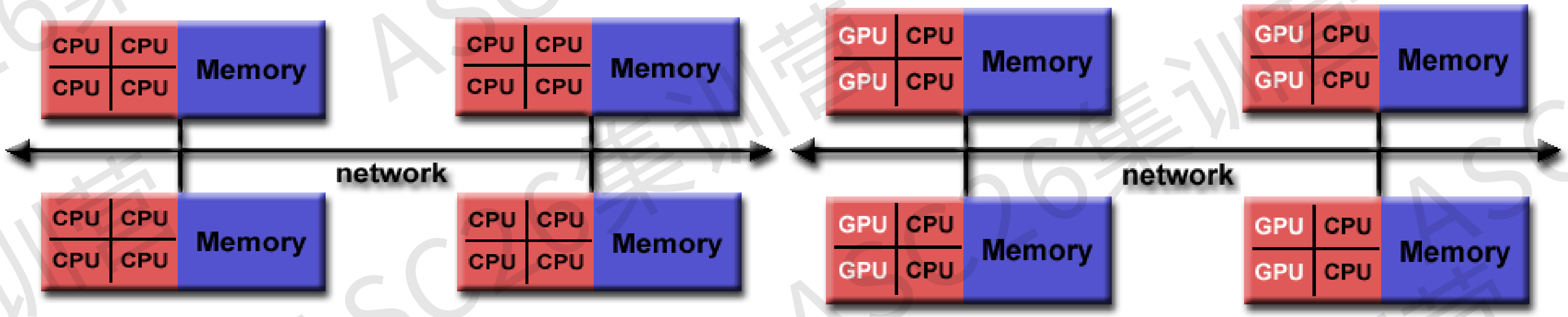
➢ Distributed memory computer systems

- Multiple nodes are connected together through a network, with each node's processor having its own local memory.
- Compared to shared memory systems, distributed systems offer excellent scalability.

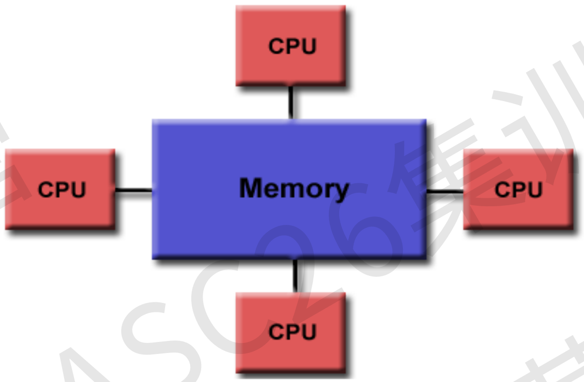


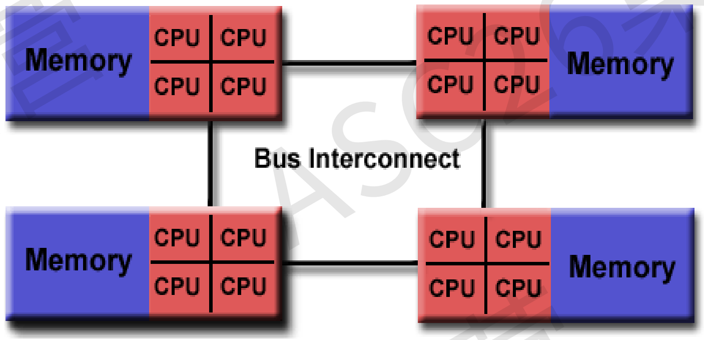
Background

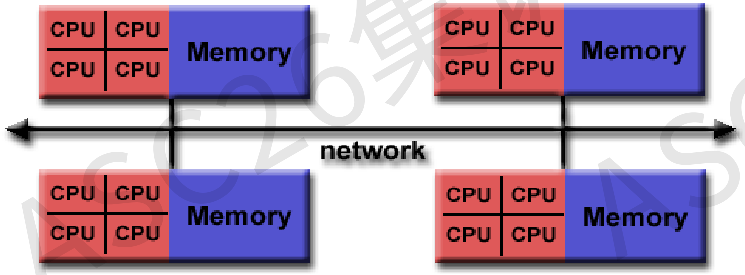
- **Computer system (classified from the perspective of storage models)**
  - Hybrid distributed-shared memory computer systems

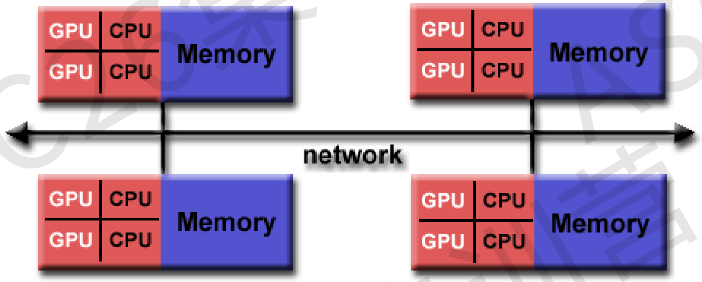


Background


CPU
CPU
Memory
CPU
CPU


Memory
CPU CPU CPU CPU
CPU CPU CPU CPU
Memory
Bus Interconnect
Memory
CPU CPU CPU CPU
CPU CPU CPU CPU
Memory


CPU CPU CPU CPU
Memory
CPU CPU CPU CPU
Memory
network
CPU CPU CPU CPU
Memory
CPU CPU CPU CPU
Memory


GPU CPU GPU CPU
Memory
GPU CPU GPU CPU
Memory
network
GPU CPU GPU CPU
Memory
GPU CPU GPU CPU
Memory

Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

01 OpenMP Introduction

■ What is OpenMP ?

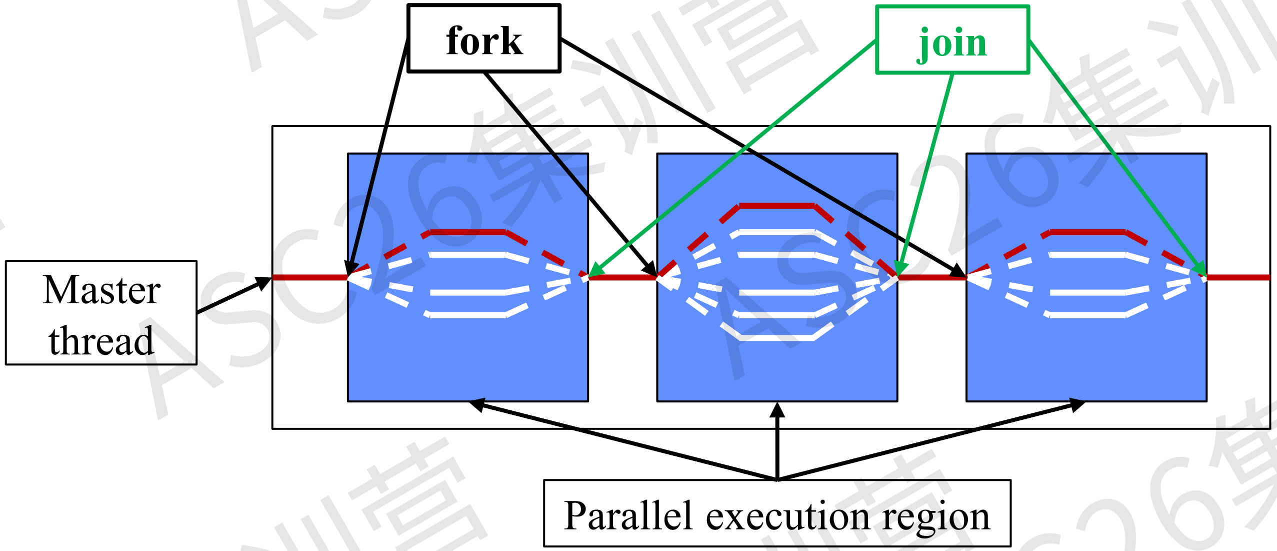
- A parallel programming model designed for shared-memory system
- An industry-standard API for multithreaded programming
- Consisting of a set of compiler directives, runtime library routines, and environment variables
- Facilitating multithreaded programming in Fortran, C, and C++
- Offering simplicity in programming, good portability, and excellent scalability

OpenMP™ www.openmp.org

01 OpenMP Introduction

■ Parallel Execution Model

- ➢ OpenMP is a thread-based parallel programming model
- ➢ It adopts the Fork-Join parallel execution model
- ➢ Utilizing thread pool technology, where multiple threads are initiated once the program starts.



01 OpenMP Introduction

■ OpenMP Example in C

```
#include <omp.h>
#include <stdio.h>
int main() {
  int nthreads,tid;
  #pragma omp parallel private(nthreads,tid)
  {
    tid=omp_get_thread_num();
    printf("Hello, world from OpenMP thread %d\n", tid);
    if (tid==0) {
      nthreads=omp_get_num_threads();
      printf(" Number of threads %d\n", nthreads);
    }
  }
  return 0;
}
```

- **Compiler directive: #pragma omp**
- **Header file: omp.h**

- **Compile**

```
gcc -fopenmp hello.c
icc -openmp hello.c
```

01 OpenMP Introduction

■ OpenMP Example in C

```
#include <omp.h>
#include <stdio.h>
int main() {
  int nthreads,tid;
  #pragma omp parallel private(nthreads,tid)
  {
    tid=omp_get_thread_num();
    printf("Hello, world from OpenMP thread %d\n", tid);
    if (tid==0) {
      nthreads=omp_get_num_threads();
      printf(" Number of threads %d\n", nthreads);
```

- Compiler directive: #pragma omp
- Header file: omp.h

```
Hello World from OpenMP thread 2
Hello World from OpenMP thread 0
Number of threads 4
Hello World from OpenMP thread 3
Hello World from OpenMP thread 1
```

01 OpenMP Introduction

■ OpenMP Example in Fortran

```
program hello
use omp_lib
implicit none
integer :: tid, nthreads

!$omp parallel private(tid)
    tid = omp_get_thread_num()
    write(*,100) "Hello, world from OpenMP thread ", tid
    if (tid==0) then
        nthreads=omp_get_num_threads();
        write(*,100) "Number of threads ", nthreads
    endif
!$omp end parallel

100 format(1X,A,I1,/)
end
```

- Compiler directive: **!$omp**
- Module: **omp_lib**

- Compile

```
gfortran -fopenmp hello.f90
ifort -openmp hello.f90
```

01 OpenMP Introduction

Compiler Directives

- OpenMP achieves parallelization by adding compiler directives to serial programs.
- Compiler directives consist of three parts: a directive prefix, a directive itself, and clauses, with the general format:

#pragma omp	directive-name	[clause, ...]
Directive Prefix. Such a prefix is required for all directives.	**Directive**. A valid directive must appear between the directive prefix and its clauses.	**Clauses**. In the absence of other constraints, clauses can be unordered. This part may also be omitted entirely.

01 OpenMP Introduction

- **Compiler Directives can be broadly categorized into four types**

  - **Parallel Region Directive**
    - Generates a parallel region, i.e., creates multiple threads to execute tasks in parallel.
    - All parallel tasks must be placed within a parallel region to be potentially executed in parallel.
  - **Work-Sharing Directive**
    - Responsible for dividing tasks and distributing them among threads.
    - Work-sharing directives do not create new threads and must therefore be placed inside a parallel region.
  - **Synchronization Directive**
    - Handles synchronization between parallel threads.
  - **Data Environment**
    - Manages the attributes (shared or private) of variables within a parallel region, as well as data transfer between boundaries (serial regions and parallel regions).

01 OpenMP Introduction

- **Most OpenMP compiler directives apply to the structured block that follows them**
  - ➢ A structured block is a block of statements with only one entry point (at the top) and one exit point (at the bottom), with no branches that jump outside the block.
  - ➢ Exception: Fortran's STOP statement and C/C++'s exit() are allowed within a structured block.

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
more: res[id] = do_big_job (id);
  if (conv (res[id]) goto more;
}
printf (“All done\n”);
```

A structured block
One entry point, one exit point

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Not a structured block
Two entry points, one exit point

Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

02 Parallel Region

- Code inside a parallel region will be executed in parallel by multiple threads.
- At the end of the parallel region, there is an implicit synchronization (barrier).
- Clauses are used to specify additional information for the parallel region. If there are multiple clauses, they are separated by spaces.
- Specific format:

```
#pragma omp parallel [clause[[,]clause]…]
clause=
        if (scalar_expression)
        private (list)
        shared (list)
        default (shared | none)
        firstprivate (list)
        reduction (operator: list)
        copyin (list)
```

Fortran	**!$omp parallel [clause clause ...]** ***structured-block*** **!$omp end parallel**

C/C++	**#pragma omp parallel [clause clause ...]**
	{
	    structured-block
	}

02 Parallel Region

■ OpenMP Example in C

```
#include <omp.h>
#include <stdio.h>
int main() {
  int nthreads,tid;
  #pragma omp parallel default(shared) private(nthreads,tid)
  {
    tid=omp_get_thread_num();
    printf("Hello, world from OpenMP thread %d\n", tid);
    if (tid==0) {
      nthreads=omp_get_num_threads();
      printf(" Number of threads %d\n", nthreads);
    }
  }
  return 0;
}
```

Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

03 Work Sharing

- **OpenMP uses work-sharing compiler directives to divide and assign tasks to multiple threads for parallel execution**

- **Work-sharing directives are mainly divided into three categories:**
  - **omp for:** Responsible for partitioning and distributing loop tasks.
  - **omp sections:** Specifies a parallel region containing multiple structured blocks that can be executed in parallel.
  - **omp task:** Explicitly defines a task, which is then placed into a task queue to await execution.

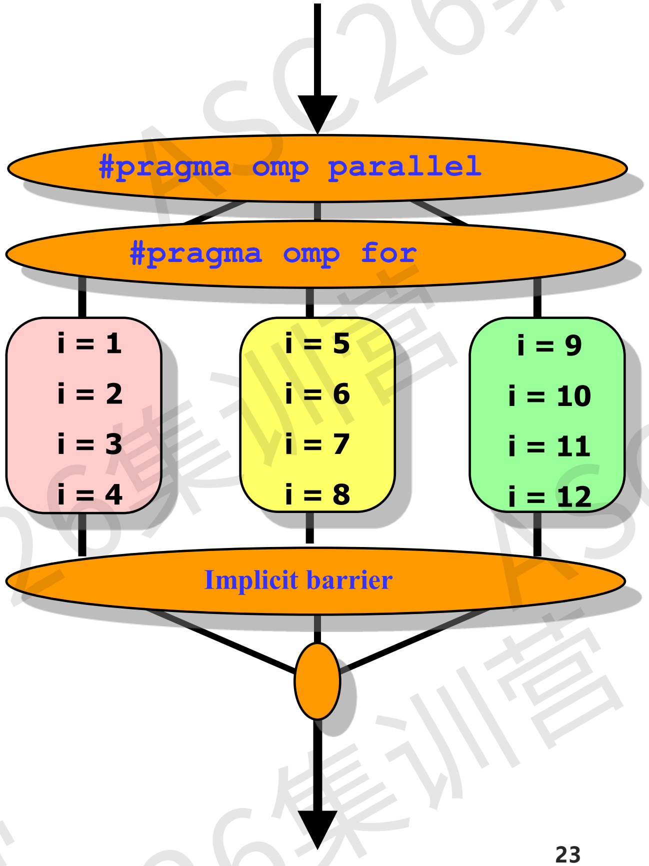
03 Work Sharing

- **When using work-sharing directives, the following points should be noted:**
  - Work-sharing directives are not responsible for creating or managing parallel regions; they must be used within a *`#pragma omp parallel`* region.
  - If a work-sharing directive region is not placed inside a parallel region, it will only be executed serially by a single thread.
  - An implicit barrier synchronization exists at the end of the work-sharing region (this can be avoided by using the *`nowait`* clause).

03 Work Sharing——omp for

- **omp for directive**
  - Each thread is assigned a set of loop iterations, and there are no dependencies between the iterations.
  - There is an implicit synchronization barrier at the end of the loop execution.

```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
       c[i] = a[i] + b[i];
```



03 Work Sharing——omp for

- The for directive specifies that the following loop is executed in parallel by threads in the thread pool.
- Specific format：

```
#pragma omp for [clause[[,]clause]…]
[clause]=
        Schedule(type [,chunk])
        ordered
        private (list)
        firstprivate (list)
        lastprivate (list)
        shared (list)
        reduction (operator: list)
        nowait
```

Fortran	**!$omp do [clause clause ...]** *do-loops* **!$omp end do**

C/C++	**#pragma omp for [clause clause ...]**
	{
	for-loops
	}

03 Work Sharing——omp for

- The parallel region and the work-sharing construct can be combined into a single compiler directive.

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```

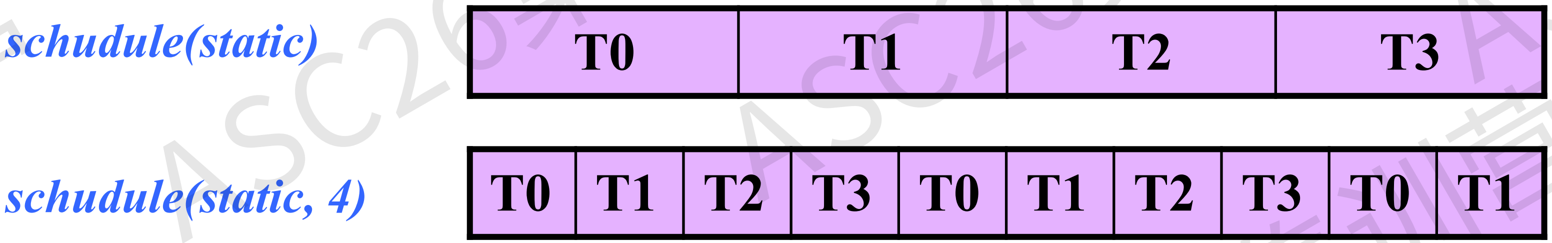
03 Work Sharing——omp for

- *schedule* **clause**
  - The schedule clause describes how loop iterations are divided among the threads in the thread team.
  - Syntax: *schedule(type[, size])*
    - *type*: specifies the scheduling strategy.
    - *size*: specifies the chunk size (the number of loop iterations) and must be an integer.
  - Scheduling strategy types
    - *static*: Static scheduling; the chunk size is fixed and specified by size.
    - *dynamic*: Dynamic scheduling; the chunk size is fixed and specified by size.
    - *guided*: Dynamic scheduling; the chunk size decreases over time, with the minimum chunk size specified by size.
    - *runtime*: The scheduling strategy is determined at runtime.

03 Work Sharing——omp for

- *schedule(static, size)*
  - **Omitting size**: The iteration space is divided into chunks of equal (or near-equal) size, with each thread assigned one chunk.
  - **Specifying size**: The iteration space is divided into chunks of a specified size, which are then assigned to threads in a round-robin fashion.

Example: Given 4 threads and a total of 40 iterations:



03 Work Sharing——omp for

- ***schedule(dynamic, size)***
  - The iteration space is divided into chunks of a specified size, which are then assigned to threads on a first-come, first-served basis.
  - When size is omitted, the default value is 1.
- ***schedule(guided, size)***
  - Similar to dynamic scheduling, but the size of assigned chunks starts large and decreases over time, using the GSS (Guided Self-Scheduling) algorithm.
  - *Size* specifies the minimum chunk size; when size is omitted, the default value is 1.
- ***schedule(runtime)***

```
export OMP_SCHEDULE=DYNAMIC, 4;
```

  - The scheduling policy depends on the value of the *OMP_SCHEDULE* environment variable.
  - It is illegal to specify a size when using runtime.

03 Work Sharing——omp sections

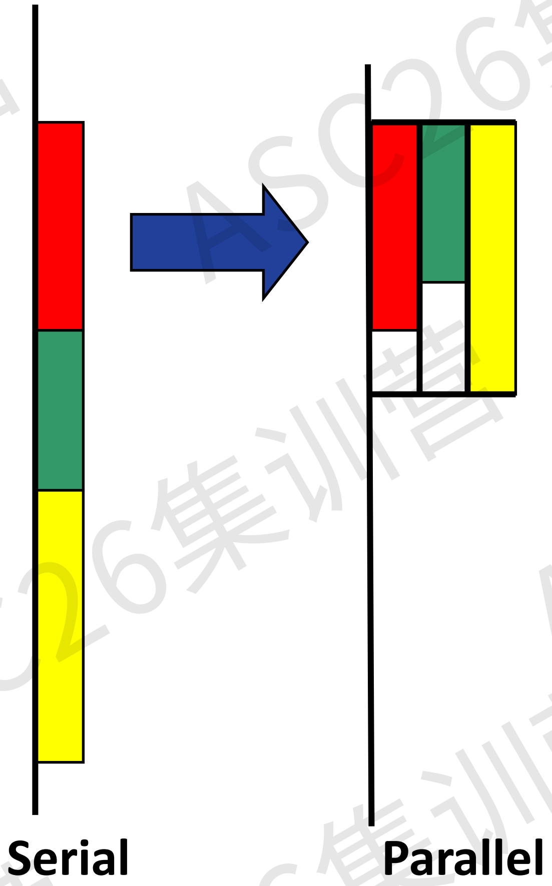
- Code between two *#pragma omp section* directives is called a section.
- Any number of sections can be defined, and they are assigned to multiple threads for concurrent execution.
- Each section is executed exactly once by only one thread.
- If the number of threads exceeds the number of sections, each thread executes at most one section.
- If the number of threads is fewer than the number of sections, each thread may execute more than one section.

```
#pragma omp sections clause1 clause2 ...
{
     #pragma omp section
     Structured block
     #pragma omp section
     Structured block
}
```

03 Work Sharing——omp sections

- omp sections example

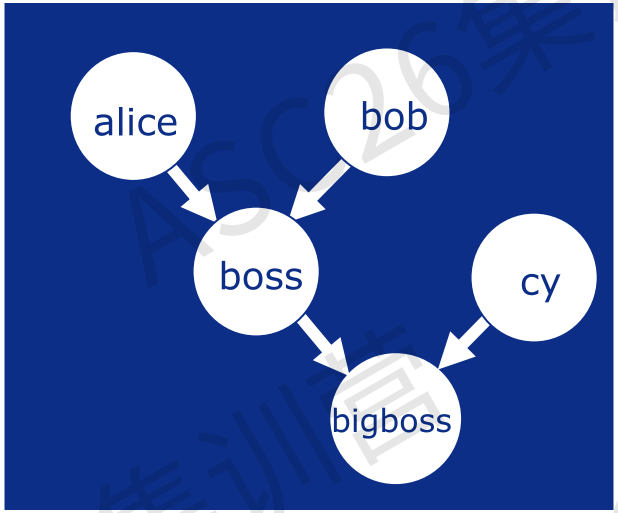
```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



03 Work Sharing——omp sections

■ **Functional Parallelism Example**

```
#pragma omp parallel sections
{
#pragma omp section /*Removable*/
   a = alice();
#pragma omp section
   b = bob();
#pragma omp section
   c = cy();
}
s = boss(a, b);
printf ("%6.2f\n", bigboss(s,c));
```

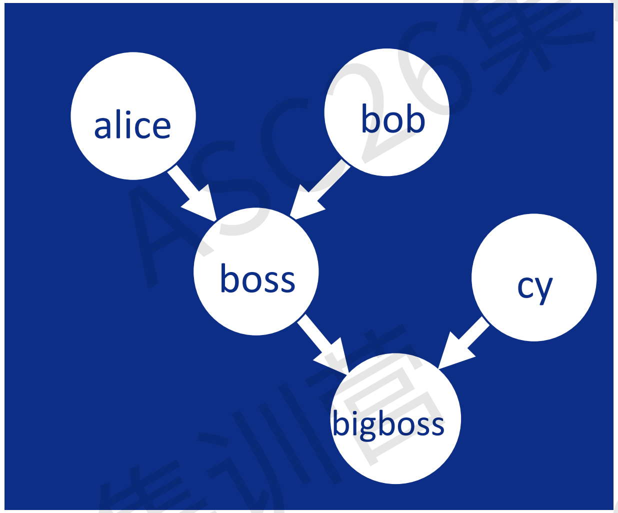


```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
     bigboss(s,c));
```

03 Work Sharing——omp sections

- **Functional Parallelism Example**

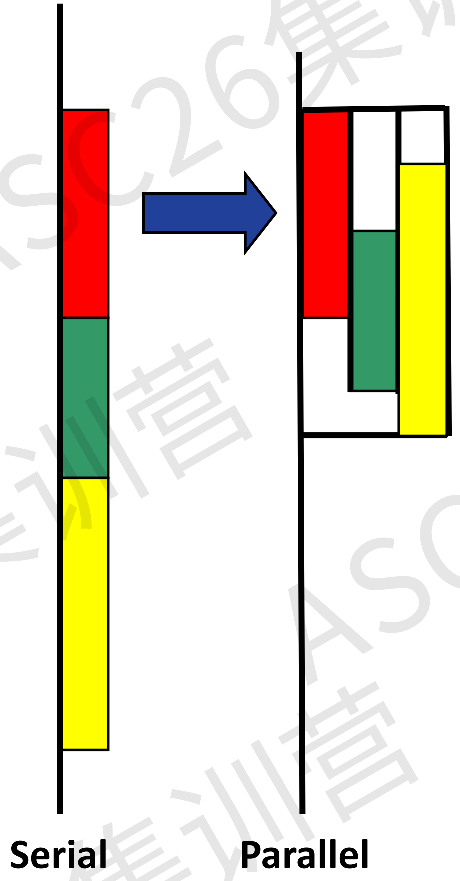
```
#pragma omp parallel sections
{
#pragma omp section  /*Removable*/
   a = alice();
#pragma omp section
   b = bob();
}
#pragma omp parallel sections
{
#pragma omp section  /*Removable*/
   c = cy();
#pragma omp section
   s = boss(a, b);
}
printf ("%6.2f\n", bigboss(s,c));
```



```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
     bigboss(s,c));
```

03 Work Sharing——omp task

- **New Features in OpenMP 3.0**
- **Ideal for irregular parallelism:**
  - Loops with undetermined boundaries
  - Recursive algorithms
  - Producer/Consumer patterns
- **#pragma omp task explicitly defines a task.**
  - A task is an independent unit of work that may be executed immediately by the encountering thread, or deferred to other threads in the thread pool.
  - Task execution depends on the OpenMP Task Scheduler at runtime.
  - The difference between *task* and *for/sections*: task defines work "dynamically."



03 Work Sharing——omp task

```
#pragma omp parallel num_threads(8)
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A thread pool of 8 threads is created.

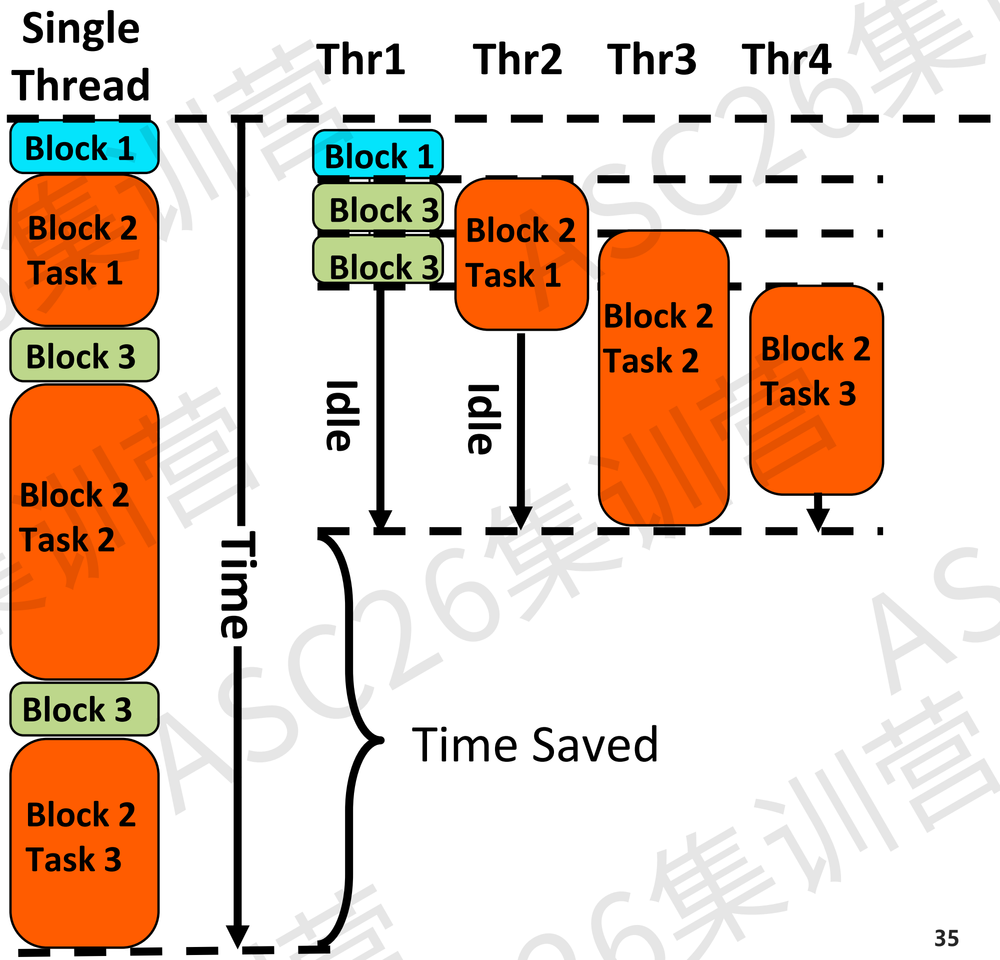
The code block specified by the *single* directive is executed by only one thread (the first one to reach it).

The thread executing the single block continuously generates tasks, which are then assigned to idle threads in the thread pool by the runtime system.

All threads synchronize at the end of the single code block.

03 Work Sharing——omp task

```
#pragma omp parallel
{
   #pragma omp single
   { // block 1
      node * p = head;
      while (p) {  //block 2
      #pragma omp task
         process(p);
      p = p->next;  //block 3
      }
   }
}
```



Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

04 Data Environment

- **Data Scoping: Which variables are shared across multiple threads, and which are private?**
- **OpenMP is a parallel programming model for shared-memory systems; threads communicate via shared variables.**
- **Shared Variables:**
  - C/C++: Global variables (file or namespace scope), static variables, constants, etc.
  - Fortran: COMMON blocks, SAVE variables, MODULE variables, etc.
- **Private Variables:**
  - Variables declared inside a parallel region (explicitly private).
  - Variables explicitly declared using the private clause.
  - Loop iteration variables.
  - Variables on the stack of functions called within a parallel region (local variables defined inside functions).

04 Data Environment

■ **Data Race Issues**

➢ The following loop cannot execute correctly.

```
#pragma omp parallel for
for(k=0;k<100;k++) {
   x=array[k];
   array[k]=do_work(x);
 }
```

➢ The correct approach:

- Declare as a private variable directly.

```
#pragma omp parallel for private(x)
for(k=0;k<100;k++) {
  x=array[k];
  array[k]=do_work(x);
}
```

- Declare variables inside the parallel region; such variables are inherently private.

```
#pragma omp parallel for
 for(k=0;k<100;k++) {
   int x;
   x=array[k];
   array[k]=do_work(x);
  }
```

04 Data Environment

- **Two ways to control the data environment during parallel execution:**
  - Independent OpenMP directives.
    - #pragma omp threadprivate(list)
  - Data-sharing attribute clauses of OpenMP directives.
    - private
    - shared
    - default
    - firstprivate
    - lastprivate
    - copyin
    - reduction

04 Data Environment

threadprivate Directive

- Used to declare a global or static variable as private to each thread (rather than private to a specific parallel region). For example, a Thread ID that retains the same value across multiple parallel regions.
- The directive must immediately follow the variable declaration.

```
int A(100), B;
double C;
#pragma omp threadprivate(A, B, C)
```

- **Execution Mechanism:**
  - When the program first enters a parallel region, each thread creates a copy of the variable marked as threadprivate.
  - The initial value of each copy is unpredictable (random memory address, uninitialized).

```
int a;
#pragma omp threadprivate(a)
#pragma omp parallel
{
   a = OMP_get_thread_num();
}
#pragma omp parallel
{
   printf(“a=%d\n”,a);
}
```

04 Data Environment

```
#include <stdio.h>
#include <omp.h>
int global_var = 20;
#pragma omp threadprivate(global_var)

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        global_var = global_var + thread_id + 1;
        printf("In parallel 1 (Thread %d): global_var = %d\n", thread_id, 
global_var);
    }
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        printf("before: In parallel 2 (Thread %d): global_var = %d\n", 
thread_id, global_var);
        global_var = global_var + thread_id + 1;
        printf("after: In parallel 2 (Thread %d): global_var = %d\n", 
thread_id, global_var);
    }
    // 并行区域结束后显示变量的值
    printf("After parallel: global_var = %d\n", global_var);
    return 0;
}
```

04 Data Environment

```c
#include <stdio.h>
#include <omp.h>
int global_var = 20;
#pragma omp threadprivate(global_var)

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num
        global_var = global_var + thread_i
        printf("In parallel 1 (Thread %d):
global_var);
    }
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num
        printf("before: In parallel 2 (Thr
thread_id, global_var);
        global_var = global_var + thread_i
        printf("after: In parallel 2 (Thre
thread_id, global_var);
    }
    // 并行区域结束后显示变量的值
    printf("After parallel: global_var = %d\n", global_var);
    return 0;
}
```

Execution output:

```
In parallel 1 (Thread 0): global_var = 21
In parallel 1 (Thread 2): global_var = 23
In parallel 1 (Thread 1): global_var = 22
In parallel 1 (Thread 3): global_var = 24
before: In parallel 2 (Thread 0): global_var = 21
before: In parallel 2 (Thread 2): global_var = 23
after: In parallel 2 (Thread 2): global_var = 26
after: In parallel 2 (Thread 0): global_var = 22
before: In parallel 2 (Thread 1): global_var = 22
after: In parallel 2 (Thread 1): global_var = 24
before: In parallel 2 (Thread 3): global_var = 24
after: In parallel 2 (Thread 3): global_var = 28
After parallel: global_var = 22
```

04 Data Environment

- ***copyin*** **Clause**
  - The copyin clause copies the value of a threadprivate variable from the master thread to the corresponding variables in each thread within the parallel region.
  - Arguments in the copyin clause must be declared as threadprivate.
  - Syntax: copyin(list)

04 Data Environment

```c
#include <stdio.h>
#include <omp.h>
int global_var = 20;
#pragma omp threadprivate(global_var)

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        global_var = global_var + thread_id + 1;
        printf("In parallel 1 (Thread %d): global_var = %d\n", thread_id,
global_var);
    }
    #pragma omp parallel num_threads(4) copyin(global_var)

    {
        int thread_id = omp_get_thread_num();
        printf("before: In parallel 2 (Thread %d): global_var = %d\n",
thread_id, global_var);
        global_var = global_var + thread_id + 1;
        printf("after: In parallel 2 (Thread %d): global_var = %d\n",
thread_id, global_var);
    }
    // 并行区域结束后显示变量的值
    printf("After parallel: global_var = %d\n", global_var);
    return 0;
}
```

04 Data Environment

```c
#include <stdio.h>
#include <omp.h>
int global_var = 20;
#pragma omp threadprivate(global_var)

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num
        global_var = global_var + thread_i
        printf("In parallel 1 (Thread %d):
global_var);
    }
    #pragma omp parallel num_threads(4) co

    {
        int thread_id = omp_get_thread_num
        printf("before: In parallel 2 (Thr
thread_id, global_var);
        global_var = global_var + thread_i
        printf("after: In parallel 2 (Thread %d): global_var = %d\n",
thread_id, global_var);
    }
    // 并行区域结束后显示变量的值
    printf("After parallel: global_var = %d\n", global_var);
    return 0;
}
```

Execution output:

```
In parallel 1 (Thread 0): global_var = 21
In parallel 1 (Thread 1): global_var = 22
In parallel 1 (Thread 3): global_var = 24
In parallel 1 (Thread 2): global_var = 23
before: In parallel 2 (Thread 3): global_var = 21
after: In parallel 2 (Thread 3): global_var = 25
before: In parallel 2 (Thread 1): global_var = 21
after: In parallel 2 (Thread 1): global_var = 23
before: In parallel 2 (Thread 2): global_var = 21
after: In parallel 2 (Thread 2): global_var = 24
before: In parallel 2 (Thread 0): global_var = 21
after: In parallel 2 (Thread 0): global_var = 22
After parallel: global_var = 22
```

04 Data Environment

- *private* **clause**
  - The private clause specifies that the variables in its list are local to each thread within the parallel region.
  - Syntax: ***private(list)***
  - private variables are "undefined" upon entering and exiting the parallel region. This means there is no association between the private variable inside the parallel region and the variable with the same name outside the region.

```
int main(int argc, _TCHAR* argv[]){
  int A=100,B,C=0;
  #pragma omp parallel for private(A,B)
  for(int i = 0; i<10;i++){
    B = A + i; // "A" is uninitialized;
Error.!
    printf("%d\n",i);
  }

  C = B; // "B" is uninitialized; Error.!
  printf("A:%d\n", A);
  printf("B:%d\n", B);
  return 0;
}
```

04 Data Environment

```
#include <stdio.h>
#include <omp.h>
int main() {
    int var = 1;
    #pragma omp parallel private(var) num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        var = thread_id + 1;
        printf("In parallel 1 (Thread %d): var = %d\n", thread_id, var);
    }
    #pragma omp parallel private(var) num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        printf("Before: In parallel 2 (Thread %d): var = %d\n", thread_id, var);
        var = var + thread_id + 1;
        printf("After: In parallel 2 (Thread %d): var = %d\n", thread_id, var);
    }
    // 并行区域结束后显示变量的值
    printf("After parallel: var = %d\n", var);
    return 0;
}
```

04 Data Environment

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int var = 1;
    #pragma omp parallel private(var
    {
        int thread_id = omp_get_thre
        var = thread_id + 1;
        printf("In parallel 1 (Threa
    }
    #pragma omp parallel private(var
    {
        int thread_id = omp_get_thre
        printf("Before: In parallel
        var = var + thread_id + 1;
        printf("After: In parallel 2
    }
    // 并行区域结束后显示变量的值
    printf("After parallel: var = %d\n", var);
    return 0;
}
```

Execution output:

```
In parallel 1 (Thread 3): var = 4
In parallel 1 (Thread 0): var = 1
In parallel 1 (Thread 2): var = 3
In parallel 1 (Thread 1): var = 2
Before: In parallel 2 (Thread 3): var = 0
After: In parallel 2 (Thread 3): var = 4
Before: In parallel 2 (Thread 1): var = 0
After: In parallel 2 (Thread 1): var = 2
Before: In parallel 2 (Thread 2): var = 0
After: In parallel 2 (Thread 2): var = 3
Before: In parallel 2 (Thread 0): var = 32661
After: In parallel 2 (Thread 0): var = 32662
After parallel: var = 1
```

04 Data Environment

- ***firstprivate*** **Clause**
  - firstprivate variables are initialized upon entering the parallel region using the value of the variable from outside the region.
  - Syntax: ***firstprivate(list)***
- ***lastprivate*** **Clause**
  - If a private variable inside the parallel region needs to pass its calculated value back to the variable with the same name outside the region upon exit, lastprivate can be used.
  - Syntax: ***lastprivate(list)***
  - The value from the last logical iteration of a loop is assigned to the variable outside the parallel region.

```
int A = 100;
#pragma omp parallel for lastprivate(A)
for(int i = 0; i<10;i++){
   A = 10 + i;
}
printf("%d\n",A);
```

04 Data Environment

- ***shared*** **Clause**
  - The shared clause declares that variables are shared among multiple threads, meaning the variables inside and outside the parallel region point to the same memory location. Consequently, one must be cautious of data races.
  - Syntax: ***shared(list)***

```
int sum = 0;
#pragma omp parallel for shared(sum)
for(int i = 0; i < COUNT; i++){
   sum = sum + i;
}
printf("%d\n",sum);
```

- ***default*** **Clause**
  - Specifies the default data-sharing attributes for variables within a parallel region.
  - Syntax: ***default(shared | none)***
  - default(shared): Variables within the parallel region default to shared if not explicitly declared as private.
  - default(none): Forces the explicit declaration of data-sharing attributes for all variables; otherwise, a compilation error will occur.

04 Data Environment

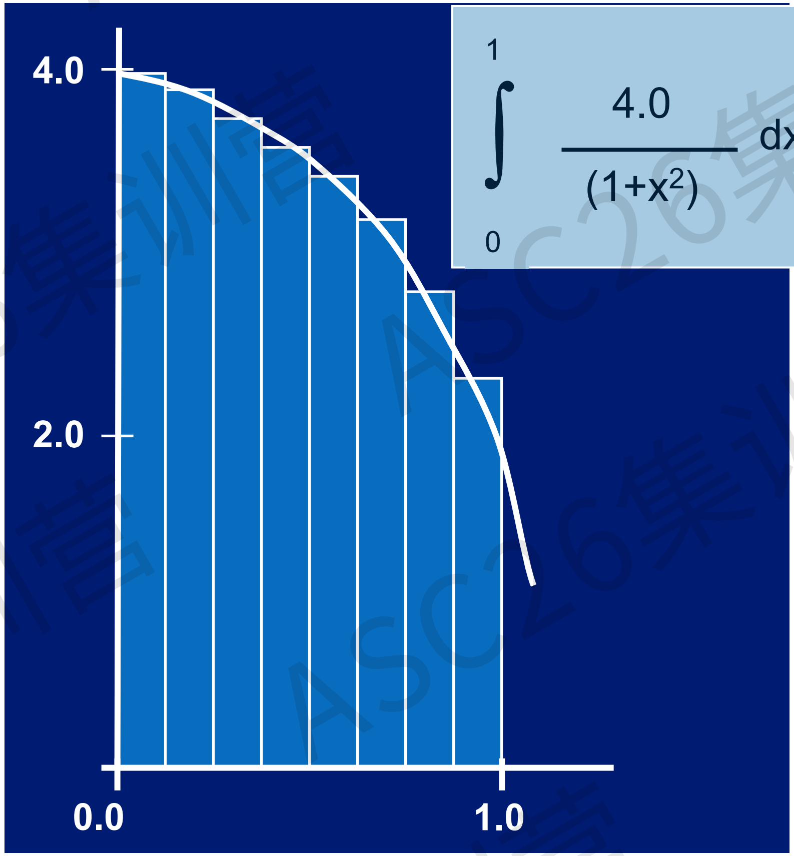
■ *reduction* Clause

- specifies an operator for a variable. Each thread creates a private copy of the reduction variable. At the end of the parallel region, the values of these private copies are combined according to the specified operator, and the final result is assigned to the original variable.
- Common Reduction Operators and Initial Values:+(0),-(0),*(1),^(0),&(~0),|(0),&&(1),||(0)
- Syntax: *reduction(operator:list)*

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < COUNT; i++){
   sum = sum + i;
}
printf("%d\n",sum);
```

04 Data Environment

■ **Numerical Integration Method to Calculate Pi**



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
#pragma omp parallel for \
   private(i, x) reduction(+:sum)
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf(“Pi = %f\n”,pi);
}
```

Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

05 Synchronization

- **OpenMP Directives for Thread Synchronization**
  - single
  - master
  - critical
  - barrier
  - atomic
  - ordered

05 Synchronization

master Directive

- Specifies that a structured block is executed only by the master thread; other threads skip the block and continue execution. It is commonly used for I/O operations.
- There is no implicit barrier at the end of the block.
- Syntax: #pragma omp master [clauses]

```
#pragma omp parallel {
   DoManyThings();
#pragma omp master {
//if not master skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

```
#pragma omp parallel {
   DoManyThings();
#pragma omp single {
     ExchangeBoundaries();
   }
// threads wait here for single
   DoManyMoreThings();
```

05 Synchronization

- *critical* **Directive**
  - Specifies that a structured block can only be executed by one thread at a time. Other threads are blocked outside the critical section. It is used to protect modifications to shared variables and avoid data races.
  - If the name is omitted, the name is assumed to be null.
  - When using named critical sections, an application can have multiple distinct critical sections. Generally, all critical sections should be named.
  - Syntax: ***#pragma omp critical [(name)]***

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
    #pragma omp critical (RES_lock)
    consum (B, RES);
  }
}
```

05 Synchronization

barrier Directive

- The barrier directive is used to synchronize all threads within a thread team.
- Threads that arrive first are blocked here, waiting for the remaining threads to catch up.
- An implicit barrier exists at the end of *parallel*, *for*, *sections*, and *single* structured blocks.
- The implicit barrier at the end of *for*, *sections*, and *single* blocks can be removed using the *nowait* clause.
- Deadlock Warning: Either all threads must encounter the barrier, or no threads should encounter it; otherwise, a deadlock will occur.
- Syntax: ***#pragma omp barrier***

```
#pragma omp parallel shared (A, B, C)
{
            DoSomeWork(A,B);
            printf("Processed A into B\n");
#pragma omp barrier
            DoSomeWork(B,C);
            printf("Processed B into C\n");
}
```

05 Synchronization

atomic Directive

- Specifies that a specific memory location will be updated atomically.
- Syntax:

  #pragma omp atomic

  statement
- Supported statement formats for *atomic*:

x binop = expr
x++
++x
x--
--x

x is a scalar
expr is a scalar expression that does not reference *x* and is not overloaded
binop is one of *+, *, -, /, &, ^, |, >>, or <<,* and is not overloaded.

05 Synchronization

atomic Directive

- The *atomic* directive allows parallel updates to different array elements, whereas the *critical* directive serializes all updates to the array.
- To avoid data races during shared memory updates, *atomic* should be prioritized over *critical* sections whenever possible.

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```

05 Synchronization

- *ordered* **Directive**
  - Specifies that the structured block within a parallel loop must be executed in the sequential order of the loop iterations. Only one thread can execute the ordered section at any given time, following the logical order of the loop index.
  - It can only appear within the dynamic extent of a parallel for construct.
  - Syntax: *#pragma omp ordered*

```
vector<int> v;
#pragma omp parallel for ordered
for (int i = 0; i < n; ++i){
      ... ... ...
   #pragma omp ordered
   v.push_back(i);
}
```

Outline for OpenMP

01 OpenMP Introduction

02 Parallel Region

03 Work Sharing

04 Data Environment

05 Synchronization

06 Library Routines and Environment Variables

06 Library Routines and Environment Variables

- **OpenMP Runtime Library Routines**
  - Set/Get the number of threads or thread ID.
    - omp_get_num_threads( ) / omp_set_num_threads( )
    - omp_get_thread_num( )
    - omp_get_max_threads( )
  - Determine if the code is currently executing within a parallel region.
    - omp_in_parallel( )
  - Obtain the number of processor cores available in the system.
    - omp_get_num_procs( )
  - Program timing (Wall-clock time).
    - omp_get_wtime( )

```
double start = omp_get_wtime();
#pragma omp parallel
{
    ... //work to be timed
}
double end = omp_get_wtime();
//Get wall-clock time in seconds.
double time = end – start;
```

06 Library Routines and Environment Variables

- **Environment Variables**
  - **OMP_NUM_THREADS**: Sets the maximum number of threads to use.
    - setenv OMP_NUM_THREADS 4 // Windows (CMD/PowerShell style)
    - export OMP_NUM_THREADS=4 // Linux/Unix (Bash style)
  - **OMP_SCHEDULE**: Sets the scheduling type for DO (Fortran) or for (C/C++) loops.
    - setenv OMP_SCHEDULE "DYNAMIC, 4"
  - **OMP_DYNAMIC**: Determines whether the number of threads used for a parallel region can be adjusted dynamically. The value is either TRUE or FALSE (Default: TRUE).
  - **OMP_NESTED**: Determines whether nested parallelism is enabled (Default: FALSE).

Outline for MPI

01 MPI Introduction

02 MPI Basic Functions

03 Point-to-Point Communication

04 Collective Communication

Outline for MPI

01 MPI Introduction

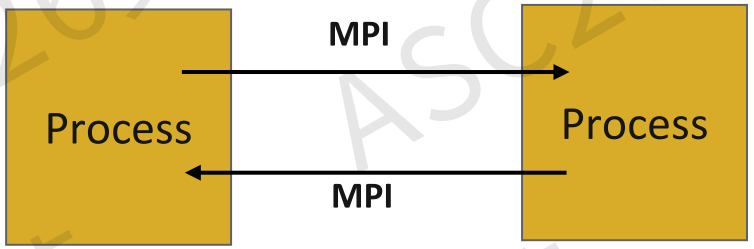
02 MPI Basic Functions

03 Point-to-Point Communication

04 Collective Communication

01 MPI Introduction

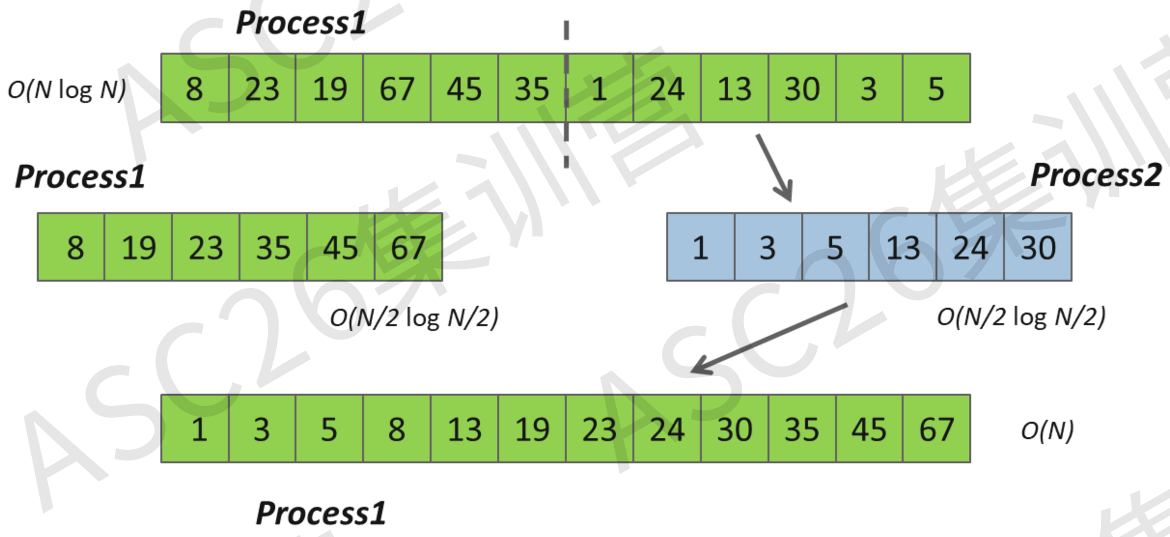
- **Message-Passing Programming Model**
- Processes have independent address spaces; processes communicate via MPI (Message Passing Interface).
- Inter-process communication includes：
  - Synchronization
  - Data communication (data is transferred from one process's address space to another process's address space)



01 MPI Introduction

Message-Passing Programming Model

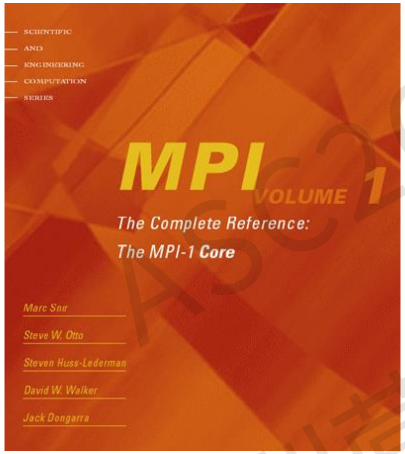
- Communication between processes is achieved by sending and receiving messages.
- A message is an encapsulation of data.
- Example: parallel sorting

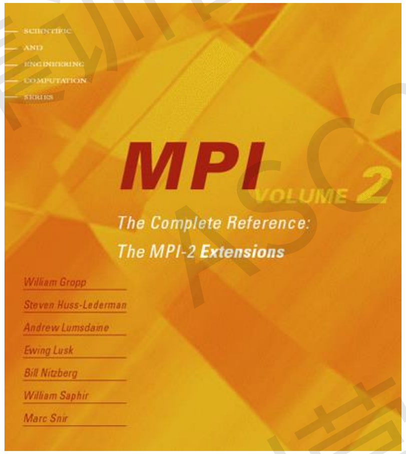


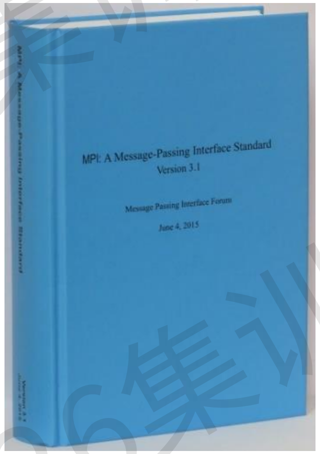
01 MPI Introduction

■ What is MPI?

- ➢ It is a functional library specification, not a programming language; operations are invoked like library calls.
- ➢ It is a standard and specification, not a specific implementation, and is language-independent.
- ➢ It is a message-passing programming model, and has become the representative and de facto standard for this model.






MPI: A Message-Passing Interface Standard
Version 3.1
Message Passing Interface Forum
June 4, 2015

01 MPI Introduction

- **History of MPI（www.mpi-forum.org）**
  - MPI-1: 1994
    - Supports classical message-passing programming (point-to-point communication, collective communication, etc.)
    - MPICH: the most popular open-source MPI implementation, jointly developed by Argonne National Laboratory and Mississippi State University
  - MPI-2: 1997
    - Dynamic process management, parallel I/O, remote memory access, support for F90 and C++
  - MPI-3: 2012
  - MPI-4: 2021
  - MPI-5: 2025

01 MPI Introduction

Why use MPI?

- **Standardization**: MPI has become the standard for parallel programming on distributed-memory systems
- **Portability**: applications can be ported across MPI-supported platforms without modifying source code.
- **Performance optimization**: vendors' MPI implementations can exploit hardware characteristics for optimization.
- **Functionality**: the MPI standard defines rich functionality.
- **Availability**: there are multiple open-source and commercial implementations.
  - Major open-source implementations include MPICH and Open MPI.
  - Industry implementations based on MPICH include:
    - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH, MPICH-MX

Outline for MPI

01 MPI Introduction

02 MPI Basic Functions

03 Point-to-Point Communication

04 Collective Communication

02 MPI Basic Functions

- "Hello World" Example

Hello world(C)

```
#include <stdio.h>
#include "mpi.h"

main( int argc, char *argv[] )
{
   MPI_Init( &argc, &argv );
   printf( "Hello, world!\n" );
   MPI_Finalize();
}
```

Hello world(Fortran)

```
program main
include 'mpif.h'
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

02 MPI Basic Functions

- **MPI Function Conventions in C and Fortran**
  - C
    - Must include mpi.h
    - MPI functions return an error code or the success flag MPI_SUCCESS
    - Prefix MPI_; only the first letter after MPI or MPI_ is capitalized, the rest are lowercase
  - Fortran
    - Must include mpif.h
    - Call MPI as subroutines; the last argument is the return code
    - Prefix MPI_; function names are all uppercase
  - MPI function parameters are marked as:
    - IN： input parameter; not modified by the routine
    - OUT： output parameter; may be modified by the routine
    - INOUT： used for both directions of data transfer

02 MPI Basic Functions

- **MPI Initialization：** *int MPI_Init(int *argc, char **argv)*
  - ➢ This is the first call in an MPI program. It performs all MPI initialization. The first executable statement of any MPI program is this call.
  - ➢ Starts the MPI environment

- **MPI Finalization：** *int MPI_Finalize(void)*
  - ➢ This is the last call in an MPI program. It terminates the MPI program and must be the last executable statement; otherwise, the behavior is unpredictable.

02 MPI Basic Functions

Compiling and Running the "Hello World" Example

- MPI exists as a function library. Compiling an MPI program calls the native compiler plus related settings; these details are encapsulated by the executable wrapper script ***mpicc***.
- **Compile**:
  - Normal program: ***gcc hello.c -o hello***
  - MPI program： ***mpicc hello.c -o hello***
- **Run**:
  - Normal program： ***./hello***
  - MPI program： ***mpiexec –n 16 ./hello***

Specify the number of processes

mpicc -show

Output of *mpicc –show*:

```
gcc -I/usr/lib/x86_64-
linux-gnu/openmpi/include
-I/usr/lib/x86_64-linux-
gnu/openmpi/include/openmp
i -L/usr/lib/x86_64-linux-
gnu/openmpi/lib -lmpi
```

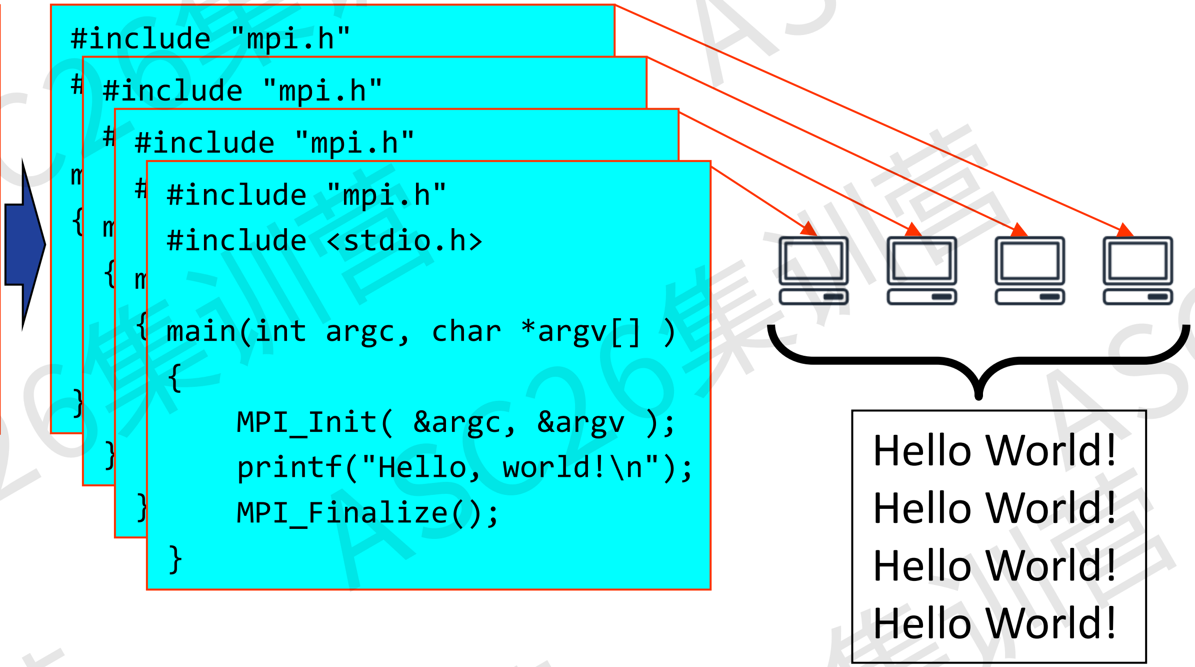
02 MPI Basic Functions

■ How is “Hello” executed?

➢ **SPMD**: Single Program Multiple Data(SPMD)

```
#include "mpi.h"
#include <stdio.h>

main(int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf("Hello, world!\n");
    MPI_Finalize();
}
```



```
#include "mpi.h"
#include <stdio.h>

main(int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf("Hello, world!\n");
    MPI_Finalize();
}
```

```
Hello World!
Hello World!
Hello World!
Hello World!
```

02 MPI Basic Functions

- **Improving the “Hello World” Example**
  - How many processes are used for parallel computation?
  - Which process am I?

```
#include <stdio.h>
#include "mpi.h"

main( int argc, char *argv[] )
{
   int  myid, numprocs;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myid );
   MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
   printf(“I am %d of %d\n", myid, numprocs );
   MPI_Finalize();
}
```

02 MPI Basic Functions

- **Process group**
  - A finite, ordered subset of MPI processes.
  - Each process in the group is assigned a unique rank within the group, used to identify the process.
  - Rank ranges from *[0, number_of_processes -1]*
- **Communicator**
  - A communicator includes a process group and a communication context, and describes communication relationships among processes.
  - Communicators are divided into intra-communicators and inter-communicators, used for intra-group and inter-group communication
  - most MPI users only need intra-communicators.

02 MPI Basic Functions

- **Communicator**
  - The communication context acts like a “super tag” to safely distinguish different communications and avoid interference.
  - MPI includes several predefined communicators.
    - MPI_COMM_WORLD is the set of all MPI processes and is created automatically after MPI_Init.
    - MPI_COMM_SELF is a communicator containing only the calling process.
  - Any MPI communication function must occur within a communicator.

02 MPI Basic Functions

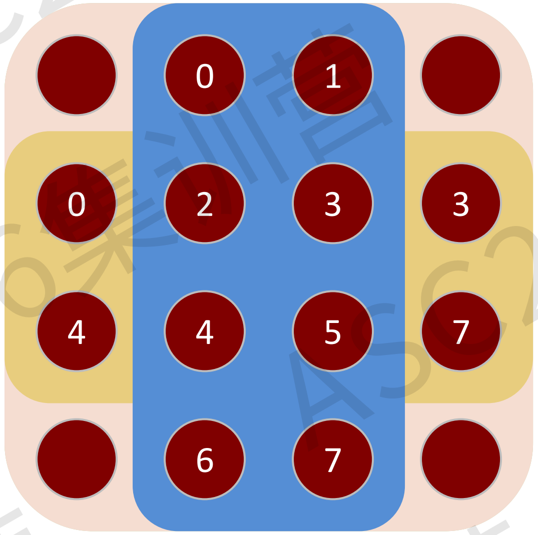
■ **Communicator**

- ➢ Communicators can be created manually in programs.
- ➢ Simple applications typically only need the default communicator MPI_COMM_WORLD.

```
mpiexec -n 16 ./test
```

A communicator does not need to include all processes in the system.

Each process in a communicator has a rank.



When starting an MPI program, the predefined communicator MPI_COMM_WORLD is created automatically.

A communicator can be duplicated.

The same process may have different ranks in different communicators.

02 MPI Basic Functions

- Get the number of processes in a given communicator:

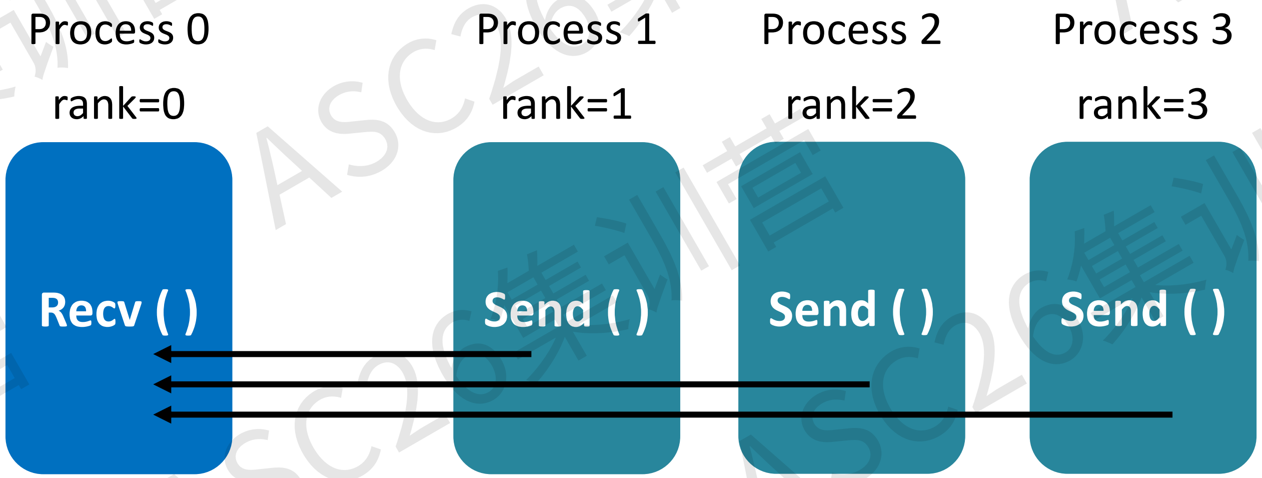
```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- Get the process rank:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

02 MPI Basic Functions

- Message-Passing “Greetings” Example



02 MPI Basic Functions

■ Message-Passing “Greetings” Example

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[])
{
  int numprocs, myid, source;
  MPI_Status status;
  char message[100];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,
            &myid);
  MPI_Comm_size(MPI_COMM_WORLD,
            &numprocs);
```

```
  if (myid != 0) {
    strcpy(message, "Hello World!");
    MPI_Send(message,strlen(message)+1,
        MPI_CHAR, 0,99, MPI_COMM_WORLD);
  }
  else {/* myid == 0 */
    for(source=1; source<numprocs; source++){
      MPI_Recv(message, 100, MPI_CHAR, source,
            99, MPI_COMM_WORLD,  &status);
      printf("%s\n", message);
    }
  }
  MPI_Finalize();
} /* end main */
```

02 MPI Basic Functions

■ **Sending a message**

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,*

int tag, MPI_Comm comm);

- ➢ One process sends data to another process (or a group of processes).
- ➢ Sending a message requires：
  - ● What data to send?
    - – Buf is the message address， count is the number of elements， datatype is the element type.
  - ● Who to send to?
    - – The process with rank dest in communicator comm
  - ● User-defined message tag （tag）

int → MPI_INT
double → MPI_DOUBLE
char → MPI_CHAR

02 MPI Basic Functions

■ Receiving a message

*int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,*

*MPI_Comm comm, MPI_Status *status);*

➢ The receiver needs：

- The data type(datatype)、size(count)and destination buffer(buf)
- Receive from which sender? （the process with rank source in communicator comm）
- User-defined message tag（tag）
- status is the returned status containing additional information such as the actual number of elements received

02 MPI Basic Functions

Message status

- The message status (MPI_Status type) stores status information about a received message, including:
  - MPI_SOURCE: rank of the sending process
  - MPI_TAG: tag of the received message
  - MPI_ERROR: error status

```
typedef struct _MPI_Status {
  int count;
  int cancelled;
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
} MPI_Status, *PMPI_Status;
```

- It is the last parameter of MPI_Recv
- If no information is needed, use MPI_STATUS_IGNORE
- The actual number of received elements can be obtained via:

*MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)*

02 MPI Basic Functions

- The 6 most basic MPI functions:
  - MPI_Init(…);
  - MPI_Comm_size(…);
  - MPI_Comm_rank(…);
  - MPI_Send(…);
  - MPI_Recv(…);
  - MPI_Finalize();

Outline for MPI

01 MPI Introduction

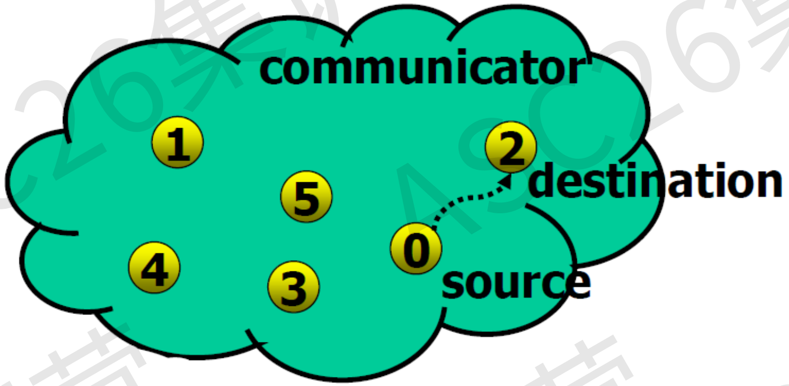
02 MPI Basic Functions

03 Point-to-Point Communication

04 Collective Communication

03 Point-to-Point Communication

- **What is point-to-point communication?**
  - Communication between two MPI processes
  - The source process sends a message to the destination process
  - The destination process receives the message
  - Communication occurs within the same communicator



03 Point-to-Point Communication

Terminology of MPI Communication Functions

- **Blocking:** the send/receive call waits for completion before returning; after it returns, the resources used in the call can be reused
- **Non-blocking**: the call can return without waiting for completion, but that does not mean the resources can be reused immediately
- **Local**: completion of the call does not depend on other processes
- **Non-local**: completion depends on other processes; e.g., a sender may wait until the receiver has received before returning
- **Collective**: all processes in the group participate

03 Point-to-Point Communication

■ **Overview of MPI point-to-point functions**

- MPI provides both blocking and non-blocking mechanisms for point-to-point communication.
- It also supports four communication modes, referring to buffer management and synchronization between sender and receiver:
  - Synchronous mode
  - Buffered mode
  - Standard mode
  - Ready mode
- Combining different modes with blocking/non-blocking mechanisms yields a rich set of point-to-point functions.

03 Point-to-Point Communication

Overview of MPI point-to-point functions

- MPI send supports four modes; combined with blocking properties, this yields 8 kinds of send operations in MPI.
- MPI receive has only two kinds: blocking receive and non-blocking receive.
- Non-blocking calls returning does not mean the communication is complete; MPI provides completion tests mainly via MPI_Wait and MPI_Test.
- MPI_Sendrecv_replace means sending and receiving use the same buffer.

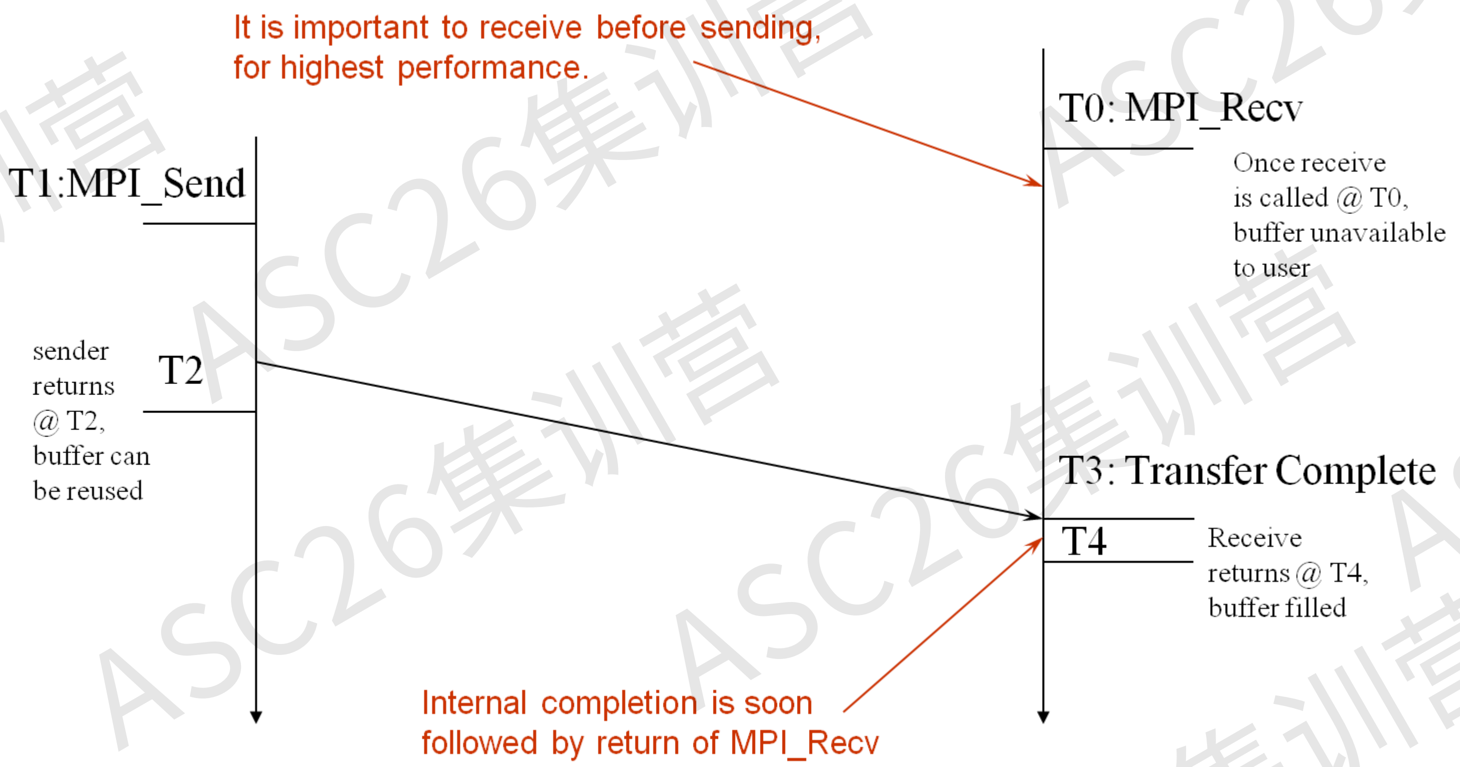
Communication Mode	Blocking	Non-blocking
Synchronous	MPI_Ssend	MPI_Issend
Buffered	MPI_Bsend	MPI_Ibsend
Ready	MPI_Rsend	MPI_Irsend
Standard	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

03 Point-to-Point Communication

- **Blocking vs. Non-blocking**
  - **Blocking communication functions**： *MPI_Send/MPI_Recv*
    - The process is blocked; when the call returns, the memory used in communication can be reused.
      - For send: the send buffer buf can be reused/modified; modifications do not affect the data sent to the receiver.
      - For receive: the message has been received into buf; data can be read from it.
    - The exact completion semantics depend on system buffering and message size.
    - Blocking communication is easy to use but can easily cause deadlocks.
  - **Non-blocking communication functions**： *MPI_Isend/MPI_Irecv*
    - The call returns immediately; you must separately test for completion.
    - Mainly used to overlap computation and communication to improve performance.

03 Point-to-Point Communication

■ **Blocking Communication**

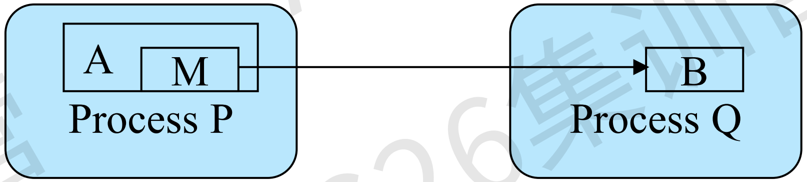

It is important to receive before sending, for highest performance.
T0: MPI_Recv
Once receive is called @ T0, buffer unavailable to user
T1:MPI_Send
sender returns @ T2, buffer can be reused
T2
T3: Transfer Complete
T4
Receive returns @ T4, buffer filled
Internal completion is soon followed by return of MPI_Recv

Sending process **Receiving process**

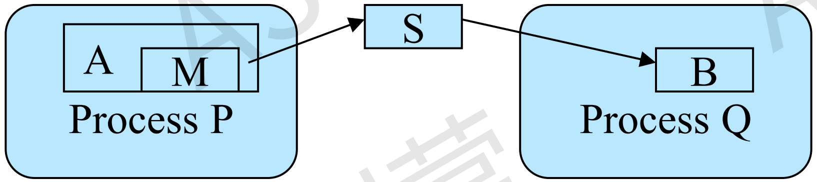
03 Point-to-Point Communication

What are buffers in MPI communication?

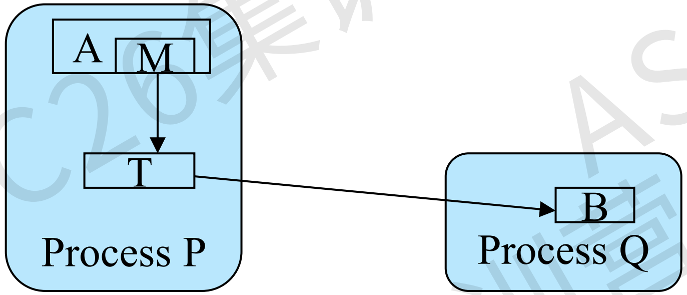
- Variables declared in the application, used as the starting address of buffers in message-passing statements
- A memory area created and managed by the system (varies by implementation/user), used to temporarily store messages during message passing; also called the system buffer
- A user-allocated memory area used as an intermediate buffer to hold arbitrary messages that may occur in the application



User buffer



System buffer



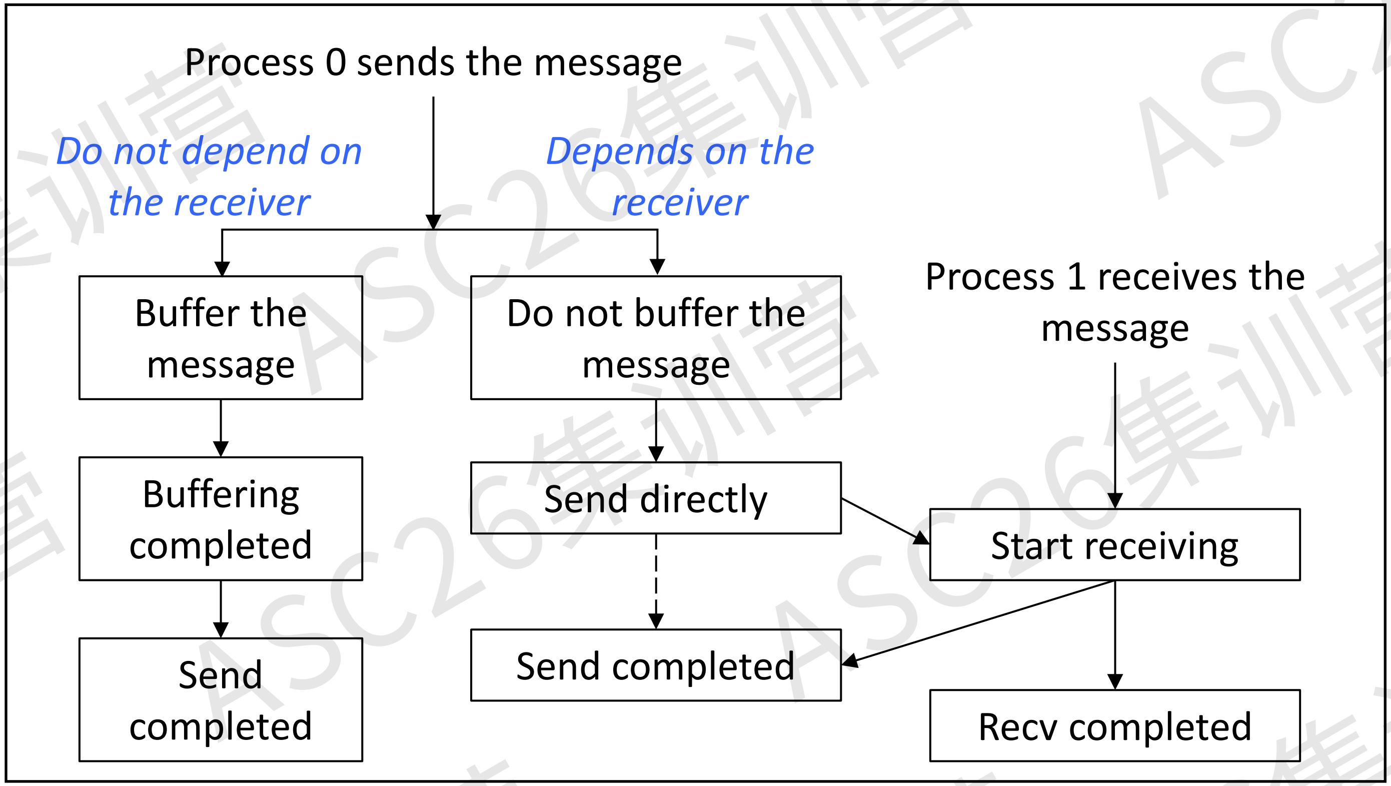
User-provided buffer

03 Point-to-Point Communication

- **Standard communication mode**
  - Whether send data is buffered is determined by the MPI implementation, not by the user program.
    - If data is buffered, the send returning correctly does not depend on the receiver.
    - If data is not buffered and is sent directly, the send returns correctly only after the matching receive is executed and data reception has begun.
  - Equivalent to “no fixed mode”; the concrete mode is chosen by the implementation.
  - OpenMPI uses buffered mode for short messages, and a synchronous-like mode for long messages.

03 Point-to-Point Communication

■ **Standard communication mode**

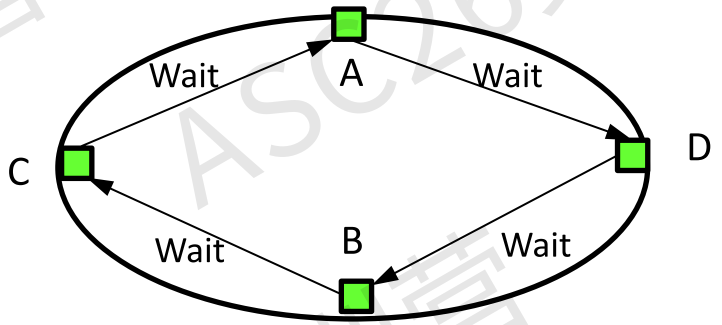


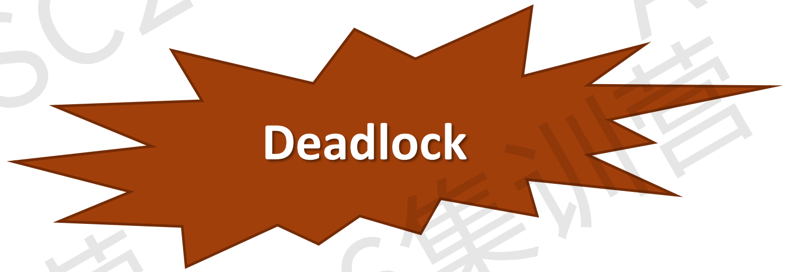
03 Point-to-Point Communication

■ **Writing safe MPI programs**

➢ MPI programs can easily deadlock if communication calls are ordered improperly.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)     A
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)             C
ELSE IF( rank .EQ. 1)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)     B
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)             D
END IF
```

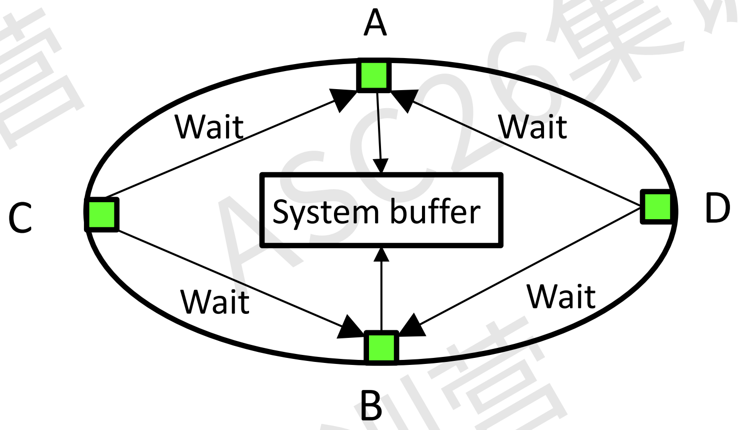




03 Point-to-Point Communication

- Writing safe MPI programs

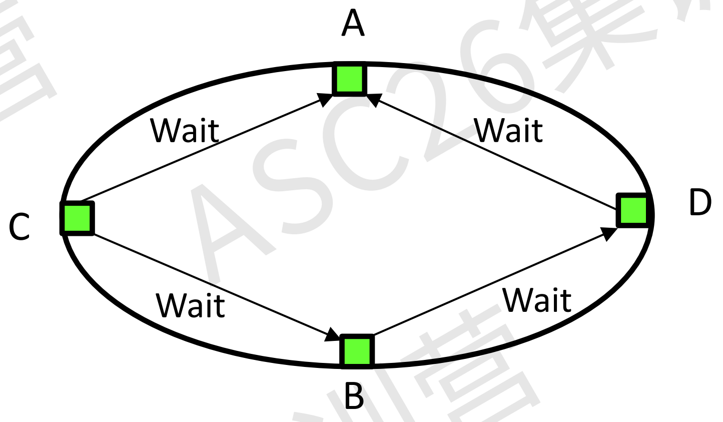
```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)            A
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)    C
ELSE IF( rank .EQ. 1)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)            B
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)    D
END IF
```



03 Point-to-Point Communication

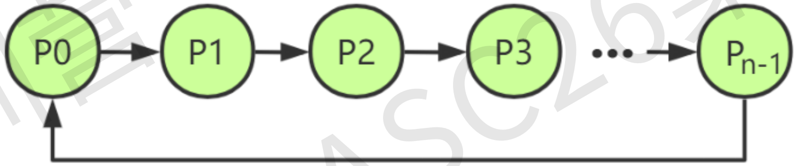
- **Writing safe MPI programs**

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)           A
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)   C
ELSE IF( rank .EQ. 1)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)   D
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)           B
END IF
```



03 Point-to-Point Communication

- **Bundling send and receive**
  - Data rotation



```
int MPI_Sendrecv(
     void *          sendbuf,         //starting address of send buffer
     int             sendcount,       //number of elements to send
     MPI_Datatype sendtype,           //datatype of sent data
     int             dest,            //destination process rank
     int             sendtag,         //send message tag
     void *          recvbuf,         //starting address of receive buffer
     int             recvcount,       //maximum number of elements to
     receive
     MPI_Datatype recvtype,           //datatype of received data
     int             source,          //source process rank
     int             recvtag,         //receive message tag
     MPI_Comm     comm,              //communicator
     MPI_Status *   status            //returned status
)
```

03 Point-to-Point Communication

■ **Bundling send and receive**

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF( rank .EQ. 1)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

```
RECVCALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
  CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, tag,
+                recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
IF(rank.EQ.1) THEN
  CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, tag,
+                recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
```

03 Point-to-Point Communication

■ Non-blocking communication

➢ Comparison with blocking communication

Type	Functions	Return Semantics	Buffer Safety / Access to Buffers	Characteristics
Blocking	MPI_Send MPI_Recv	• Blocking calls do not return until the specified operation completes. • Or, they return only after the MPI library has safely buffered (copied) the data involved.	After the call returns, it is safe to access/modify the involved buffers	• Program design is relatively simple. • Improper use can easily lead to deadlock.
Non-blocking	MPI_Isend MPI_Irecv	• Returns immediately; the actual communication proceeds in the background (handled by the MPI implementation). • You must use additional calls to wait for or test completion (e.g., wait/test routines).	After the call returns, it is unsafe to access/modify the buffers involved until completion	• Enables overlap of computation and communication. • Program design is relatively complex.

03 Point-to-Point Communication

- **Non-blocking send**

  int MPI_Isend(void buf, int count, MPI_Datatype datatype, int dest, int tag,*

  *MPI_Comm comm, MPI_Request *request)*

  - This function only posts a send request and returns immediately.
  - The MPI system completes message sending in the background.
  - The function creates a request object for this send and returns it via request.
  - request can be used later by query/wait functions.

- **Non-blocking receive**

  int MPI_Irecv(void buf, int count, MPI_Datatype datatype, int source, int tag,*

  MPI_Comm comm, MPI_Request request)*

03 Point-to-Point Communication

■ Using MPI_Wait

int MPI_Wait(MPI_Request request, MPI_Status * status);*

- Takes the non-blocking *request* object request as an argument, blocks until the corresponding non-blocking communication completes, stores related information in status, and frees the request object (request = MPI_REQUEST_NULL).

```
MPI_Request request;
MPI_Status status;
int x,y;
if(rank == 0){
    MPI_Isend(&x,1,MPI_INT,1,99,comm,&request)
    …
    MPI_Wait(&request,&status);
} else {
    MPI_Irecv(&y,1,MPI_INT,0,99,comm,&request)
    …
    MPI_Wait(&request,&status);
}
```

03 Point-to-Point Communication

■ **Using MPI_Test**

*int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);*

- MPI_Test returns immediately.
- If the corresponding non-blocking communication has completed, it sets the completion flag *flag = true*; otherwise, it sets *flag = false*.

```
MPI_Request request;
MPI_Status status;
int x,y,flag = 0;
if(rank == 0){
    MPI_Isend(&x,1,MPI_INT,1,99,comm,&request)
    while(!flag)
        MPI_Test(&request,&flag,&status);
} else {
    MPI_Irecv(&y,1,MPI_INT,0,99,comm,&request)
    while(!flag)
        MPI_Test(&request,&flag,&status);
}
```

Outline for MPI

01 MPI Introduction

02 MPI Basic Functions

03 Point-to-Point Communication

04 Collective Communication

04 Collective Communication

- **Collective communication is a global communication operation in which all processes in a group participate.**
- **Collective communication generally provides three functions: Data movement, data aggregation, and synchronization**
  - Data movement mainly transfers data within the group.
  - Data aggregation performs certain operations on the given data based on communication.
  - Synchronization ensures all processes in the group reach a consistent execution point.
- **By communication direction, collective communication can be divided into: One-to-many, many-to-one, and many-to-many communication**

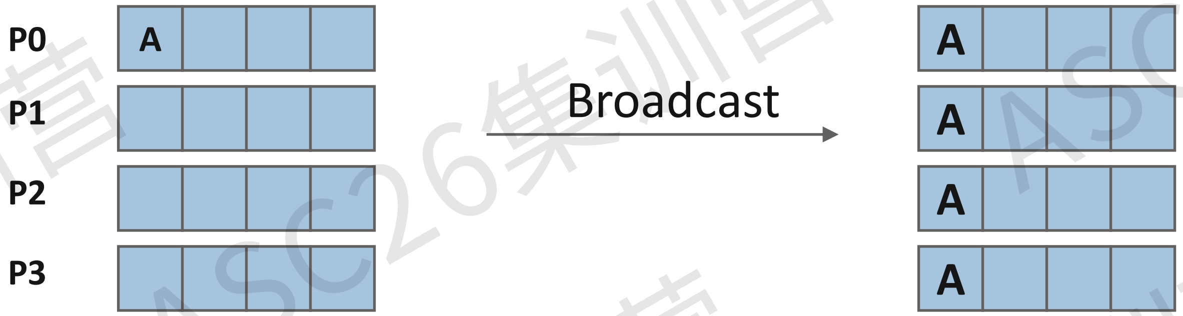
04 Collective Communication

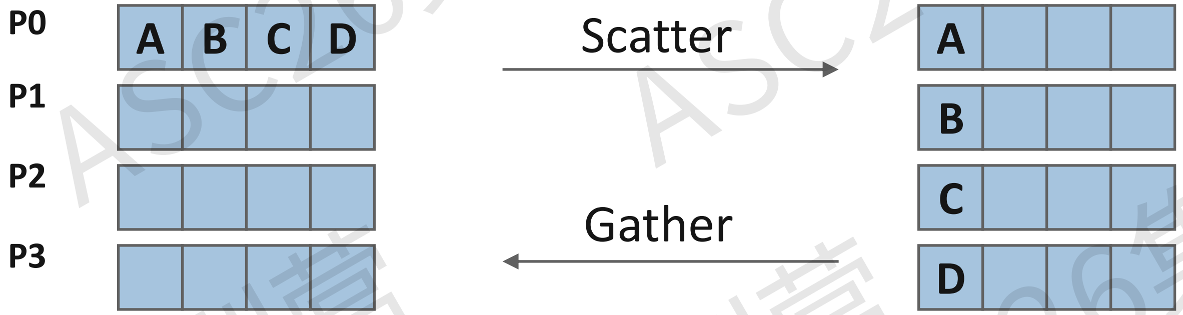
- **Collective communication functions**
- **All**: Deliver the result to all processes.
- **V**: Variety, with more flexible data objects and operations

Category	Function	Purpose
Data movement	MPI_Bcast	One-to-many, data broadcast
	MPI_Gather	Many-to-one, data gather
	MPI_Gatherv	Generalized form of MPI_Gather
	MPI_Allgather	All-process variant of MPI_Gather (gather result delivered to all processes)
	MPI_Allgatherv	Generalized form of MPI_Allgather
	MPI_Scatter	One-to-many, data scatter
	MPI_Scatterv	Generalized form of MPI_Scatter
	MPI_Alltoall	Many-to-many, data permutation (all-to-all exchange)
	MPI_Alltoallv	Generalized form of MPI_Alltoall
Data aggregation	MPI_Reduce	Many-to-one, data reduction
	MPI_Allreduce	All-process variant of the above; result available on all processes
	MPI_Reduce_scatter	Scatter the reduced result to all processes
	MPI_Scan	Prefix operation (scan)
Synchronization	MPI_Barrier	Synchronization operation (barrier)

04 Collective Communication

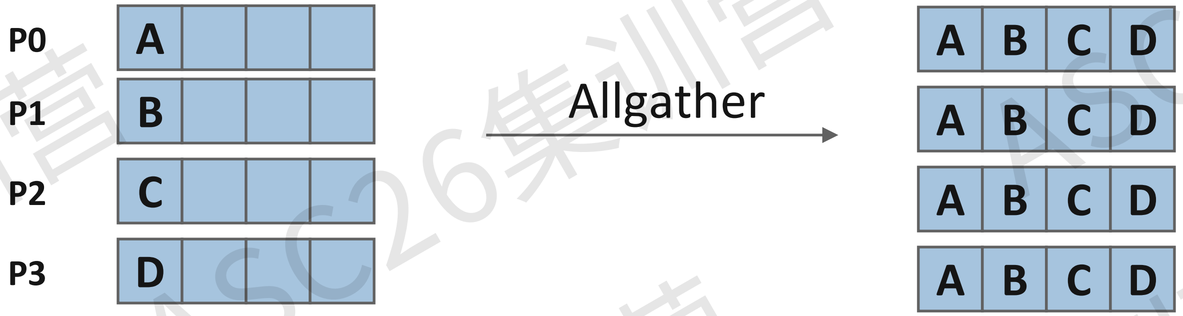
- **Data Movement**

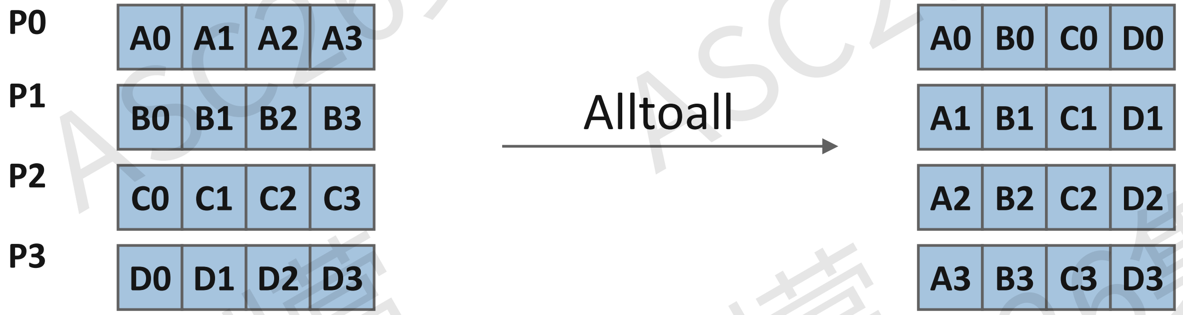




04 Collective Communication

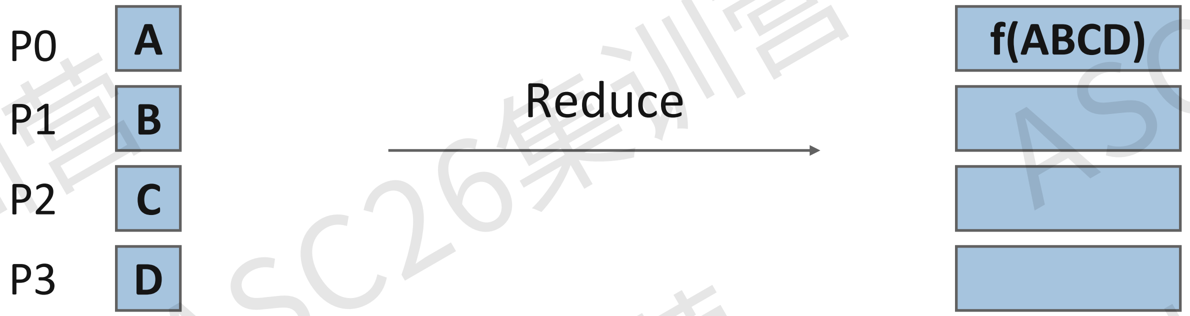
- **Data Movement**

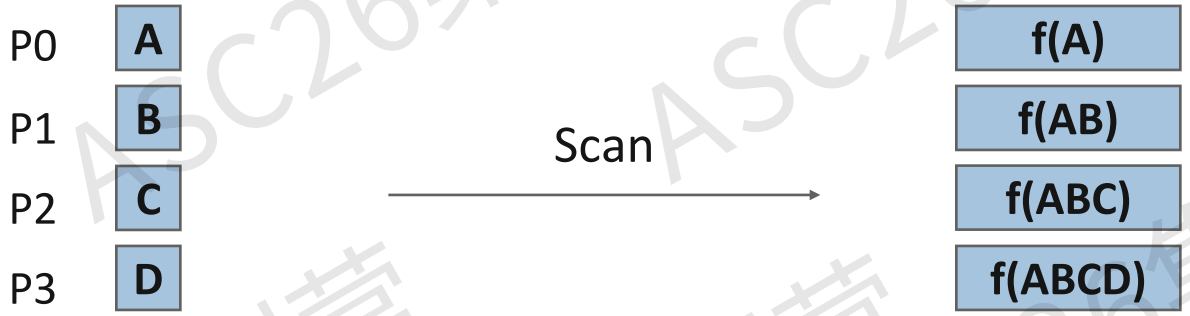




04 Collective Communication

- Data Aggregation



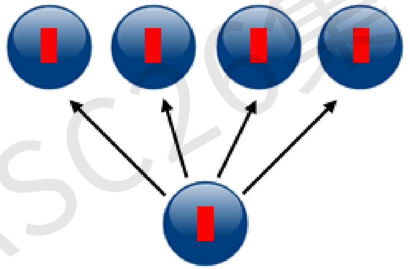


04 Collective Communication

- **Broadcast — data broadcast**

*int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype,*

int root, MPI_Comm comm);



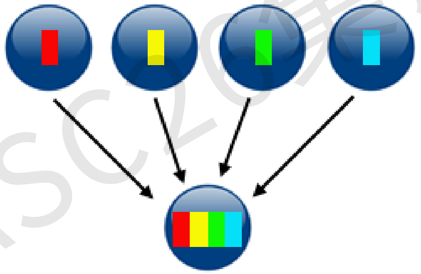
- The process designated as **root** sends the same message to **all processes** in the communicator **comm**.
- As in point-to-point communication, the message contents are specified by the triple **<buffer, count, datatype>**.
- For the **root** process, this triple defines both the **send buffer** and the **receive buffer**; for all other processes, it defines only the **receive buffer**.

```
int p, myrank;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm, &my_rank);
MPI_Comm_size(comm, &p);
if(myrank==0)
    buf = 1.0;
MPI_Bcast(&buf,1,MPI_FLOAT,0, comm);
```

04 Collective Communication

■ **Gather — data collection**

*int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);*



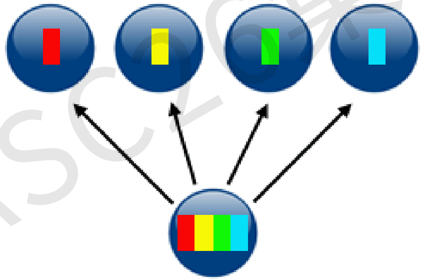
- The root process receives messages from all processes in communicator comm.
- Messages are concatenated in rank order and stored in the root’s receive buffer.
- recvcnt is the number of data elements the root receives from each process, not the total number of elements received by the root.

```
int p, myrank;
float data[10];
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm, &my_rank);
MPI_Comm_size(comm, &p);
if(myrank==0)
  buf = (float*)malloc(p*10*sizeof(float);
MPI_Gather(data,10,MPI_FLOAT,
       buf,10,MPI_FlOAT,0,comm);
```

04 Collective Communication

■ Scatter — data distribution

*int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);*



- ➢ **Scatter performs the reverse of Gather.**
- ➢ The root process sends a different message to all processes.
- ➢ Messages are stored in order of process rank in the root's send buffer.
- ➢ Each receive buffer is denoted by <recvbuf, recvcnt, recvtype>.
- ➢ For the root process, the send buffer is denoted by <sendbuf, sendcnt, sendtype>.

```
int p, myrank;
float data[10];
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm, &my_rank);
MPI_Comm_size(comm, &p);
if(myrank==0)
  buf = (float*)malloc(p*10*sizeof(float);
MPI_Scatter(buf,10,MPI_FLOAT,data,10,
          MPI_FlOAT,0,comm);
```

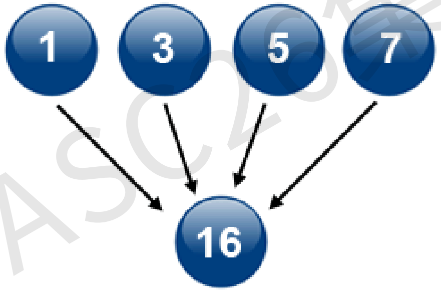
04 Collective Communication

- **Data Aggregation**
  - Collective data aggregation operations allow MPI processes to **perform certain computations while communicating**.
  - Data aggregation operations proceed in **three steps**:
    - First, communication: messages are sent to target processes as required, and target processes have received the needed messages.
    - Second, message processing: perform the computation.
    - Finally, place the processed result into the specified receive buffer
  - MPI provides two types of aggregation operations: **Reduce** and **Scan**.

04 Collective Communication

■ Reduce — data reduction

*int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);*



- Data in each process's send buffer (sendbuf) is combined with the specified operation, and the final result is stored in the root process's receive buffer (recvbuf).
- The data type of the items participating in the operation is defined by **datatype**, and the reduction operation is defined by **op**.
- The reduction operation can be predefined by MPI or user-defined
- Reduction allows each process to contribute a vector value, not just a scalar; vector length is defined by count.

```
int p, myrank;
float data = 0.0;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm,&my_rank);
data = data + myrank * 10;
MPI_Reduce(&data,&buf,1,MPI_FLOAT,MPI_
SUM,0,comm);
```

04 Collective Communication

■ **Reduce — data reduction**

➢ MPI predefined reduction operations

Operation	Meaning	Operation	Meaning
MPI_MAX	Maximum	MPI_LOR	Logical OR
MPI_MIN	Minimum	MPI_BOR	Bitwise OR
MPI_SUM	Sum	MPI_LXOR	Logical XOR
MPI_PROD	Product	MPI_BXOR	Bitwise XOR
MPI_LAND	Logical AND	MPI_MAXLOC	Maximum value and the corresponding location
MPI_BAND	Bitwise AND	MPI_MINLOC	Minimum value and the corresponding location

04 Collective Communication

■ Reduce — data reduction

- User-defined reduction operation

```
MPI_OP_CREATE(user_fn, commutes, &op);
MPI_OP_FREE(&op);
user_fn(invec, inoutvec, len, datatype);
```

- The user-defined reduction function **user_fn** performs:

```
for i from 0 to len-1
    inoutvec[i] = invec[i] op inoutvec[i];
```

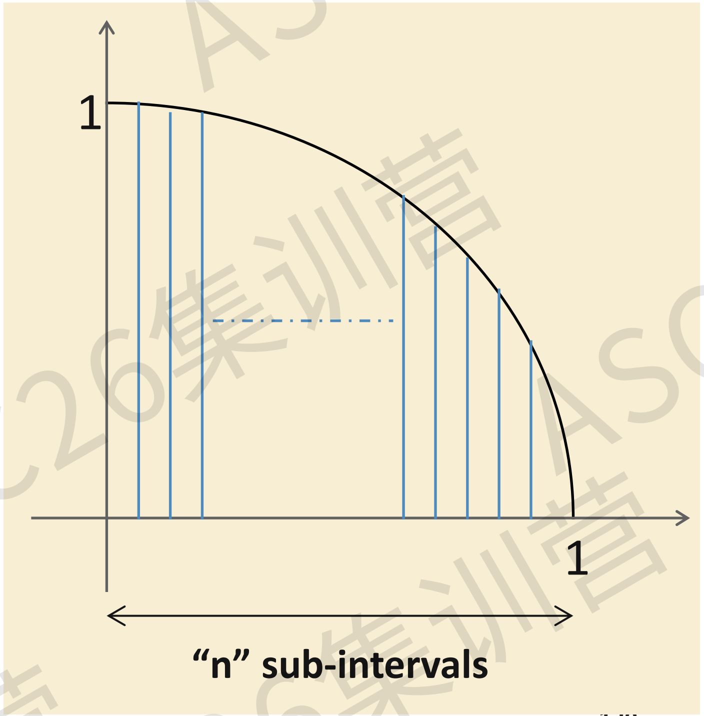
- A user-defined reduction operation does not have to be commutative, but it must be associative.

04 Collective Communication

Collective communication example: numerical integration to compute Pi

$$\pi = 4\int_0^1 \sqrt{1-x^2}\,dx \qquad \pi = 4\int_0^1 \frac{1}{1+x^2}\,dx$$

- Divide the interval into n sub-intervals
- Evenly distribute the n sub-intervals among p MPI processes
- Each process computes the sum of areas of n/p sub-intervals
- Sum the p partial sums to obtain *Pi*
- Width of each segment: $w = 1/n$
- x-coordinate of the start of segment i: $d(i) = i * w$
- Height of the small rectangle for segment i: *sqrt(1 – [d(i)]^2)*



04 Collective Communication

■ **Collective communication example: numerical integration to compute Pi**

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w    = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size)
        mypi += w * sqrt(1 - (((double) i / n) * ((double) i / n));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 * pi,
               fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

04 Collective Communication

■ **Barrier Synchronization**

int MPI_Barrier(MPI_Comm comm);

- ➢ Blocks the calling process until all processes in communicator comm have called this function.
- ➢ When MPI_Barrier returns, all processes are synchronized at the barrier.
- ➢ MPI_Barrier is implemented in software and may incur significant overhead on some machines.

04 Collective Communication

■ Characteristics of collective communication

- ➢ All processes in the communicator must call the collective function.
- ➢ Except for MPI_Barrier, each collective function uses a **standard, blocking communication mode** similar to point-to-point communication.
  - • Once a process finishes its participation in the collective operation, it returns from the collective call, but it does not guarantee that other processes have completed the collective call.
- ➢ Collective communication has no message tag parameter;
- ➢ The message envelope is defined by the communicator and source/destination.
  - • For example, in MPI_Bcast, the source is the root process, and the destinations are all processes (including the root).

ASC26 Student Supercomputer Challenge Training Camp

Thanks!

Jianhua Gao Beijing Normal University

2026/1/27