Parallel Programming: OpenMP & MPI

授課老師: 周志遠

NARLabs 財團法人國家實驗研究院
國家高速網路與計算中心
National Center for High-performance Computing
國立清華大學叢集電腦競賽團隊
Student Cluster Competition Team of NTHU

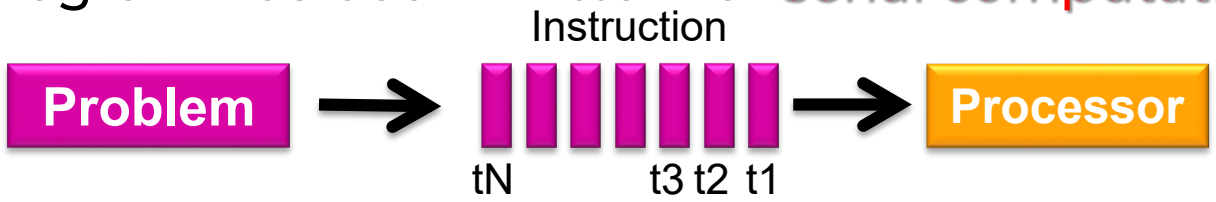
Parallel Computing Intro

- Parallel Computing Methods:
  - Embarrassingly Parallel
  - Divide-and-Conquer
  - Pipeline
- Parallel Computing Challenges:
  - Load-Balancing
  - Race condition
  - Data dependency
- SHARED-MEMORY PROGRAMMING: Pthread & OpenMP
- MESSAGE-PASSING PROGRAMMING: MPI

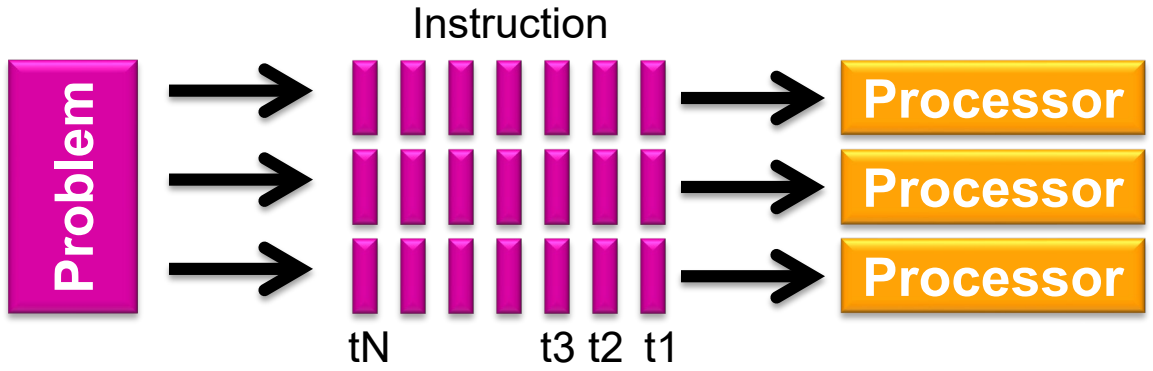
What is Parallel Computing?

"Solve a *single problem* by using *multiple processors* (i.e. *core*) working together"

- Traditionally, program has been written for serial computation

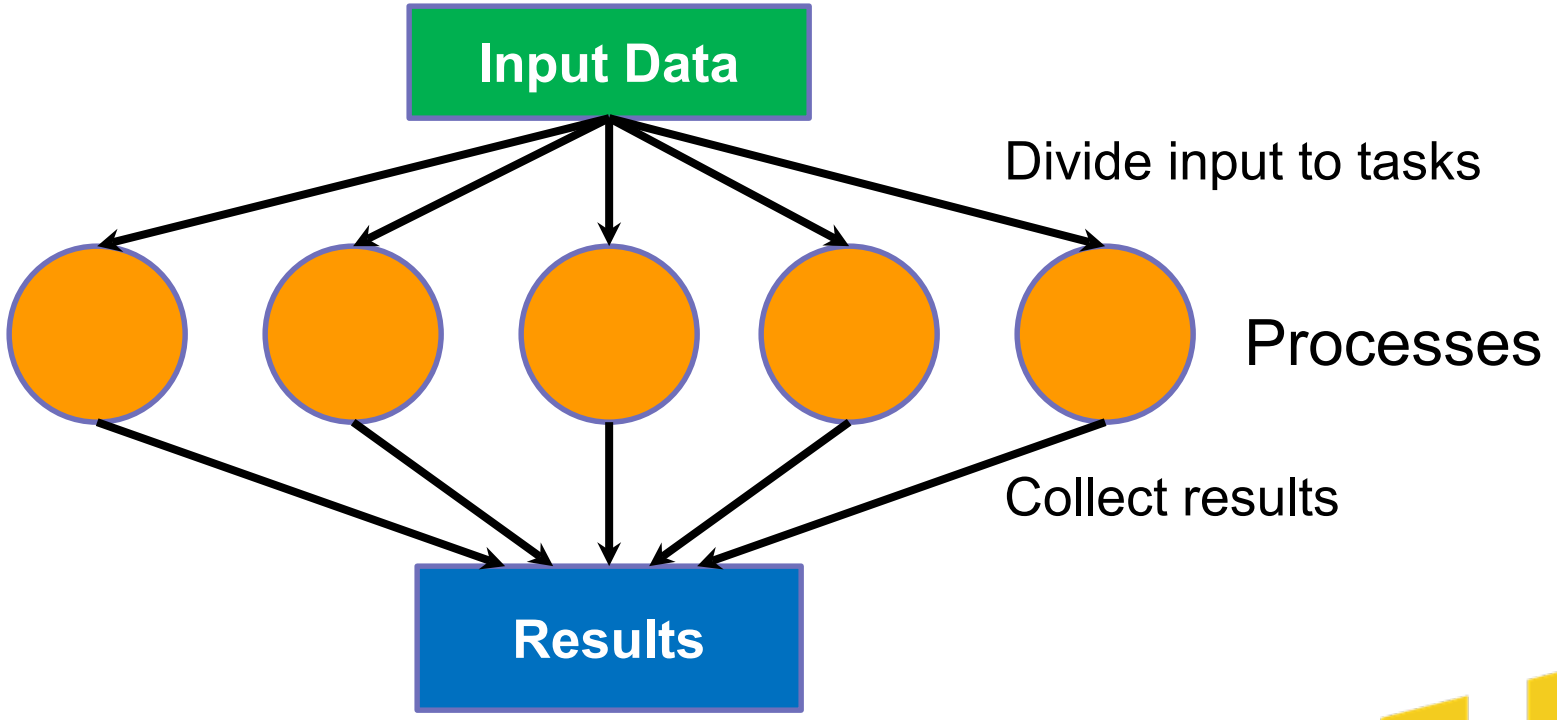


- In parallel computing, use multiple computer resources to solve a computational problem



Method 1: Embarrassingly Parallel

- A computation that can be divided into a number of completely independent tasks



Example: Image Transformations

- Low-level image operations:
  - Shifting: object shifted by Δx in the x-dimension and Δy in the y-dimension:

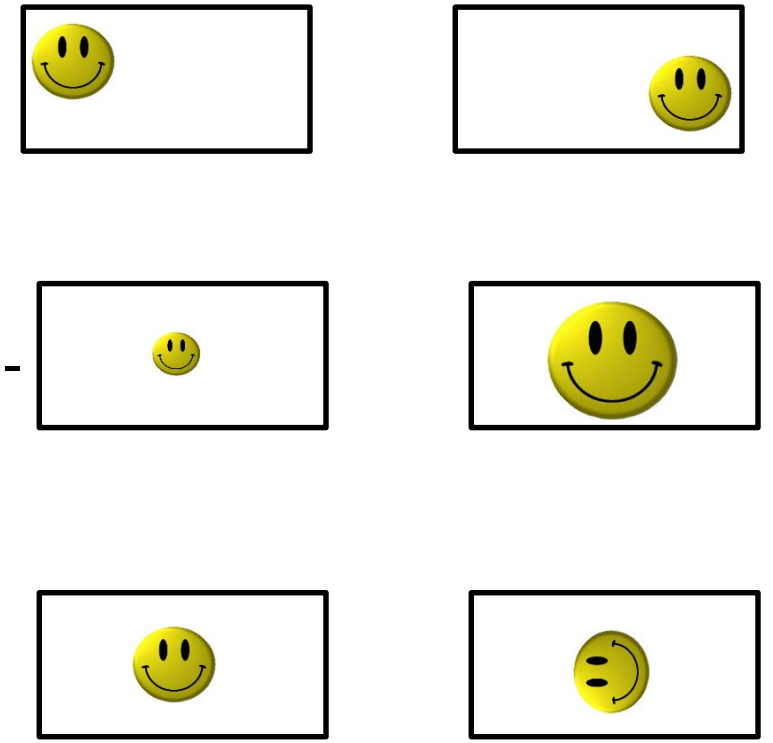
    $$x' = x + \Delta x, \quad y' = y + \Delta y$$

  - Scaling: object scaled by a factor of Sx in the x-direction and Sy in the y-direction;

    $$x' = xS_x, \quad y' = yS_y$$

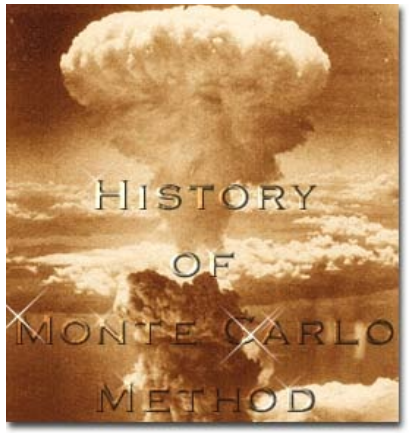
  - Rotation: object rotated through the angle θ about the origin of the coordinate system:

    $$x' = \quad x\cos\theta + y\sin\theta$$
    $$y' = -x\sin\theta + y\cos\theta$$



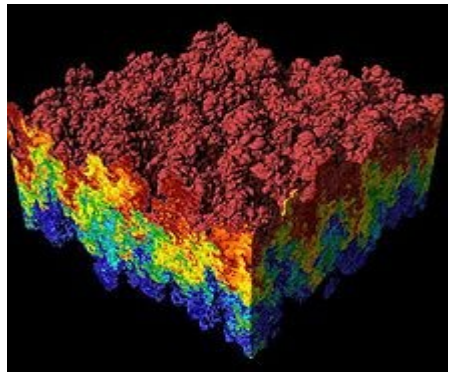
Example: Monte Carlo Methods

- Monte Carlo methods: a class of computational algorithms that rely on repeated random sampling to compute their results



  - Invented in 1940s by *John von Neumann*, *Stanislaw Ulam* and *Nicholas Metropolis*, while they were working on nuclear weapon (Manhattan Project)

  - Especially useful for simulating systems with many coupled degrees of freedom, such as fluids, disordered material

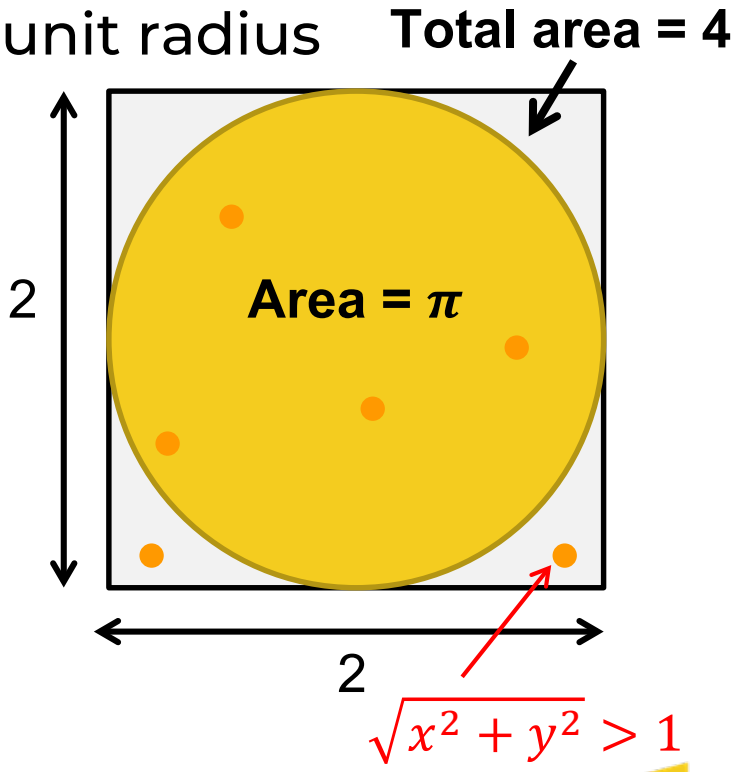


Monte Carlo Methods --- π calculation

- How to compute π ?
  - Definition of π: the area of a circle with unit radius
  - We know: $\frac{\textbf{Area of circle}}{\textbf{Area of square}} = \frac{\pi}{4}$
  - Randomly choose points from the square
  - Giving sufficient number of samples, the fraction of points **within** the circle will be π/4!!!
  - E.g.: With 10,000 randomly sample points

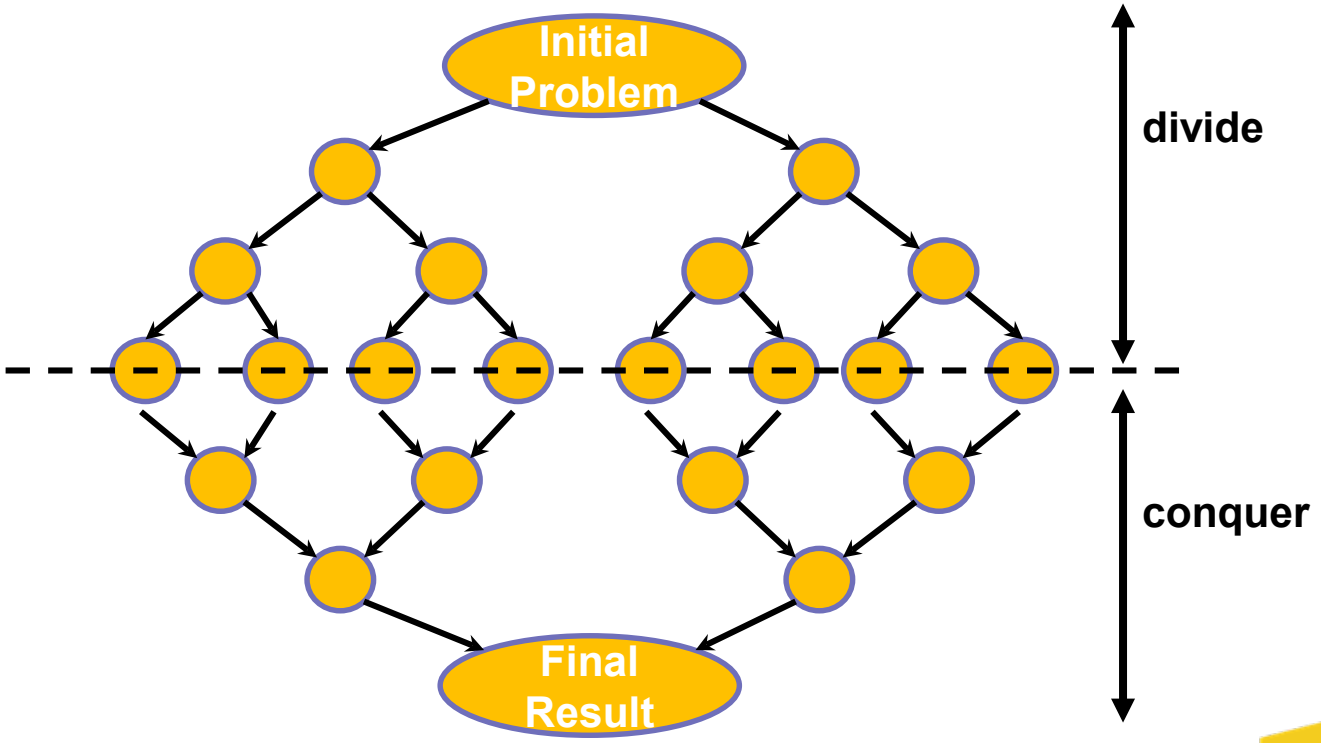
we expect 7854 points within the circle

➔7854/10000 =π/4 ➔π = 7854/10000*4 = 3.1416



Method 2: Divide & Conquer

- Recursively divide a problem into sub-problems that are of the same form as the larger problem



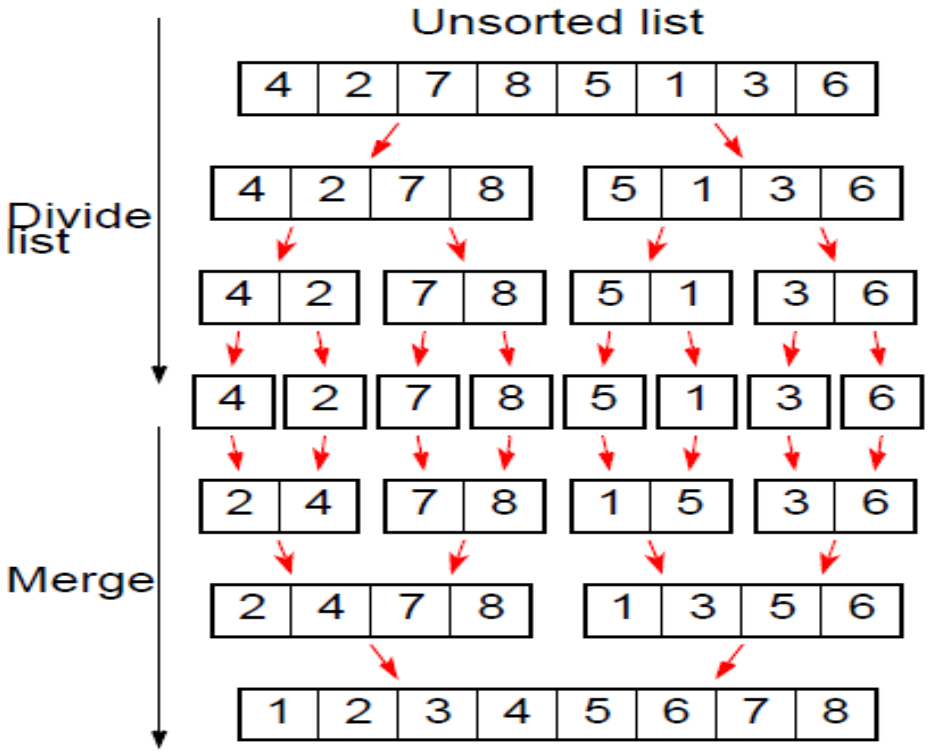
Merge Sort

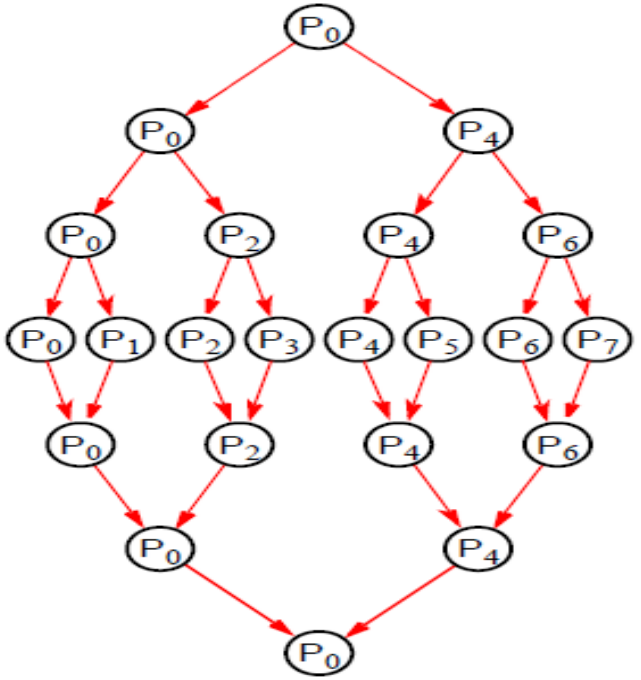
- Divide & Conquer
  - Sequential: $O(n \log n)$
  - Parallel: $O(n)$

#elements

$$T_{comm} = O\left(2\left(\frac{n}{2} + \frac{n}{4} + \cdots + 1\right)\right) = O(n)$$

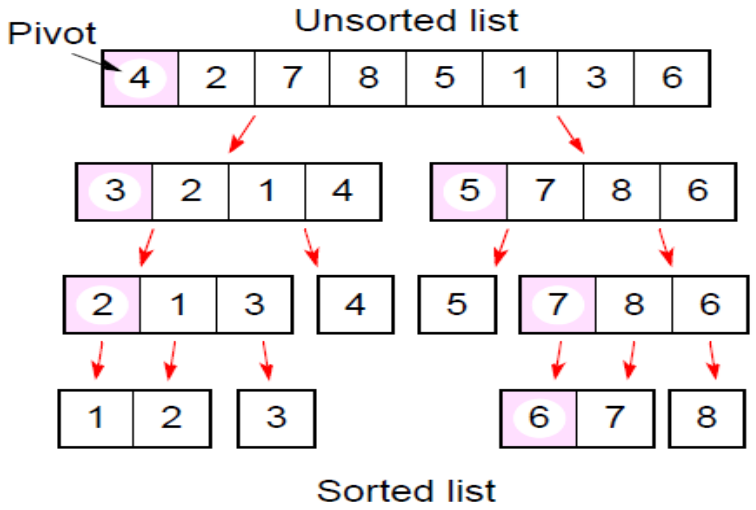
$$T_{comp} = O\left(n + \frac{n}{2} + \frac{n}{4} + \cdots 2\right) = O(n)$$



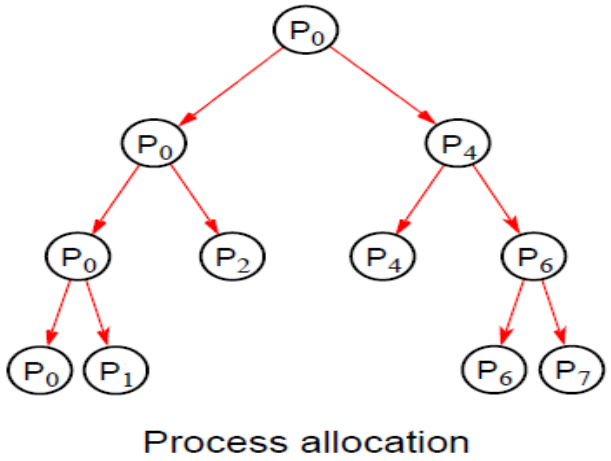


Quick Sort

- Most popular sequential sorting algorithm
  - Parallel: Iteratively pick pivot and partition numbers
- Complexity:
  - Sequential: $O(n \log n)$
  - Parallel: $O(n)$

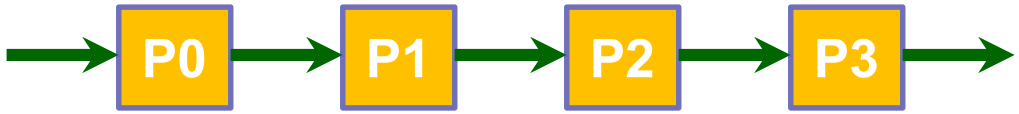


Still best choose in parallel?
Not really & load might not be balanced

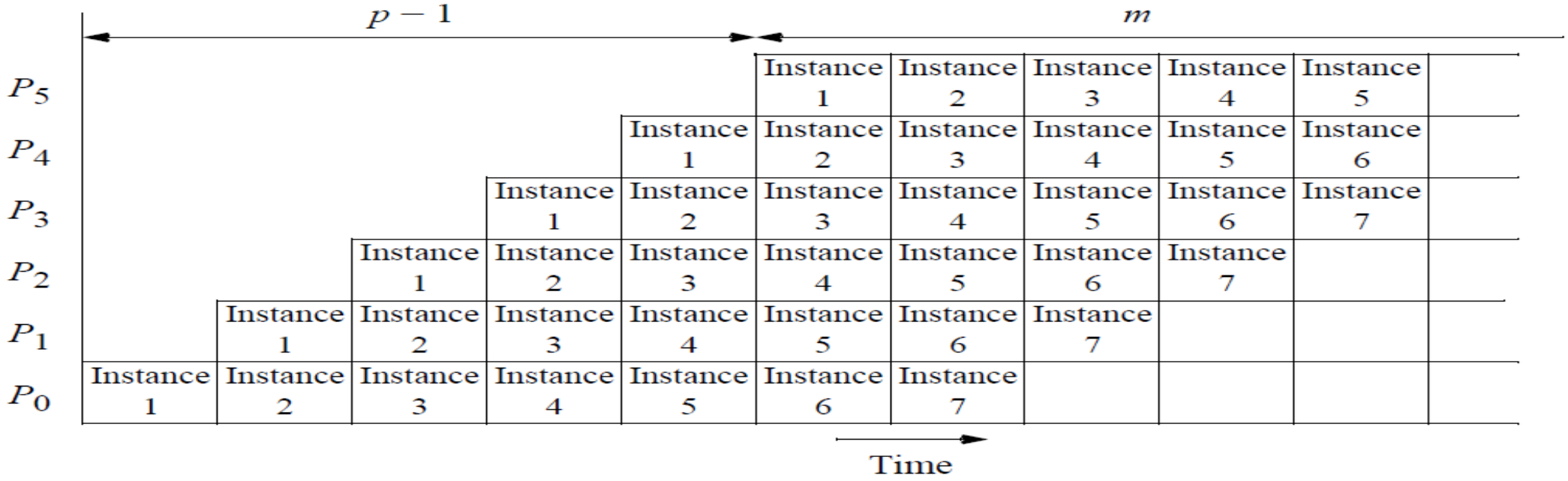


Method 3: Pipelined Computations

- A problem is divided into a series of tasks
- Tasks have to be completed one after the other
- Each task will be executed by a separate process or processor

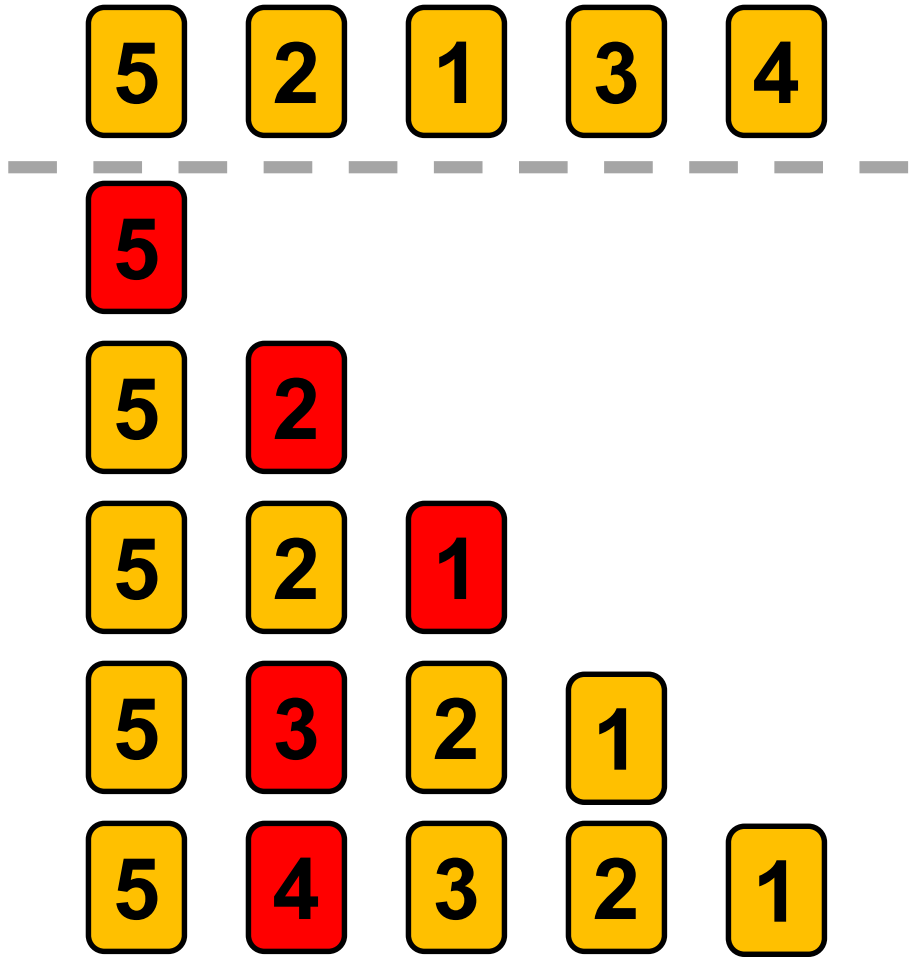


Pipelined Computations



- After the first (p-1) cycles, one problem instance is completed in each pipeline cycle
- The number of instance should be >> the number of processes

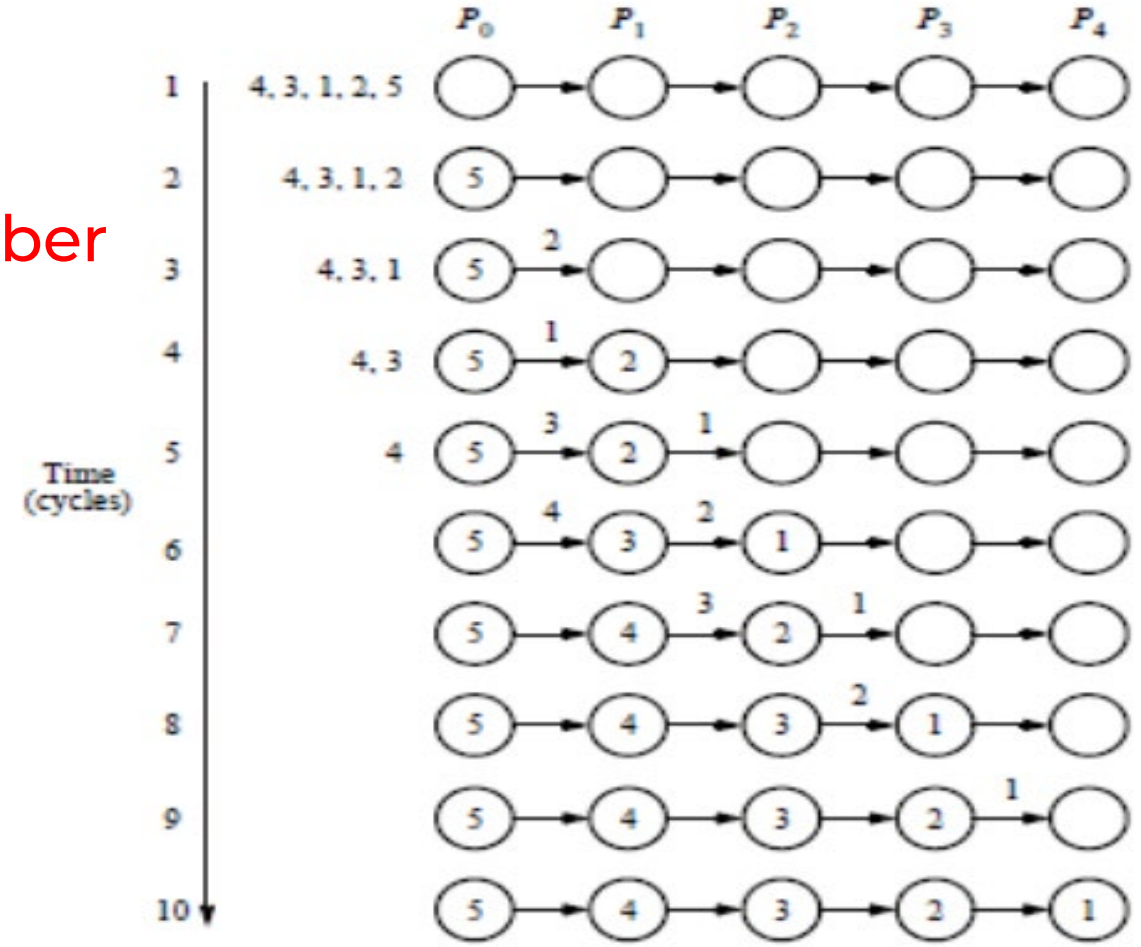
Example: Insertion Sort



Example: Insertion Sort

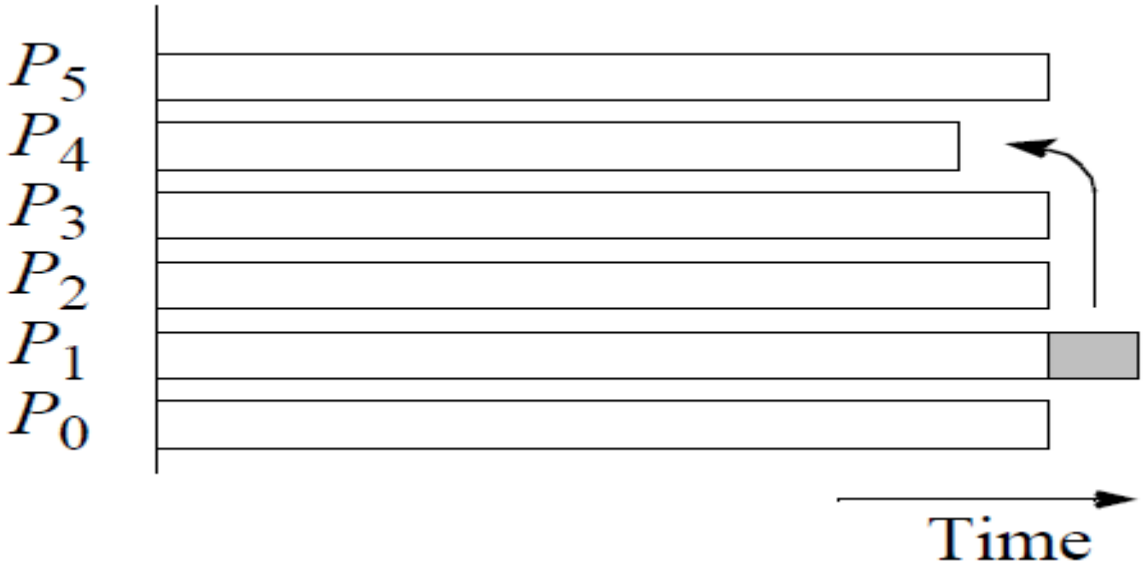
- Each process holds one number
- Compare & move the smaller number to the right

```
recv(&number, P_{i-1});
if (number > x) {
    send(&x, P_{i+1});
    x= number;
} else {
    send(&number, P_{i+1});
}
```



Challenge1: Load-Balancing

- *Load-balancing*
  - Used to distribute computations fairly across processors in order to obtain the highest possible execution speed

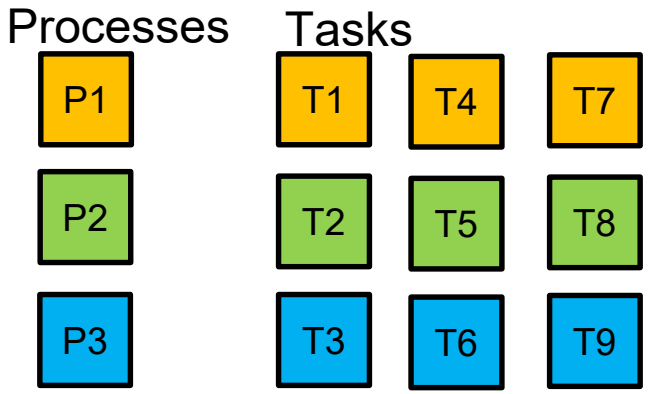


Parallel Computing Intro

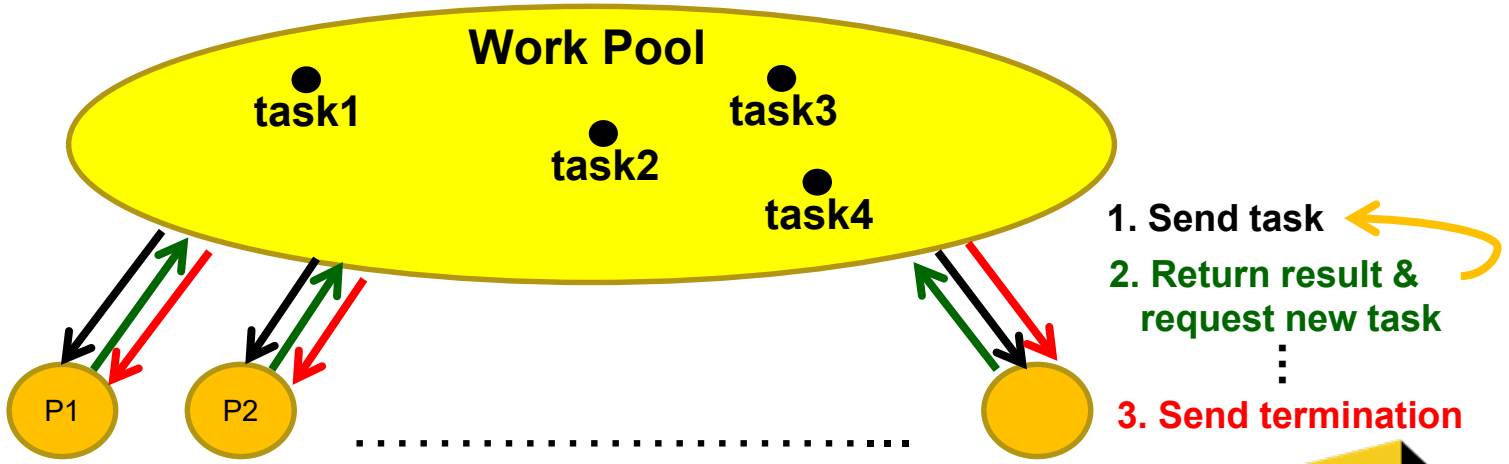
- Parallel Computing Methods:
  - Embarrassingly Parallel
  - Divide-and-Conquer
  - Pipeline
- Parallel Computing Challenges:
  - Load-Balancing
  - Race condition
  - Data dependency
- SHARED-MEMORY PROGRAMMING: Pthread & OpenMP
- MESSAGE-PASSING PROGRAMMING: MPI

Challenge1: Load-Balancing

- Static
  - *Pre-determine assignment between tasks and processes*
  - *E.g.,*
    - *Round robin*
    - *Recursive bisection*



- Dynamic
  - *Assign tasks to processes during the* execution
  - E.g.,
    - First-Come-First-Serve



Challenge2: Race Condition

- Definition: The outcome of a shared data content is decided by the execution order among processes
- Cause: Instructions of individual processes/threads may be interleaved in time
- E.g.: Assume variable "counter" is shared by processes
- The statement "*counter++*" & "counter--"may be implemented in machine language as:

```
move ax, counter        move  bx, counter
add    ax, 1            sub     bx, 1
move counter, ax        move  counter, bx
```

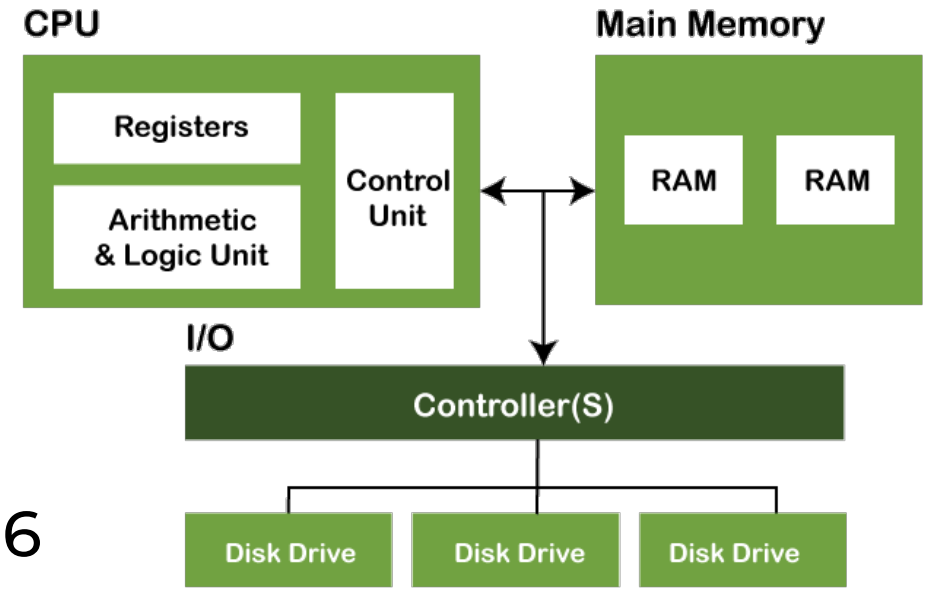
```
Process0                    Process1
 main() {                    main() {
      ⋮                           ⋮
   counter++;                    counter--
      ⋮                           ⋮
}                           ;     ⋮
                             }
```

Instruction Interleaving

- Assume counter is initially 5. One interleaving of statement is:

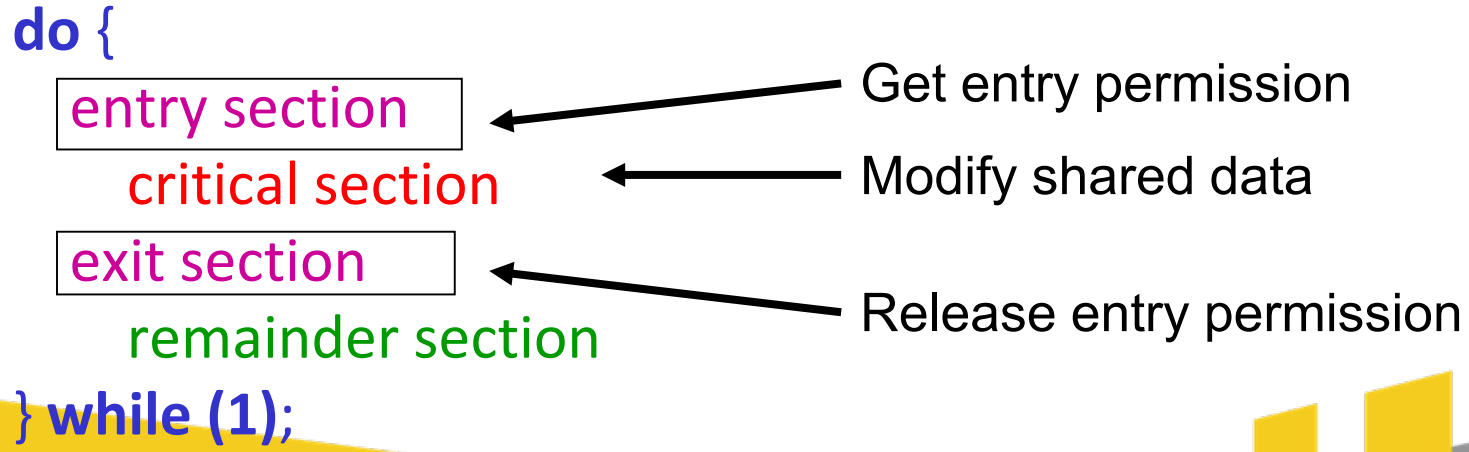
  producer: move ax, counter ➔ ax = 5
  producer: add ax, 1 ➔ ax = 6
  context switch
  consumer: move bx, counter ➔ bx = 5
  consumer: sub bx, 1 ➔bx = 4
  context switch
  producer: move counter, ax ➔ counter = 6
  context switch
  consumer: move counter, bx ➔ counter = 4

- The value of counter may be either 4, 5, or 6
- The ONLY correct result is 5!



Critical Section & Mutual Exclusion

- Critical Section is a piece of code that can only be accessed by one process/thread at a time
- Mutual exclusion is the problem to insure only one process/thread can be in a critical section
- E.g.: The design of entry section & exit section provides mutual exclusion for the critical section



Critical Section & Mutual Exclusion

```
Process0                    Process1
  main() {                    main() {
      ⋮                           ⋮
    lock()                      unlock()
    counter++;                  counter--
    unlock()                    unlock();
      ⋮                           ⋮
}                           }
```

producer: move ax, counter → ax = 5
producer: add ax, 1 → ax = 6
producer: move counter, ax → counter = 6
context switch
consumer: move bx, counter → bx = 6
consumer: sub bx, 1 → bx = 6
consumer: move counter, bx → counter = 5

consumer: move bx, counter → bx = 5
consumer: sub bx, 1 → bx = 5
consumer: move counter, bx → counter = 4
context switch
producer: move ax, counter → ax = 4
producer: add ax, 1 → ax = 4
producer: move counter, ax → counter = 5

Challenge3: Data Dependency

- What is data dependency:
  - A situation in which a program statement (instruction) refers to the data of a preceding statement.
  - Parallelism or re-ordering may not be allowed when they occur

- Flow dependency (True dependency)
  - read-after-write (RAW):
    - An instruction depends on the result of a previous instruction
  - E.g.,:
    - Line2 depends on Line1
    - Line3 depends on Line2
    - Line3 depends on Line1

```
1. A = 3
2. B = A
3. C = B
```

Challenge3: Data Dependency

- Can we parallelize the following codes?
  - Each iteration is executed by a thread

```
for (i=0; i<10; i++)
    A[i] = B[i] + 5;
```

```
for (i=0; i<10; i++)
    A[i] = B[i-1] + 5;
```

```
for (i=0; i<10; i++)
    A[i] = B[i] + C[i];
```

```
for (i=0; i<10; i++)
    A[i] = A[i-1] + 5;
```

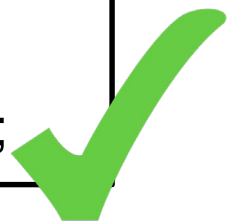
Challenge3: Data Dependency

- Can we parallelize the following codes?
  - Each iteration is executed by a thread

```
for (i=0; i<10; i++)
    A[i] = B[i] + 5;
```

```
for (i=0; i<10; i++)
    A[i] = B[i-1] + 5;
```

```
for (i=0; i<10; i++)
    A[i] = B[i] + C[i];
```

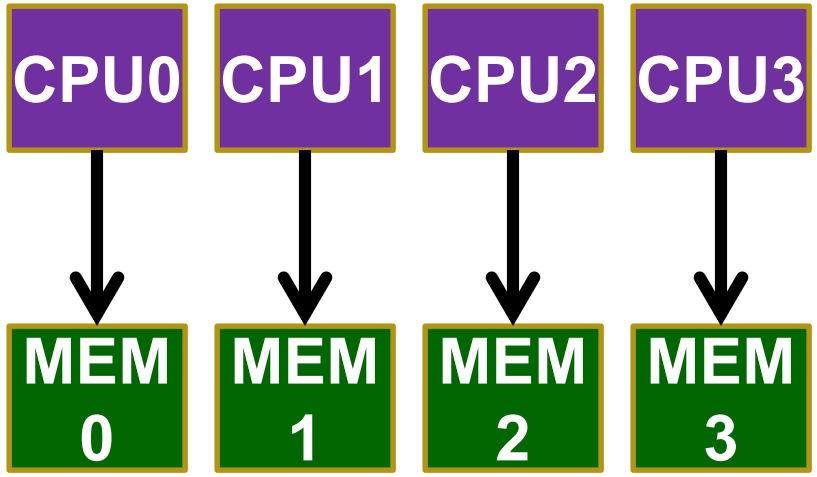


```
for (i=0; i<10; i++)
    A[i] = A[i-1] + 5;
```

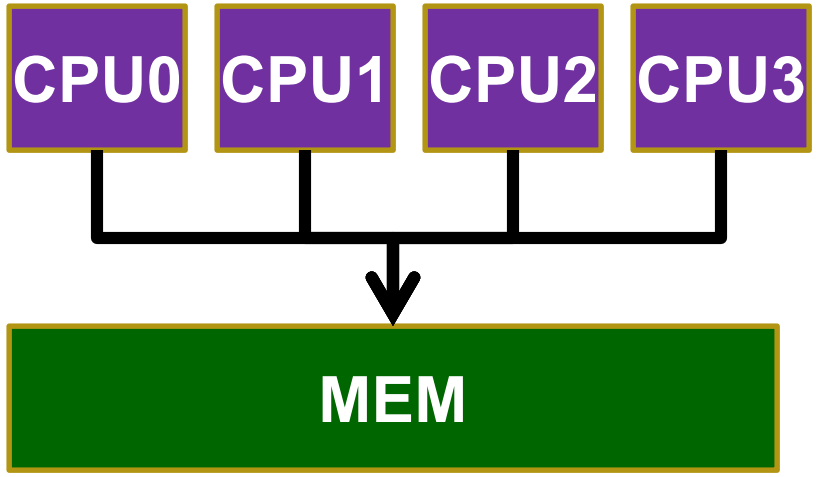
Parallel Computing Intro

- Parallel Computing Methods:
  - Embarrassingly Parallel
  - Divide-and-Conquer
  - Pipeline
- Parallel Computing Challenges:
  - Load-Balancing
  - Race condition
  - Data dependency
- Parallel Programming
  - SHARED-MEMORY PROGRAMMING: Pthread & OpenMP
  - MESSAGE-PASSING PROGRAMMING: MPI

Shared Memory vs. Distributed Memory Computer Architecture



Distributed memory



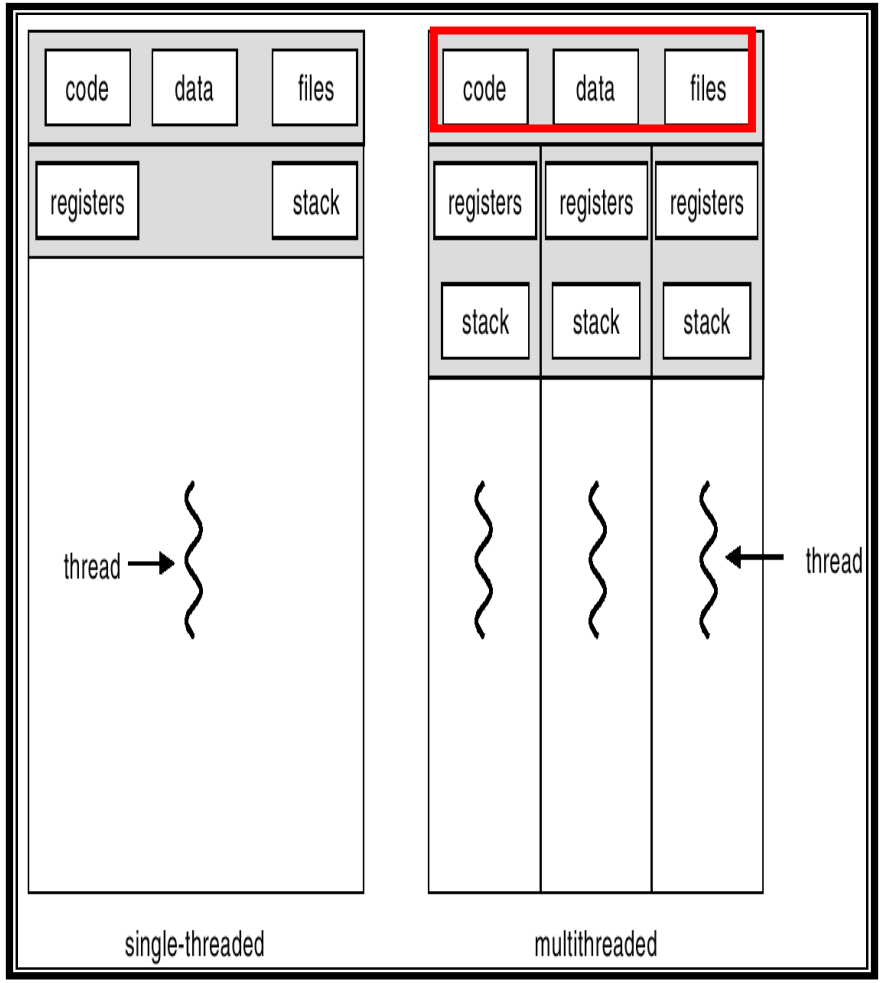
Shared memory

MPI: Message Passing Interface

Pthread: Thread Programming

Process & Thread

- Threads are created by a process
  - Thread is the basic unit for utilizing CPU cores
- All threads belonging to the same process share
  - code, global variable, heap (dynamic allocated memory), resources (e.g. open files)
- But each thread can be executed and scheduled independently



SHARED-MEMORY PROGRAMMING: Pthread & OpenMP

Why Thread? Thread is a lightweight process

- Lower creation/management cost vs. Process

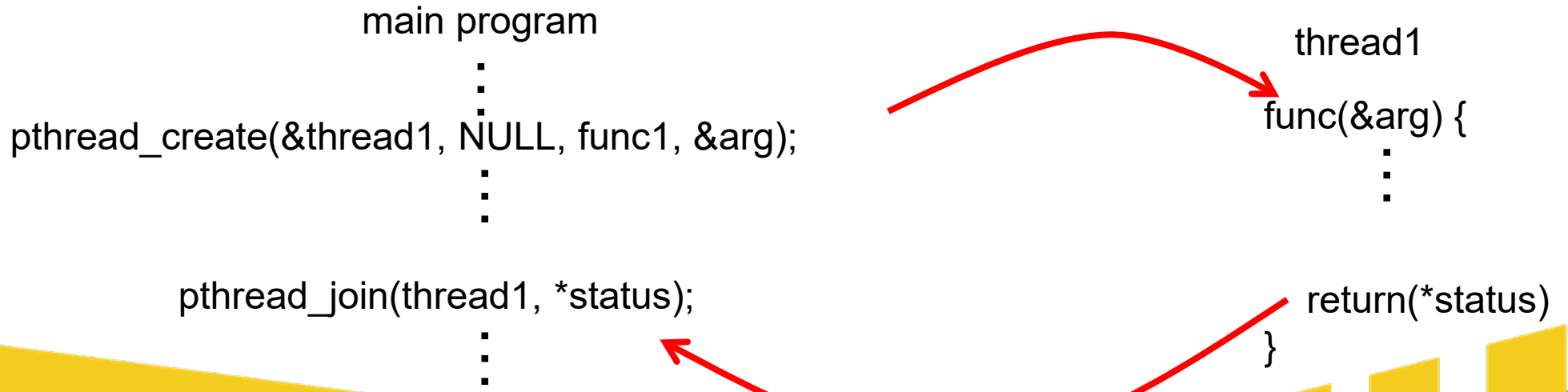
platform	fork()	pthread_create()	speedup
AMD 2.4 GHz Opteron	17.6	1.4	15.6x
IBM 1.5 GHz POWER4	104.5	2.1	49.8x
INTEL 2.4 GHz Xeon	54.9	1.6	34.3x
INTEL 1.4 GHz Itanium2	54.5	2.0	27.3x

- Faster inter-process communication vs. MPI

platform	MPI Shared Memory BW (GB/sec)	Pthreads Memory-to-CPU BW (GB/sec)	speedup
AMD 2.4 GHz Opteron	1.2	5.3	4.4x
IBM 1.5 GHz POWER4	2.1	4	1.9x
INTEL 2.4 GHz Xeon	0.3	4.3	14.3x
INTEL 1.4 GHz Itanium2	1.8	6.4	3.6x

Pthread

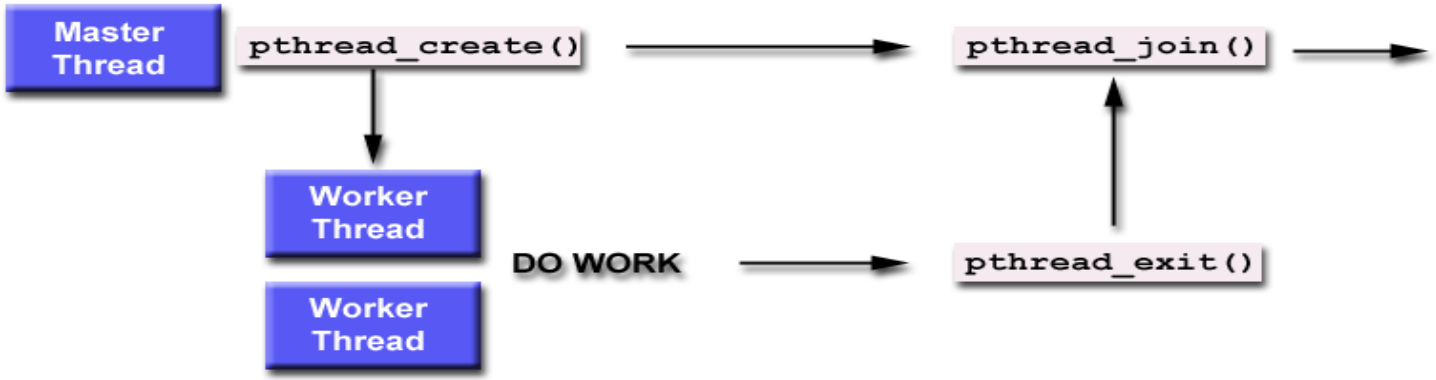
- Pthread: A library specified for portability across Unix-like systems
- pthread_create(thread,attr,routine,arg)
  - thread: An unique identifier (token) for the new thread
  - attr: It is used to set thread attributes. NULL for the default values
  - routine: The routine that the thread will execute once it is created
  - arg: A single argument that may be passed to *routine*



Pthread

- pthread_join(threadId, status)
  - Blocks until the specified threadId thread terminates
  - One way to accomplish synchronization between threads
  - Example: to create a pthread barrie

```
for (int i=0; i<n; i++)  pthread_join(thread[i], NULL);
```



Pthread: Lock/Mutex Routines

- To use mutex, it must be declared as of type pthread_mutex_t and initialized with pthread_mutex_init()
- A mutex is destroyed with pthread_mutex_destroy()
- A critical section can then be protected using pthread_mutex_lock() and pthread_mutex_unlock()
- Example:

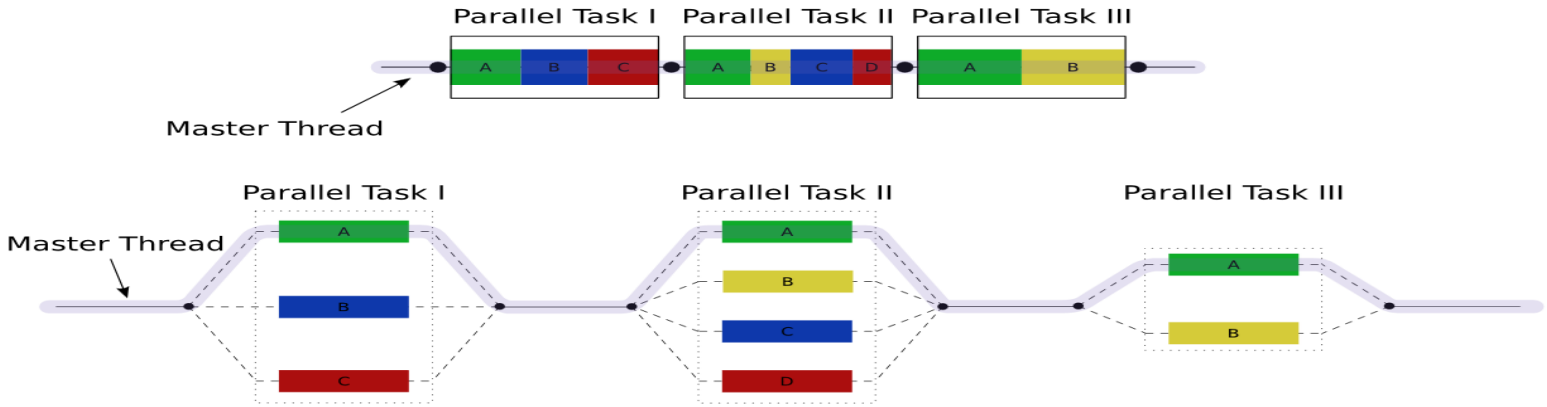
```
#include “pthread.h”
pthread_mutex_t   mutex;
pthread_mutex_init (&mutex, NULL);
pthread_mutex_lock(&mutex);          // enter critical section

          Critical Section

pthread_mutex_unlock(&mutex);        // leave critical section
pthread_mutex_destroy(&mutex);
```

specify default attribute for the mutex

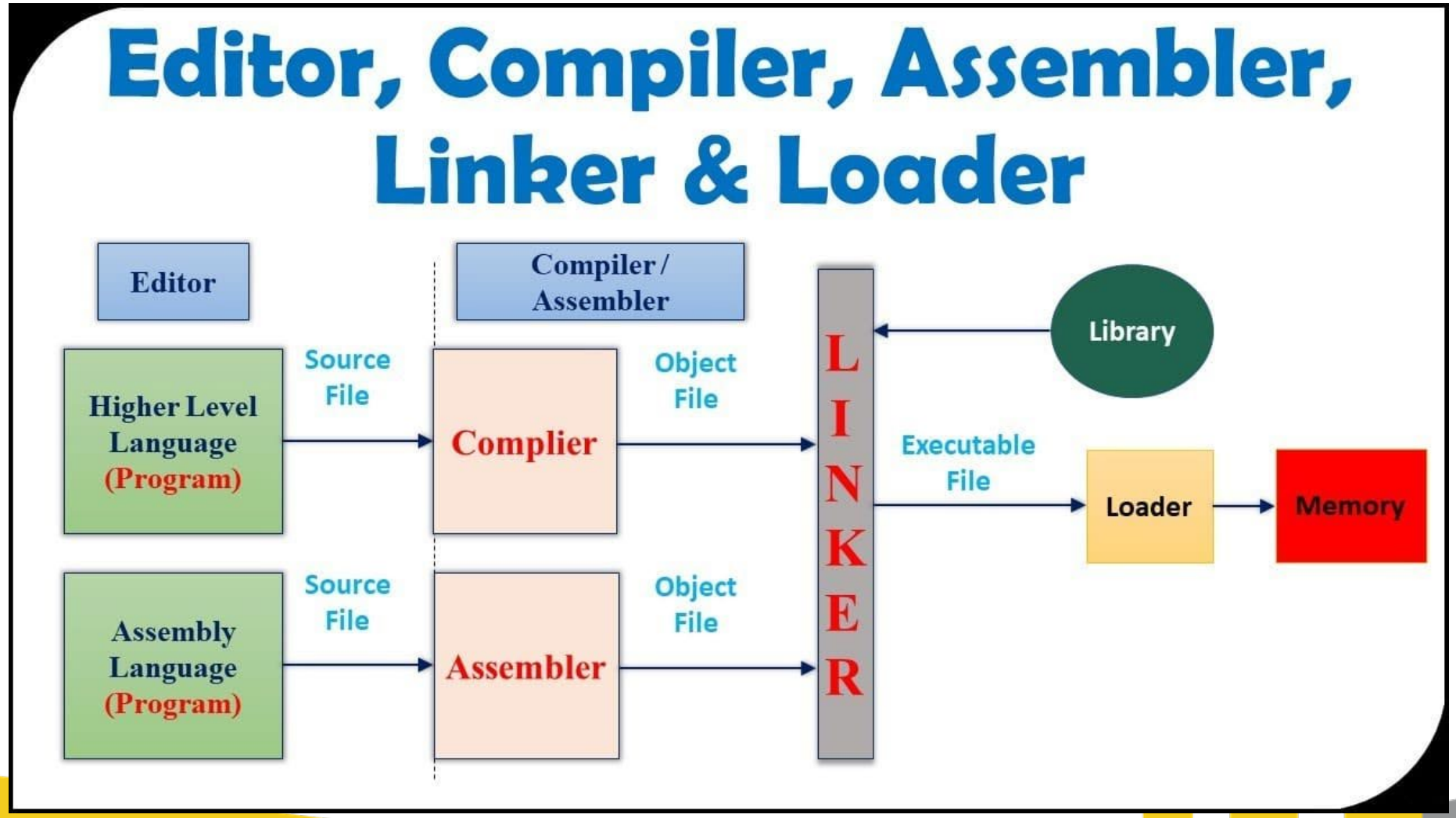
What's OpenMP

OpenMP == Open specification for Multi-Processing

- An API : multi-threaded, shared memory parallelism
- Portable: the API is specified for C/C++ and Fortran
- Fork-Join model: the master thread forks a specified number of slave threads and divides task among them
- Compiler Directive Based: Compiler takes care of generating code that forks/joins threads and divide tasks to threads



Program Compilation & Execution



Example

- Add two data arrays in parallel by specifying compiler directives:
  - Slave threads are forked and each thread works on different iterations

```
#include <omp.h>
// Serial code
int A[10], B[10], C[10];

// Beginning of parallel section. Fork a team of threads.
#pragma omp parallel for num_threads(10)
{
for (int i=0; i<10; i++)
    A[i] = B[i] + C[i];
} /* All threads join master thread and terminate */
```

OpenMP Directives

- C/C++ Format:

#pragma omp	directive-name	[clause, ...]	newline
Required.	Valid OpenMP directive: **parallel, do, for**	**Optional**. Clauses can be in any order, and repeated as necessary.	Required.

- Example:
  - #pragma omp parallel default(shared) private(beta,pi)

directive-name clause clause

- General Rules:
  - Case sensitive
  - Only one directive-name may be specified per directive
  - Each directive applies to at most one succeeding statement, which must be a structured block

Parallel Region Constructs --- Parallel Directive

- A parallel region is a block of code executed by multiple threads

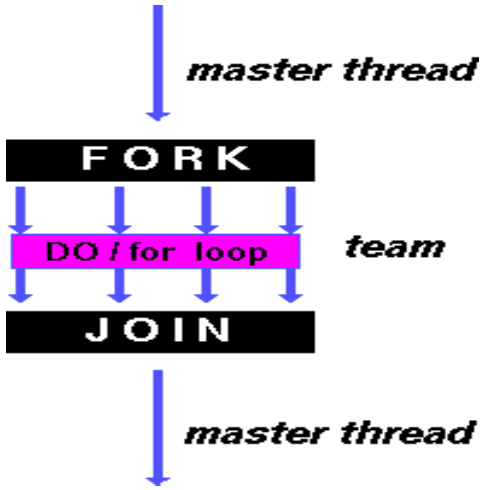
```
#pragma omp parallel [clause ……]
                        if (scalar_expression)
                        num_threads (integer-expression)
structured_block
```

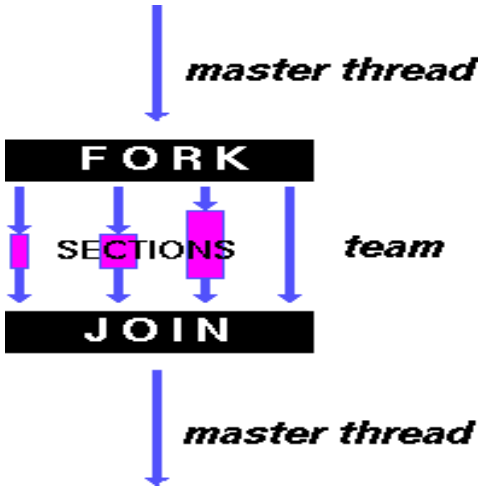
- Overview:
  - When PARALLEL is reached, a team of threads is created
  - The parallel region code is duplicated and executed by all threads
  - There is an implied barrier at the end of a parallel section.
  - One thread terminates, all threads terminate
- Limitations:
  - A parallel region must be a structured block that does not span multiple routines or code files

Work-Sharing Constructs

- Definition:
  - A work-sharing construct divides the execution of the enclosed code region among the threads that encounter it
  - Work-sharing constructs DO NOT launch new threads, hence it should be enclosed within a parallel region for parallelism

DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".





SECTIONS - breaks work into separate, discrete sections of code. Each section is executed by a thread.

DO / for Directive

- Purpose: indicate the iterations of the loop immediately following it must be executed in parallel by the team of threads

```
#pragma omp for [clause ……]
                        schedule (type [,chunk])
                        ordered
                        nowait
                        collapse (n)
for_loop
```

```
// Beginning of parallel section. Fork a team of threads.
#pragma omp parallel for num_threads(10)
{
for (int i=0; i<10; i++)
    A[i] = B[i] + C[i];
} /* All threads join master thread and terminate */
```

- Do/for Directive Specific Clauses:
  - nowait: Do not synchronize threads at the end of the loop
  - schedule: Describes how iterations are divided among threads
  - ordered: Iterations must be executed as in a serial program
  - collapse: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause

DO / for Directive --- Schedule Clause

- STATIC
  - Loop iterations are divided into chunks
  - If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads
  - Then statically assigned to threads
- DYNAMIC:
  - When a thread finishes one chunk (default size: 1), it is dynamically assigned another

E.g.,: A for loop with 100 iterations and 4 threads:

- schedule(static, 10)
  - Thread0: Iter0-10, Iter40-50, Iter80-90
  - Thread1: Iter10-20, Iter50-60, Iter90-100
  - Thread2: Iter20-30, Iter60-70
  - Thread3: Iter30-40, Iter70-80
- schedule(dynamic, 10)
  - Thread0: Iter0-10, Iter70-80, Iter80-90, Iter90-100
  - Thread1: Iter10-20, Iter50-60
  - Thread2: Iter20-30, Iter60-70
  - Thread3: Iter30-40, Iter40-50

DO / for Directive --- Schedule Clause

- GUIDED:
  - Similar to DYNAMIC except chunk size decreases over time (better load balancing)

E.g., schedule(guided, 10)

- Thread0: Iter0-10, Iter40-50, Iter80-85
- Thread1: Iter10-20, Iter50-60, Iter85-90
- Thread2: Iter20-30, Iter60-70, Iter90-95
- Thread3: Iter30-40, Iter70-80, Iter95-100

Critical Directive

- Advantage of using critical over lock:
  - no need to declare, initialize and destroy a lock
  - you always have explicit control over where your critical section ends
  - Less overhead with compiler assist

```
#include <omp.h>
main () {
   int count=0;
   #pragma omp parallel num_threads(10)
            #pragma omp critical
                     count++;
}
```

```
#include <omp.h>
main () {
   int count=0;
   omp_lock_t *lock;
   omp_init_lock(lock)
   #pragma omp parallel num_threads(10)
    {
            omp_set_lock(lock);
            count++;
            omp_unset_lock(lock);
    }
    omp_destory_lock(lock)
}
```

OpenMP Data Scope

- This is critical to understand the scope of each data
  - OpenMP is based on shared memory programming model
  - Most variables declared outside a parallel region are shared by default
- Global shared variables:
  - File scope variables, static
- Private non-shared variables:
  - Loop index variables
  - Variables declared in subroutines called from parallel regions

```
int A;
#pragma omp parallel for
   for (i=0; i < n; i++)
     int B = 10;
     A = rand();
```

Data Scope Attribute Clauses

- Data scope can be explicitly defined by clauses…
  - PRIVATE (var_list): Declares variables in its list to be private to each thread; variable value is NOT initialized & will not be maintained outside the parallel region
  - SHARED (var_list): Declares variables in its list to be shared among all threads
- By default, all variables in the work sharing region are shared except the loop iteration counter.

```
#pragma omp parallel shared (var1)
{
  int var1 = 10;
  printf("var1:%d" var1);
}
```

```
int var1 = 10;
#pragma omp parallel private (var1)
{
  printf("var1:%d" var1);
}
```

Data Scope Attribute Clauses

- FIRSTPRIVATE (var_list):
  - Same as PRIVATE clause, but the variable is INITIALIZED according to the value of their original objects prior to entry into the parallel region
- LASTPRIVATE (var_list)
  - Same as PRIVATE clause, with a copy from the LAST loop iteration or section to the original variable object
- REDUCTION (operator: var_list)
  - A private copy for each list variable is created for each thread
  - Performs a reduction on all variable instances
  - Write the final result to the global shared copy

Examples

- firstprivate (var_list)

```
int var1 = 10;
#pragma omp parallel firstprivate (var1)
{
        printf("var1:%d" var1);
}
```

- lastprivate (var_list)

```
int var1 = 10;
#pragma omp parallel lastprivate (var1) num_thread(10)
{
        int id = omp_get_thread_num();
        sleep(id);
        var1=id;
}
printf("var1:%d", var1);
```

Reduction Clause Example

- Reduction operators: +, *, &, |, ^, &&, ||

```
#include <omp.h>
main () {
   int i, n, chunk, a[100], b[100], result;
   n = 10; chunk = 2; result = 0;
   for (i=0; i < n; i++) a[i] = b[i] = I;

   #pragma omp parallel for default(shared) private(i)
                              schedule(static,chunk) reduction(+:result)
    {
         for (i=0; i < n; i++) result = result + (a[i] * b[i]);
    }
    printf("Final result= %f\n",result);
}
```

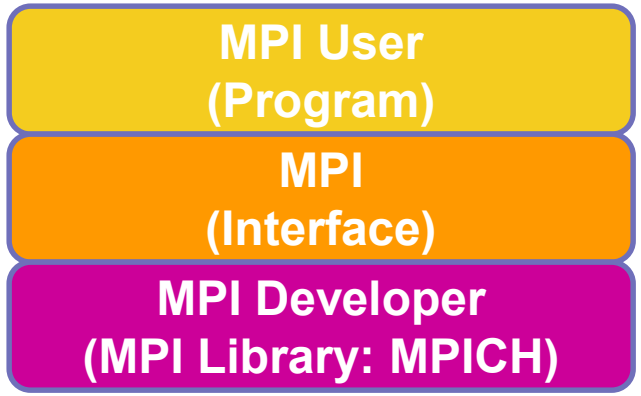
Knowledge Check

- Why threads could be better than processes?
- What does compiler directive mean?
- What is fork-and-join model? What is parallel region?
- Can we have multiple causes for a directive?
- When can’t we parallelize a for-loop?
- What does private and shared data scope mean?

MESSAGE-PASSING PROGRAMMING: MPI

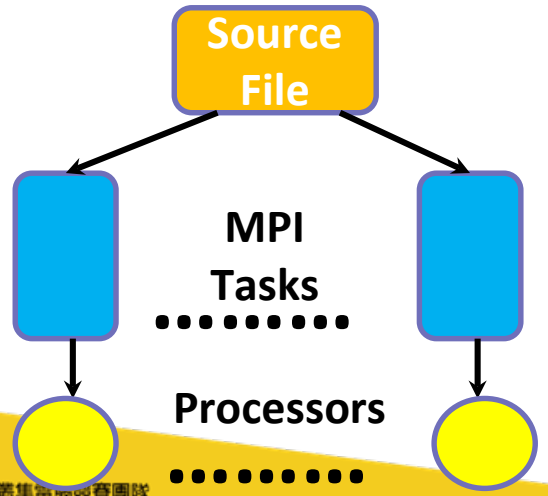
What is MPI

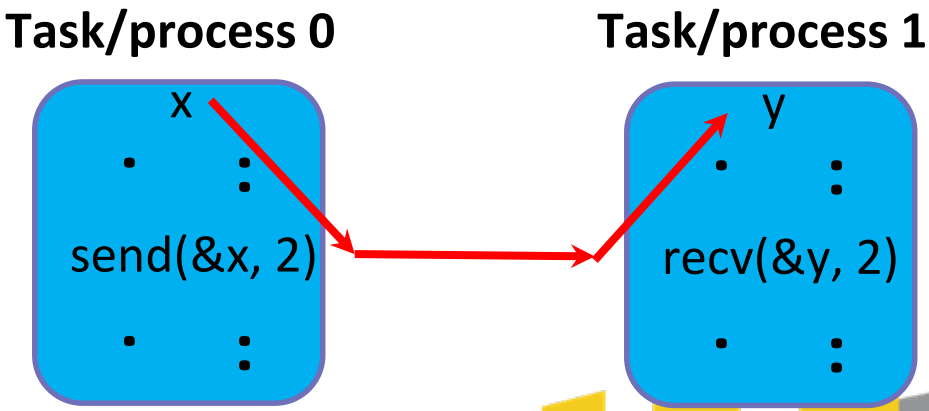
- MPI = Message Passing Interface
- A specification for the
  developers and users
  of message passing libraries
  - By itself, it is an interface NOT a library
- Commonly used for distributed memory system & high-performance computing
- Goal:
  - Portable: Run on different machines or platforms
  - Scalable: Run on million of compute nodes
  - Flexible: Isolate MPI developers from MPI programmers (users)



Programming Model

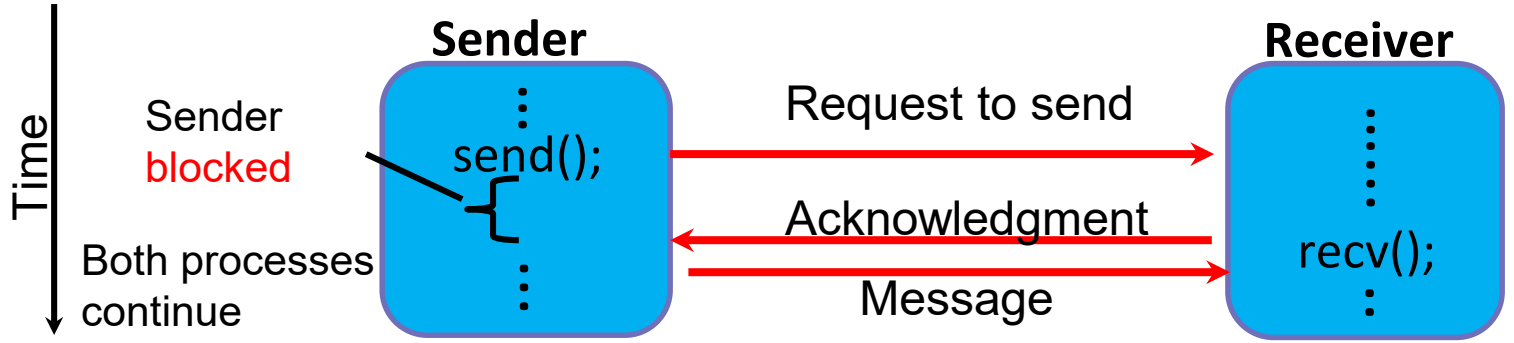
- SPMD: Single Program Multiple Data
  - Allow tasks to branch or conditionally execute only parts of the program they are designed to execute
  - MPI tasks of a parallel program can not be dynamically spawned during run time. (MPI-2 addresses this issue).
- Distributed memory
  - MPI provide method of sending & receiving message



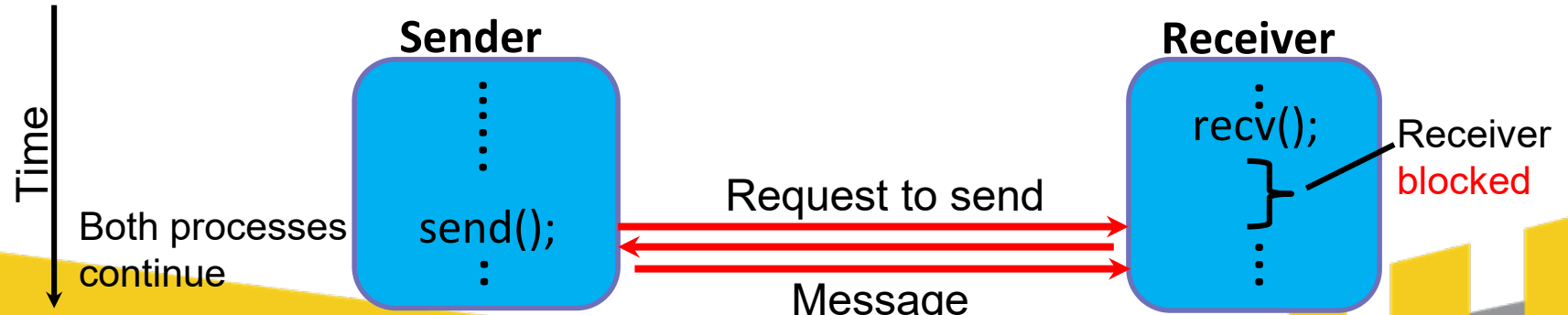


Synchronous/Blocking Message Passing

- Sender: wait until the complete message can be accepted by the receiver before sending the message

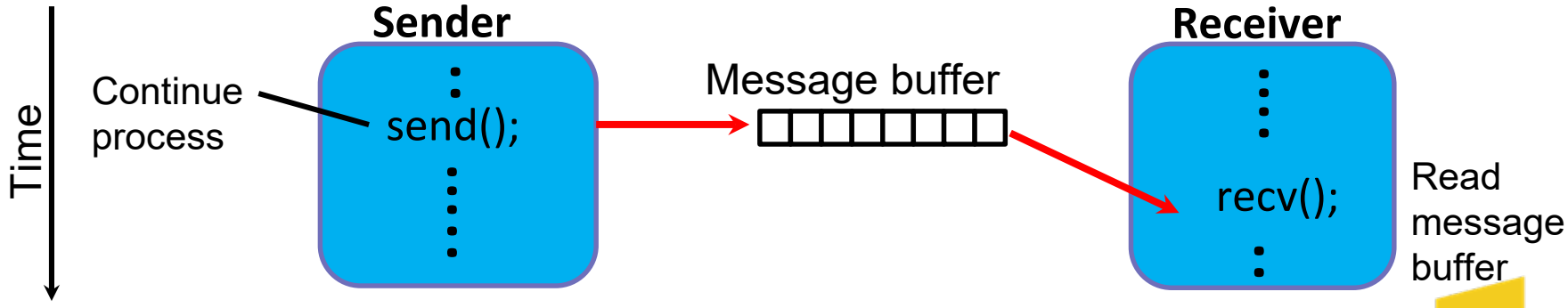


- Receiver: wait until the message it is expecting arrives



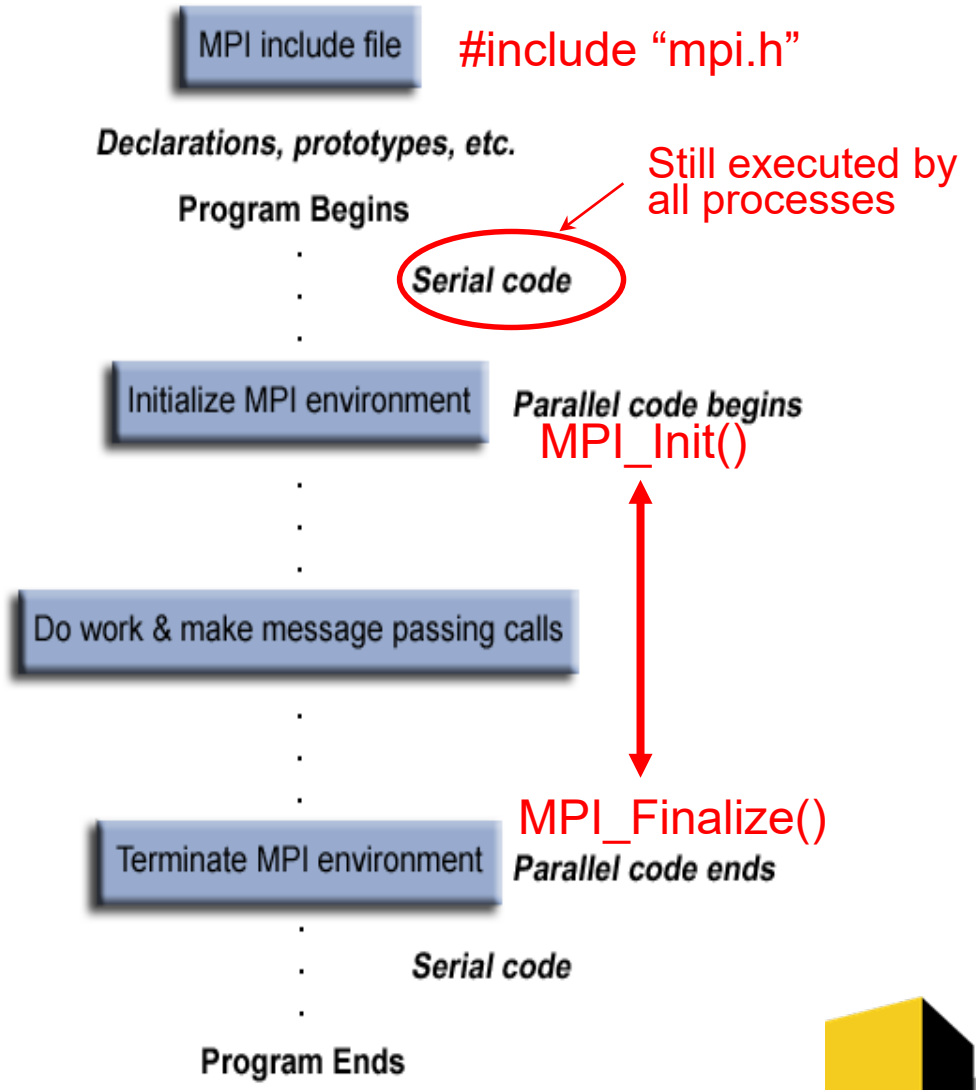
Asynchronous/Non-Blocking Message Passing

- Message-passing routines can return before the message transfer has been completed
  - Generally, a message buffer needed between source and destination to hold message
  - Message buffer is a memory space at the sender and/or receiver side
  - For send routine, once the local actions have been completed and the message is safely on its way, the process can continue with subsequent work



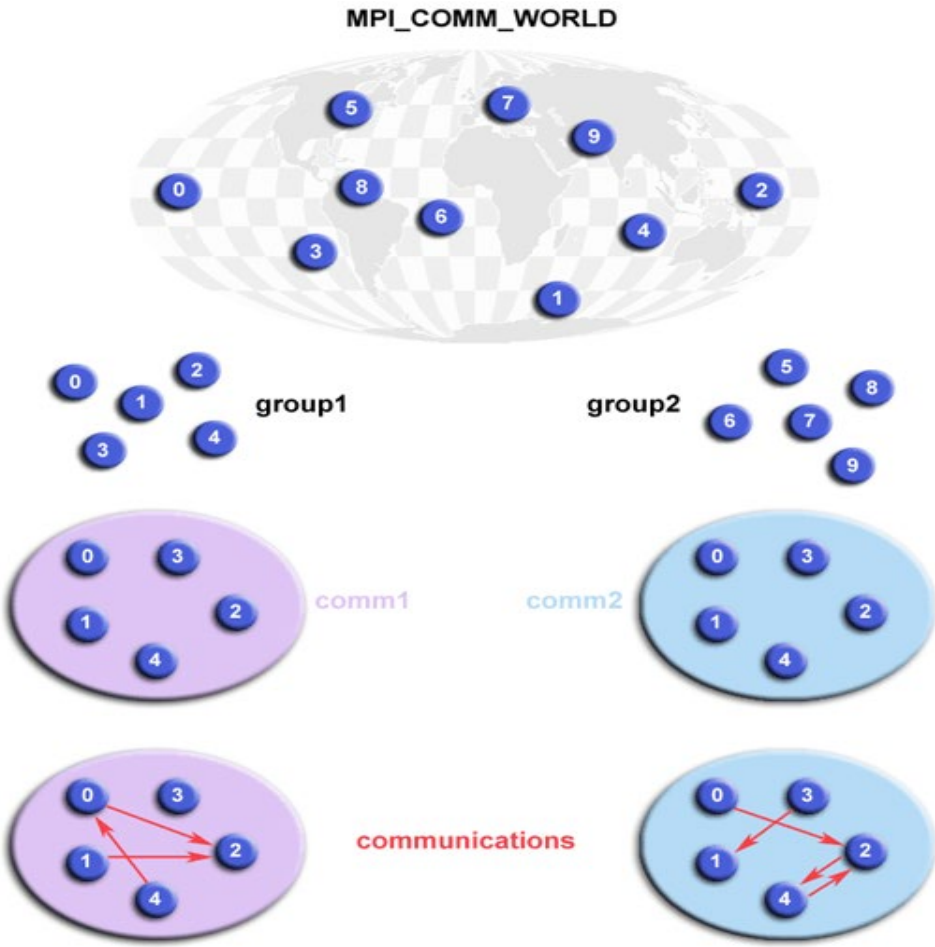
Getting Start

- Header file: “mpi.h”
  - Required for all programs that make MPI library call
- MPI calls:
  - Format: rc = MPI_Xxx(parameter, …)
  - Example: rc = MPI_Bcast (&buffer,count,datatype,root,comm)
  - Error code: return as “rc”; rc=MPI_SUCCESS if successful
- General MPI program structure:



Getting Start

- Communicators and Groups:
  - Groups define which collection of processes may communicate with each other
  - Each group is associated with a communicator to perform its communication function calls
  - MPI_COMM_WORLD is the pre-defined communicator for all processors
- Rank
  - An unique identifier (task ID) for each process in a communicator
  - Assigned by the system when the process initializes
  - Contiguous and begin at zero



Environment Management Routines

- MPI_Init ()
  - Initializes the MPI execution environment
  - Must be called before any other MPI functions
  - Must be called only once in an MPI program
- MPI_Finalize ()
  - Terminates the MPI execution environment
  - No other MPI routines may be called after it
- MPI_Comm_size (comm, &size)
  - Determines the number of processes in the group associated with a communicator
- MPI_Comm_rank (comm, &rank)
  - Determines the rank of the calling process within the communicator
  - This rank is often referred to as a task ID

Example

```
#include "mpi.h"
int main (int argc, char *argv[]) {
   int numtasks, rank, rc;
   rc = MPI_Init (&argc,&argv);
   if (rc != MPI_SUCCESS) {
            printf ("Error starting MPI program. Terminating.\n");
            MPI_Abort (MPI_COMM_WORLD, rc);
   }
   MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   printf ("Number of tasks= %d My rank= %d\n", numtasks, rank);
   MPI_Finalize ();
}
```

Point-to-Point Communication Routines

Blocking send	MPI_Send(buffer,count,type,dest,tag,comm)
Non-blocking send	MPI_Isend(buffer,count,type,dest,tag,comm,request)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm,status)
Non-blocking receive	MPI_Irecv(buffer,count,type,source,tag,comm,request)

- buffer: Address space that references the data to be sent or received
- type: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_DOUBLE, …
- count: Indicates the number of data elements of a particular type to be sent or received
- comm: indicates the communication context
- source/dest: the rank (task ID) of the sender/receiver
- tag: arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations must match message tags. MPI_ANY_TAG is the wild card.
- status: status after operation
- request: used by non-blocking send and receive operations

Blocking Example

Blocking send	MPI_Send(buffer,count,type,dest,tag,comm)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm,status)

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank); /* find process rank */
if (myRank == 0) {
    int x=10;
    MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (myRank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, status);
}
```

Non-Blocking Example

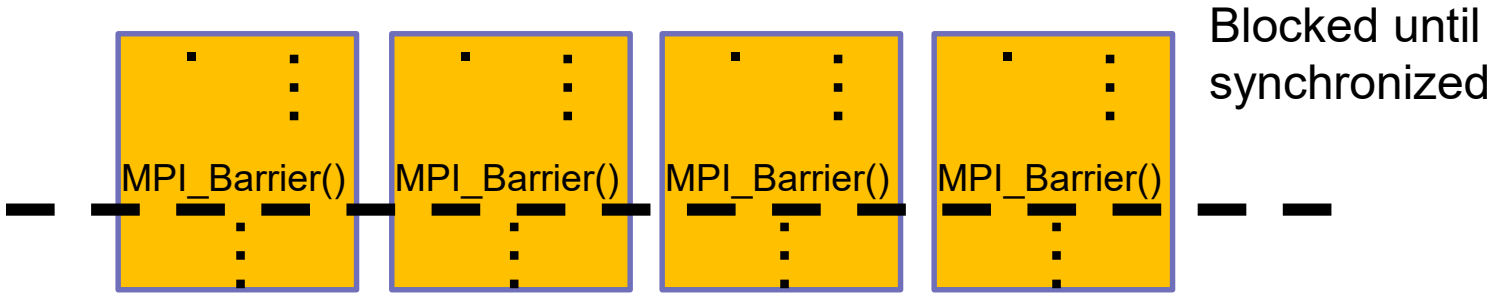
- MPI_Wait() blocks until the operation has actually completed
- MPI_Test() returns with a flag set indicating whether operation completed at that time.

Non-Blocking send	MPI_ISend(buffer,count,type,dest,tag,comm,request)
Non-Blocking receive	MPI_IRecv(buffer,count,type,source,tag,comm,requst)

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {
    int x=10;
    MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req1);
    compute();
} else if (myrank == 1) {
    int x;
    MPI_Irecv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, req1);
}
MPI_Wait(req1, status);
```

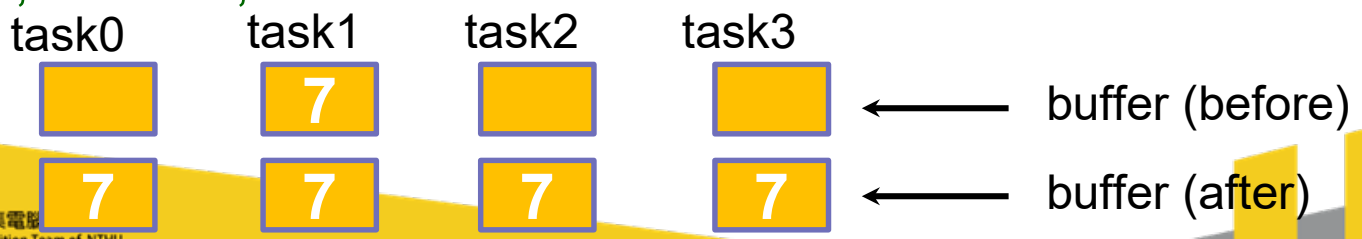
Collective Communication Routines

- MPI_Barrier (comm)
  - Creates a barrier synchronization in a group
  - Blocks until all tasks in the group reach the same MPI_Barrier call



- MPI_Bcast (&buffer, count, datatype, root, comm)
  - Broadcasts (sends) a message from the process with rank "root" to all other processes in the group

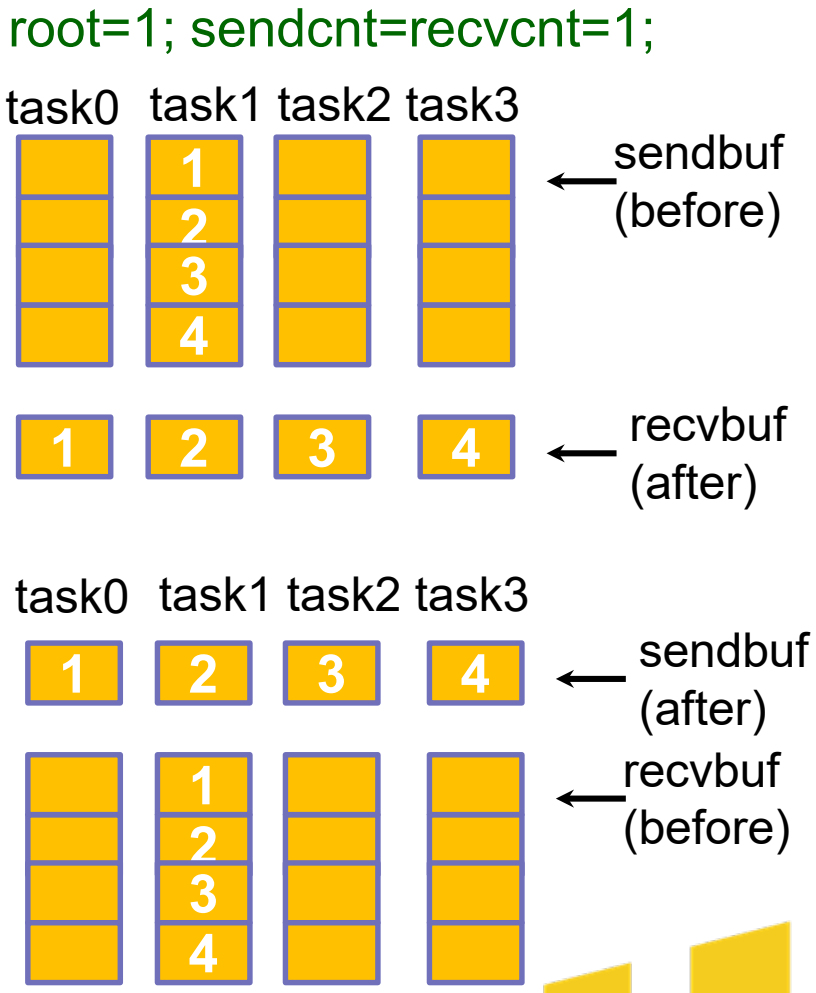
root=1; count=1;



Collective Communication Routines

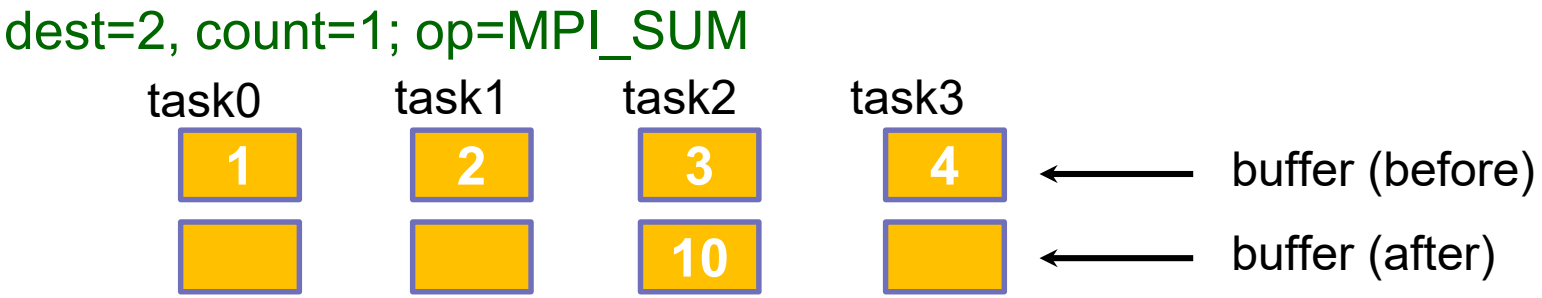
- MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)
  - Distributes distinct messages from a source task to all tasks

- MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)
  - Gathers distinct messages from each task in the group to a single destination task
  - This routine is the reverse operation of MPI_Scatter



Collective Communication Routines

- MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, dest, comm)
  - Applies a reduction operation on all tasks in the group and places the result in one task

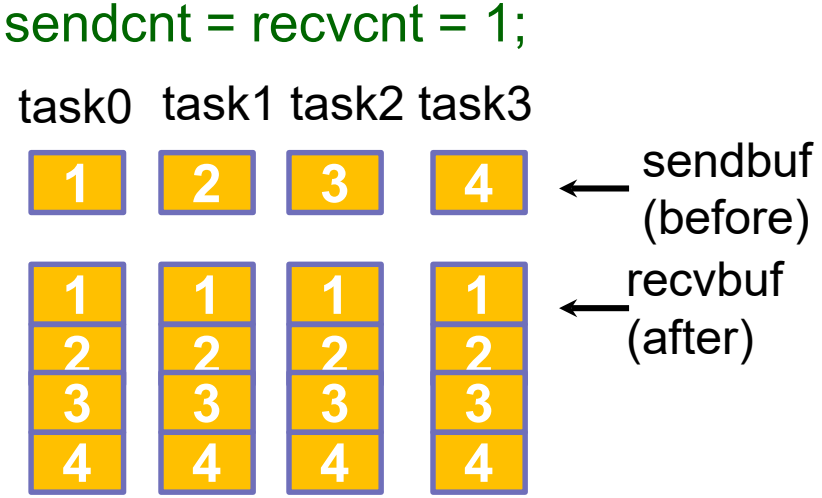


- Pre-defined Reduction Operations

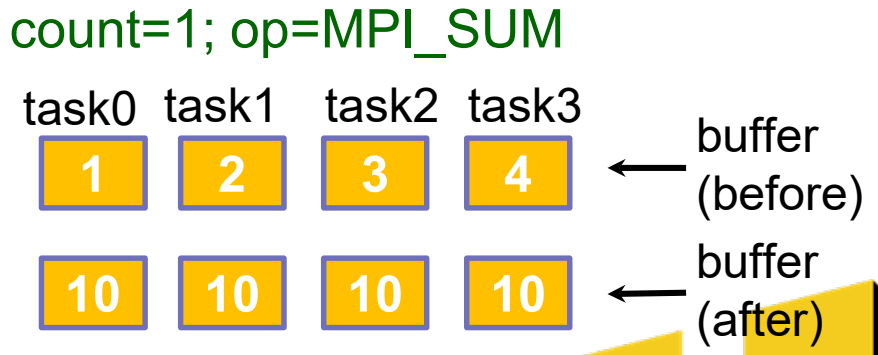
MPI_MAX	Maximum	**MPI_MIN**	Minimum
MPI_SUM	Sum	**MPI_PROD**	Product
MPI_LAND	Logical AND	**MPI_BAND**	Bit-wise AND
MPI_LOR	Logical OR	**MPI_BOR**	Bit-wise OR
MPI_LXOR	Logical XOR	**MPI_BXOR**	Bit-wise XOR

Collective Communication Routines

- MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)
  - Concatenation of data to all tasks
  - This is equivalent to an MPI_Gather followed by an MPI_Bcast



- MPI_Allreduce(&sendbuf, &recvbuf, count, datatype, op, comm)
  - Applies a reduction operation and places the result in all tasks
  - This is equivalent to an MPI_Reduce followed by an MPI_Bcast



Example: Odd-Even Sort

Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

Step 1(odd):	2	1	4	9	5	3	6	10
Step 2(even):	1	2	4	9	3	5	6	10
Step 3(odd):	1	2	4	3	9	5	6	10
Step 4(even):	1	2	3	4	5	9	6	10
Step 5(odd):	1	2	3	4	5	6	9	10
Step 6(even):	1	2	3	4	5	6	9	10
Step 7(odd):	1	2	3	4	5	6	9	10
Step 8(even):	1	2	3	4	5	6	9	10

Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

Example: Odd-Even Sort

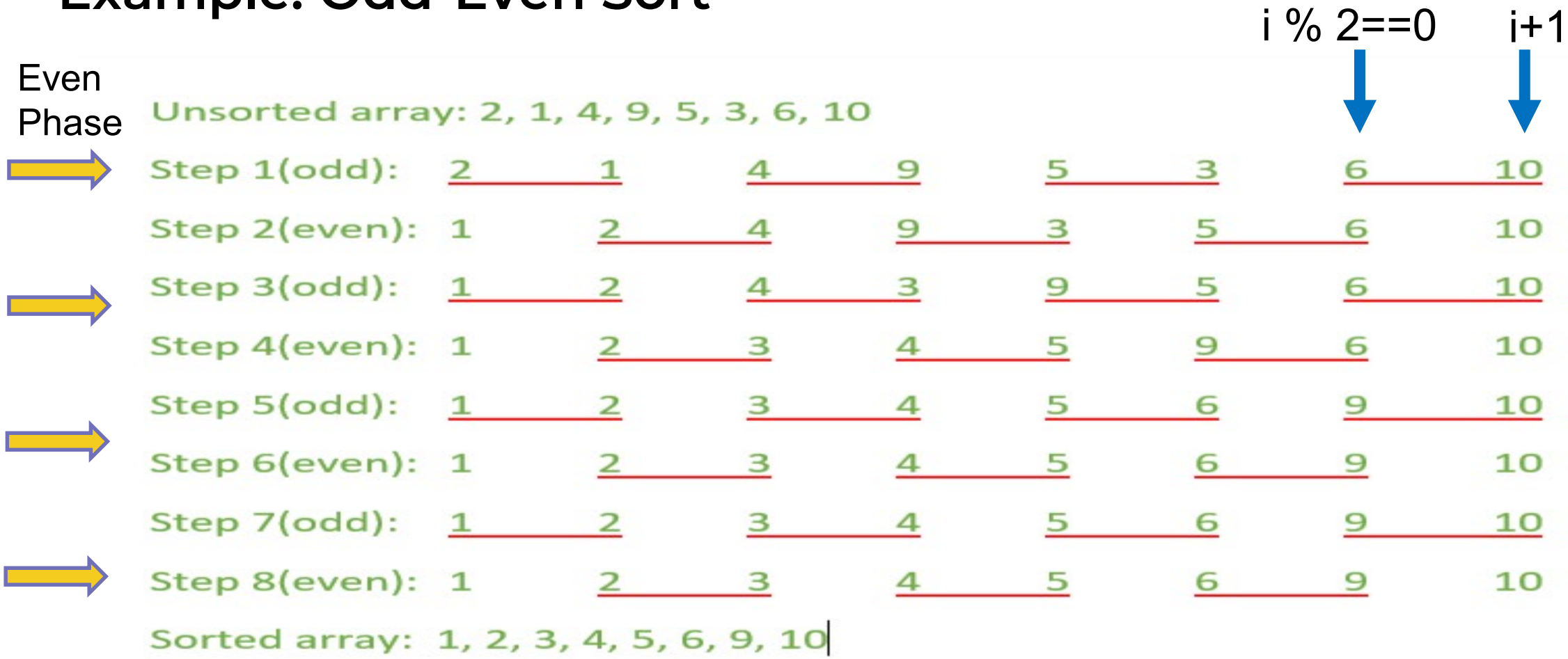
- Sequential code:

```
/* Assumes a is an array of values to be sorted. */
var sorted = false;
while(!sorted) {
    sorted=true;
    for(var i = 1; i < list.length-1; i += 2) {
        if(a[i] > a[i+1]) { swap(a, i, i+1); sorted = false; }
    }
    for(var i = 0; i < list.length-1; i += 2) {
        if(a[i] > a[i+1]) { swap(a, i, i+1); sorted = false; }
    }
}
```

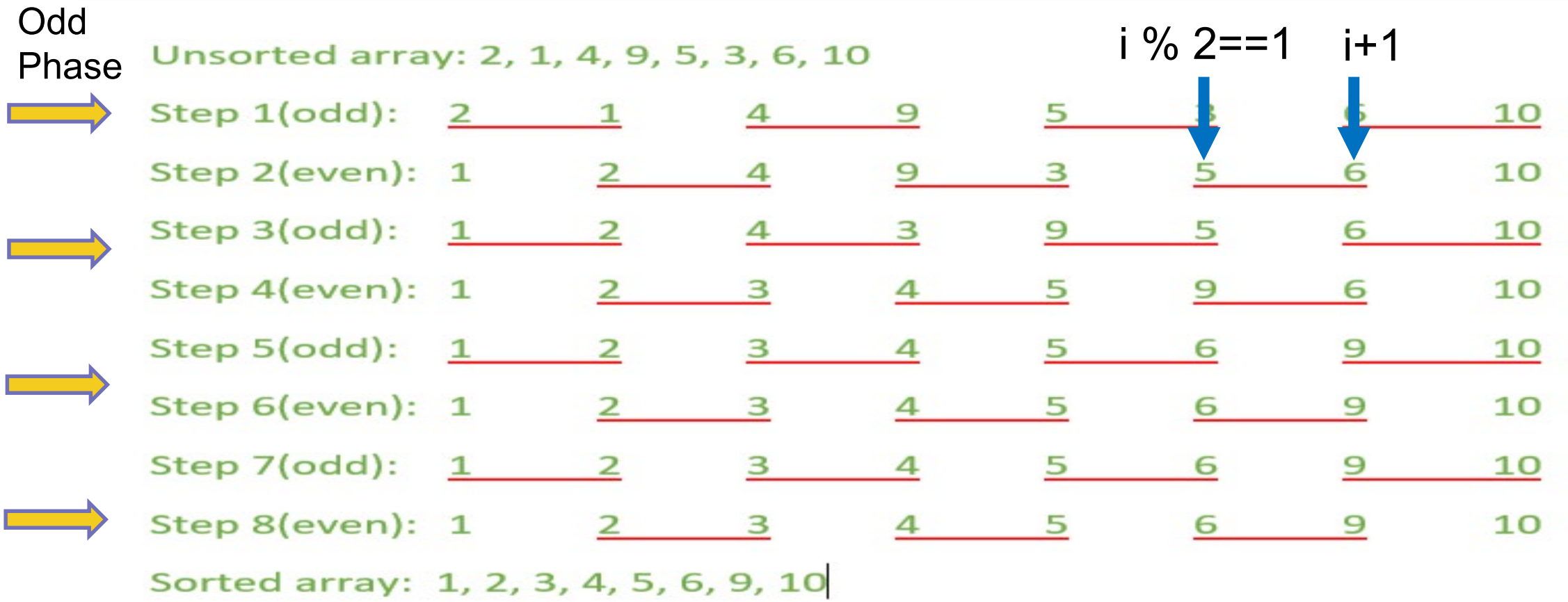
Example: Odd-Even Sort

- Parallel Code:
  1. For each process with odd rank *P*, send its number to the process with rank *P*-1.
  2. For each process with rank *P*-1, compare its number with the number sent by the process with rank *P* and send the larger one back to the process with rank *P*.
  3. For each process with even rank *Q*, send its number to the process with rank *Q*-1.
  4. For each process with rank *Q*-1, compare its number with the number sent by the process with rank *Q* and send the larger one back to the process with rank *Q*.
  5. Repeat 1-4 until the numbers are sorted.

Example: Odd-Even Sort



Example: Odd-Even Sort

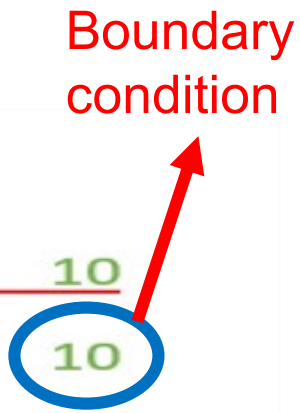


Example: Odd-Even Sort

- MPI code: Assume P_i holds a[i]

```
var sorted = false;
while(!sorted) {
      sorted=true;
      for(var iter = 1; iter < list.length-1; iter += 2) {
            if (iter % 2 == 0) {
               if (i%2 == 0) {
                        // call send & recv
                         if(a[i] > a[i+1]) { swap(a, i, i+1); sorted = false; }
               }else{
                        // call send & recv
                         if(a[i-1] > a[i]) { swap(a, i, i-1); sorted = false; }
               }
            }
            if (iter % 2 == 1) {
                        ……
            }
      }
}
```

Example: Odd-Even Sort



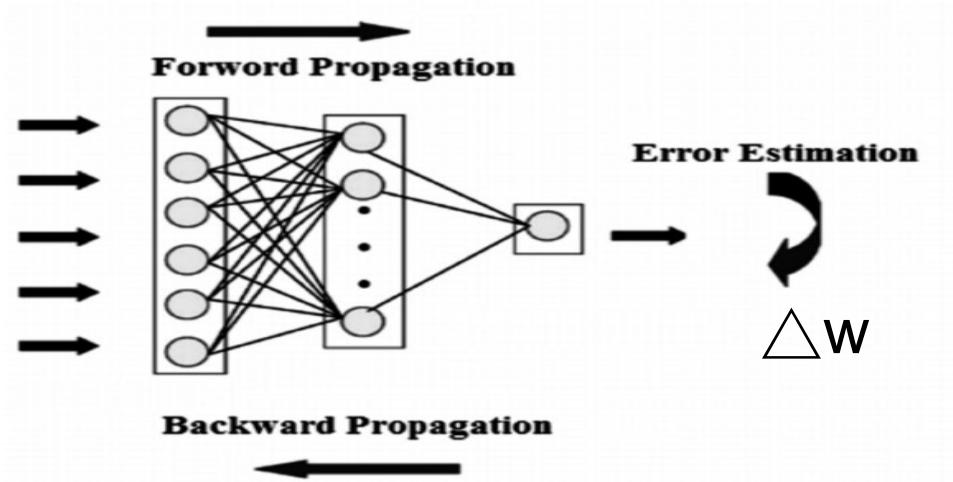
Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

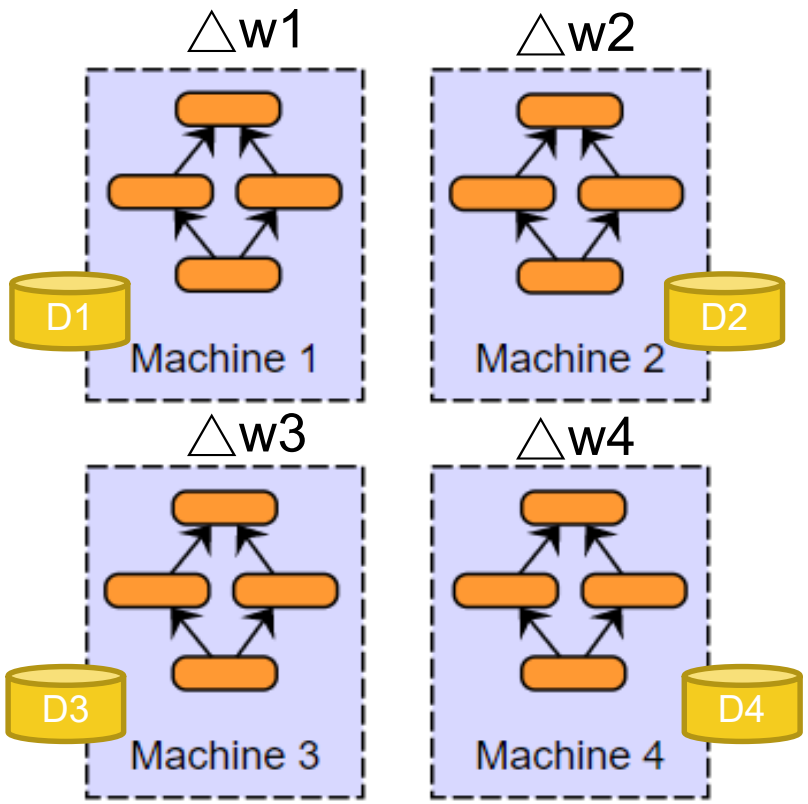
Step 1(odd):	2	1	4	9	5	3	6	10
Step 2(even):	1	2	4	9	3	5	6	10
Step 3(odd):	1	2	4	3	9	5	6	10
Step 4(even):	1	2	3	4	5	9	6	10
Step 5(odd):	1	2	3	4	5	6	9	10
Step 6(even):	1	2	3	4	5	6	9	10
Step 7(odd):	1	2	3	4	5	6	9	10
Step 8(even):	1	2	3	4	5	6	9	10

Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

Data Parallelism Model Training Revisit

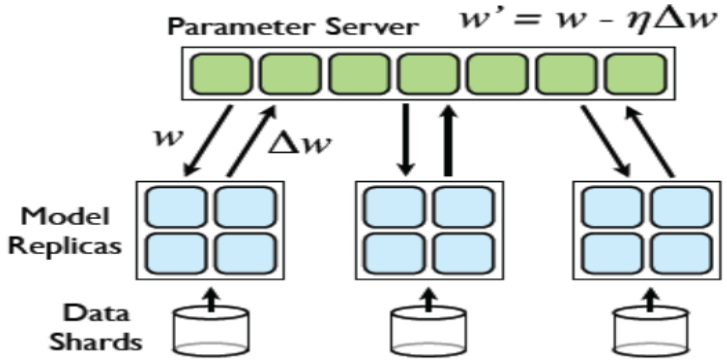
- Each process perform computation on its data independently
- Simply compute the average △w (gradient) among N processes after each iteration
  - $\triangle w = \sum \Delta w_i / N$



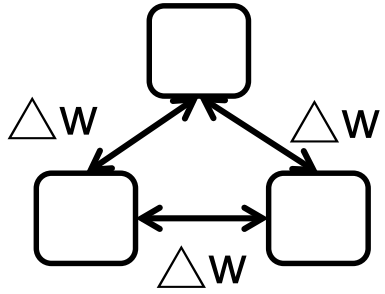


Data Parallelism Model Training Revist

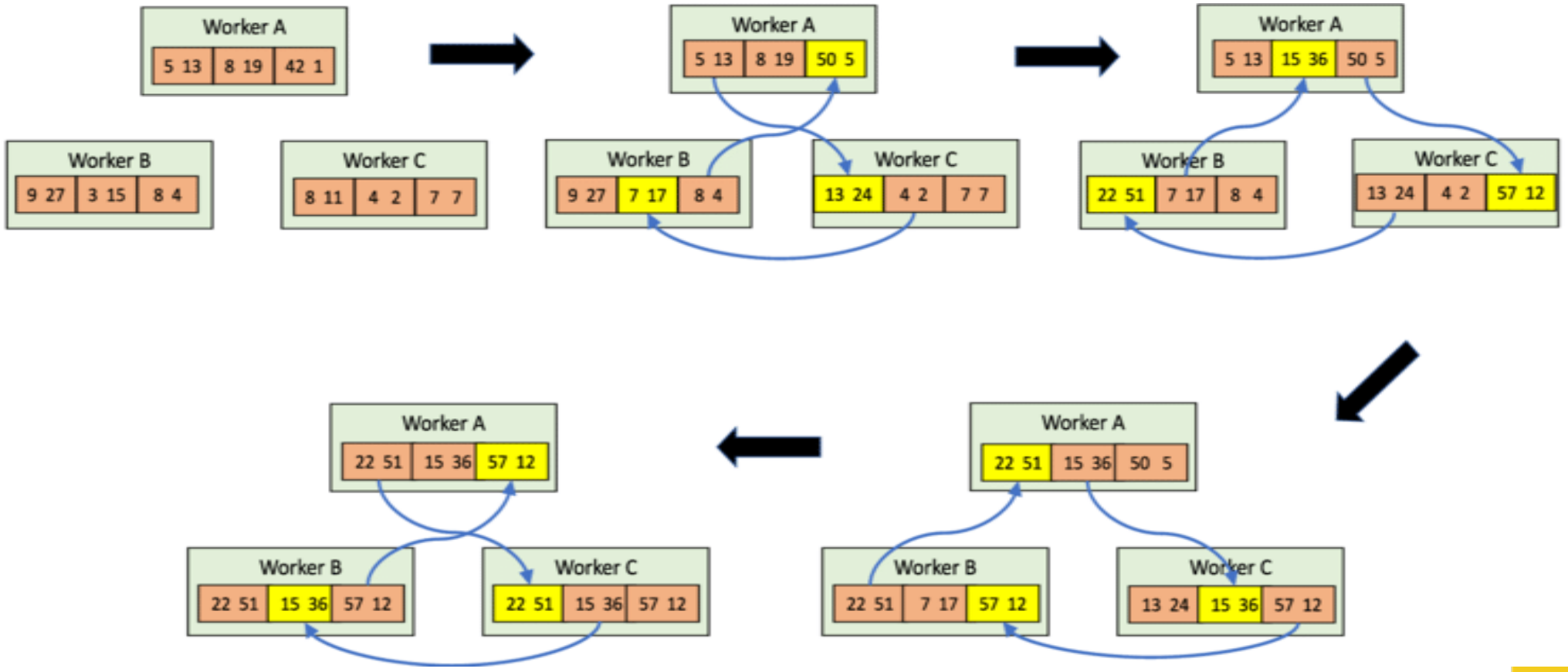
- Parameter Server (PS):
  - De-centralized across PS servers
  - Worker send gradient & receive weight
  - Support both synchronized & asynchronized SGD
  - # PS servers must be tuned
    - Too many ➔ more small messages
    - Too few ➔ network bottleneck



- AllReduce:
  - Peer to peer, fully distributed
  - Workers send gradient to each other, then compute weight by themselves
  - Balanced communication load across links
  - Need to be synchronized SGD



Horovod: Ring Allreduce



Knowledge Check

- Can race condition happen among MPI processes?
- Can data dependency problem happen among MPI processes?
- Why we have to use message passing to run program across machines or servers?
- What is collective call?
- What is non-blocking call?

Complete Course Video

- 國立清華大學開放式課程 (OCW)
- 課名: 平行程式
- 課程說明
  - 本課程將介紹平行計算的基礎觀念和電腦系統架構，並教授針對不同平行計算環境所設計的程式語言，包括多核心系統使用的Pthread、OpenMP, 叢集計算使用的MPI, GPU使用的CUDA, 以及分散式系統使用的MapReduce計算框架。修課同學必須使用 這些平行計算的語言和工具完成5個程式作業，並且以程式的執行效能結果作為學習的評量標準。