Lecture I: MPI

Lecturer : Atseng

Course Information

Join the Discord First !

All the class information call discuss in Discord

https://discord.gg/wEGVXNT7Mh

Course Contents

01 Parallel Computing Intro

02 MPI

03 Hands on!

01 Parallel Computing Intro

Why need Parallel Computing

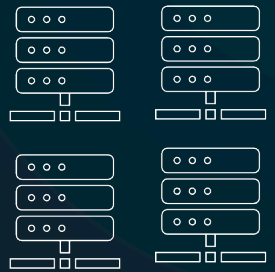
> Save time

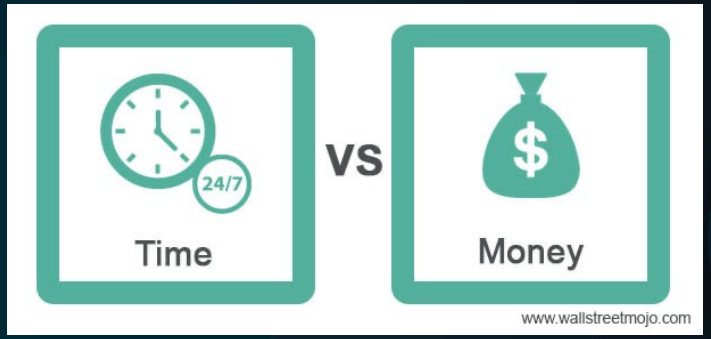
– using more resources to shorten the execution.

4 days

vs.

1 day





https://wallstreetmojo-files.s3.ap-south-1.amazonaws.com/2019/02/Time-vs-Money-differences.jpg

Why need Parallel Computing

> Solve Larger Problem

- Scientific Computing tasks
  - > scientific simulations (astrophysics, climate modeling, molecular dynamics)
    - may involve billions or even trillions of particles or elements.
    - TBs of data to be processed/analyzed

egl. Energy exploration (Italian energy company Eni deployed the HPC6)
https://www.ft.com/content/4d8f9ce3-4fb2-4560-bbf3-7ac0d3fb2872 **The link only can read once due to financial time's rule .**

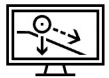
eg2. Drug Discovery and Molecular Simulations:.
https://developer.nvidia.com/blog/optimizing-drug-discovery-with-cuda-graphs-coroutines-and-gpu-workflows/

What's more about GPU application

In financial Service

Reduce Algorithm Backtesting Time

Enhance Research Signal Detection



Boost Performance and Productivity

Accelerate Risk Calculations

https://www.nvidia.com/en-us/industries/finance/ai-trading-brief/

Since GPUs are so powerful, do we still need CPUs?

Attribute	CPU	GPU
Architecture	Optimized for serial and complex tasks.	Optimized for parallel, repetitive tasks.
Application	Highly versatile (OS tasks, I/O ...)	Graphics rendering, Scientific simulations
Task Suitability	Ideal for tasks requiring complex logic, frequent branching, and low-latency responses.	Best for massively parallel tasks such as deep learning, simulations, and rendering.
Latency vs. Throughput	low-latency execution of individual tasks.	high throughput on many tasks simultaneously, even if individual task latency is higher.

Era of Parallel Computing

I. Single-Core Era

> Enabled by : Moore's Law, Voltage Scaling

> Constrained by : Power, Complexity

> Example : Assembly -> C/C++

Moore's Law: As semiconductor processes advanced, the number of transistors on a single CPU chip grew significantly, and clock speeds increased accordingly.

Voltage Scaling: Lowering voltage reduces power consumption and enables further clock speed gains.

Era of Parallel Computing

I. Single-Core Era

> Enabled by : Moore's Law, Voltage Scaling

> Constrained by : Power, Complexity

> Example : Assembly -> C/C++

Power: As clock speeds continued to rise, so did power consumption and heat generation.

Complexity: Designing faster and more complex single-core processors became increasingly difficult.

Era of Parallel Computing

2. Multi-Core Era

SMP: Chip manufacturers began placing multiple cores on the same chip (SMP, Symmetric Multi-Processing).

> Enabled by : Moore's Law, SMP

> Constrained by : Power, Parallel Software, Scalability

> Example : Pthread -> OpenMP

Era of Parallel Computing

2. Multi-Core Era

> Enabled by : Moore's Law, SMP

> Constrained by : Power, Parallel Software, Scalability

> Example : Pthread -> OpenMP

Power: Even with multiple cores, power consumption remains an issue, requiring a balance between core count and clock speed.

Parallel Software (Parallel SW) : Developers need to write or refactor applications to utilize multiple cores simultaneously.

Scalability : Developers need to write or refactor applications to utilize multiple cores simultaneously.

Era of Parallel Computing

3. Distributed System Era

> Enabled by : Networking

Networking: High-speed networks and distributed systems matured, enabling geographically dispersed machines to work together on large problems.

> Constrained by : Synchronization, Communication Overhead

> Example : MPI -> MapReduce

Era of Parallel Computing

3. Distributed System Era

> Enabled by : Networking

> Constrained by : Synchronization, Communication Overhead

> Example : MPI -> MapReduce

Synchronization: Keeping data consistent and coordinated across different nodes is difficult.

Communication Overhead: Data exchange between nodes is slower and has higher latency than intra-machine communication.

Era of Parallel Computing

4. Heterogeneous Systems Era

> Enabled by : Abundant data parallelism, more efficient GPUs

> Constrained by : Programming models, Communication Overhead

> Example : CUDA, ROCm, OpenCL

Era of Parallel Computing

4. Heterogeneous Systems Era

Abundant data Parallelism: Many applications (e.g., graphics processing, deep learning) require processing large amounts of data in parallel.

> Enabled by : Abundant data parallelism, more efficient GPUs

> Constrained by : Programming models, Communication Overhead

> Example : CUDA, ROCm, OpenCL

Heterogeneouts System（異質系統）指的是由不同類型的計算單元（如 CPU、CPU、FPGA、ASIC、TPU）組成的計算平台，以提高效能、能效和專門運算能力。這些異質運算單元協同工作，針對不同類型的計算任務發揮各自的優勢。

Era of Parallel Computing

4. Heterogeneous Systems Era

> Enabled by : Abundant data parallelism, more efficient GPUs

> Constrained by : Programming models, Communication Overhead

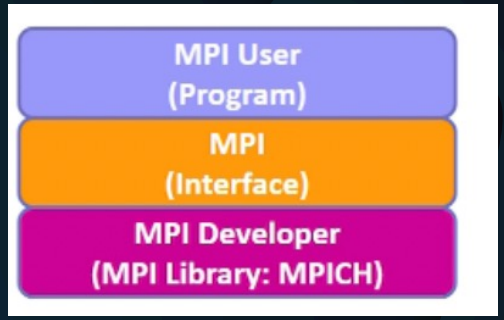
> Example : CUDA, ROCm, OpenCL

Programming models : New APIs and frameworks (e.g., CUDA, ROCm, OpenCL) are needed to efficiently leverage heterogeneous systems with both CPUs and GPUs.

Communication Overhead: Data transfers and synchronization between CPU and GPU can introduce significant overhead.

02 Messaging Passing Interface

Messaging Passing Interface

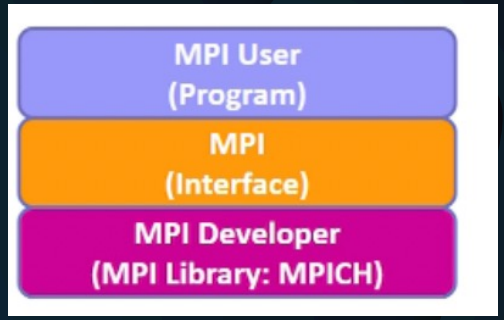


MPI = Messaging Passing Interface

A specification for the developers and users of message passing libraries

By itself, it is an interface NOT a library

Messaging Passing Interface

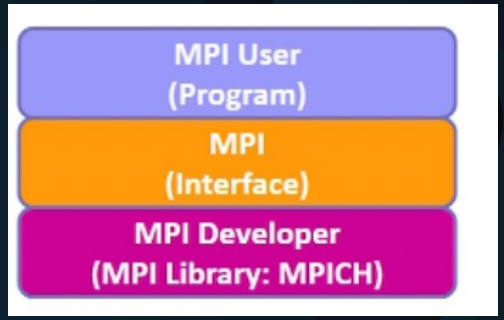


MPI = Messaging Passing Interface

Goal :

- Portable: Run on different machines or platforms .
- Scalable: Run on million of compute nodes .
- Flexible: Isolate MPI developers from MPI programmers (users)

Messaging Passing Interface



MPI = Messaging Passing Interface

MPI user : Writes MPI-based applications. (program).

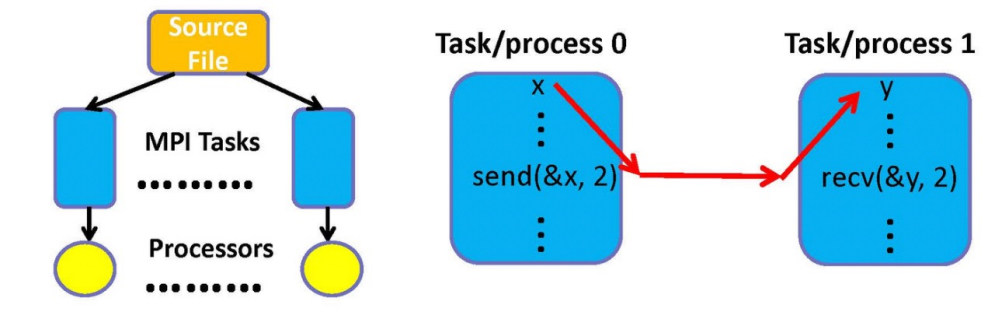
MPI Interface : Provides a standardized API for program execution.

MPI Developer : Develops the MPI libraries such as MPICH and OpenMPI.

Programming Model

Single Program Multiple Data

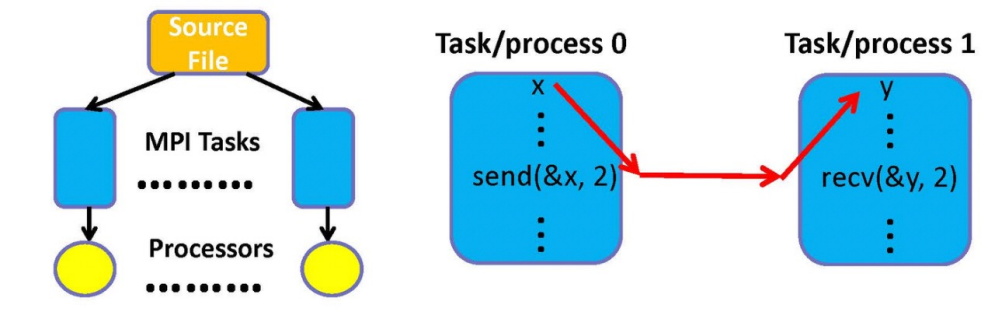
> Allow tasks to branch or conditionally execute only parts of the program they are designed to execute.



Programming Model

Distributed memory

> MPI provides a method of sending & receiving messages



Communication Method

Synchronous communication --- sending and receiving data occurs simultaneously.

Asynchronous communication --- sending and receiving data occurs non-simultaneously.

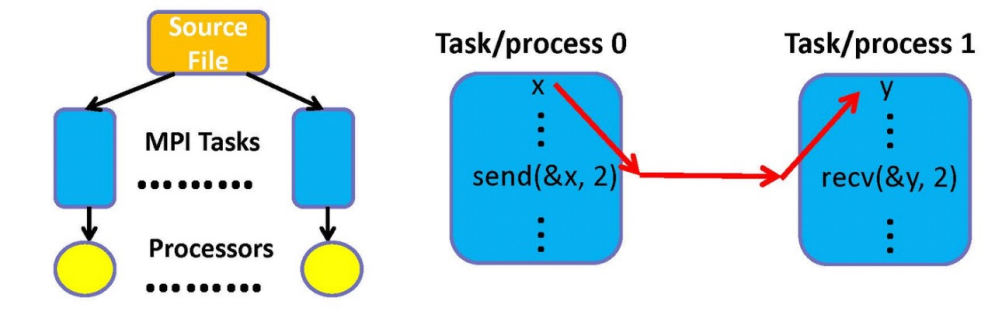
Blocking --- has been used to describe routines that do not return until the transfer is completed.

Non-blocking --- has been used to describe routines that return whether or not the message has been received.

Programming Model

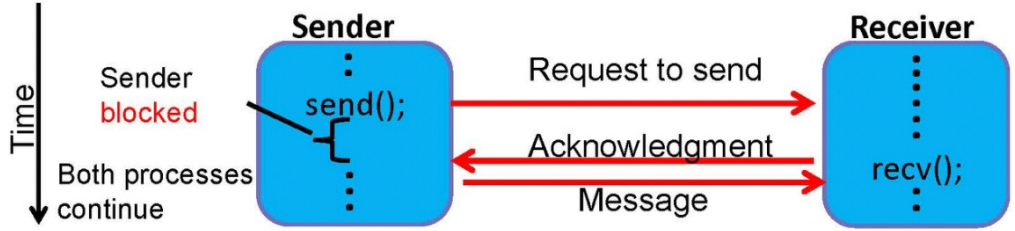
Distributed memory

> MPI provides a method of sending & receiving messages

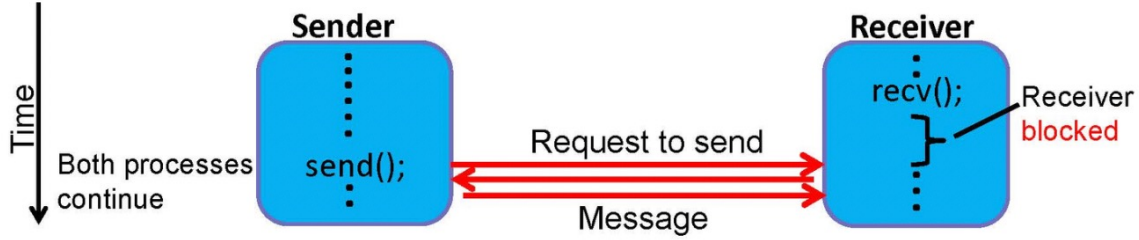


Synchronous/Blocking Messaging Passing

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



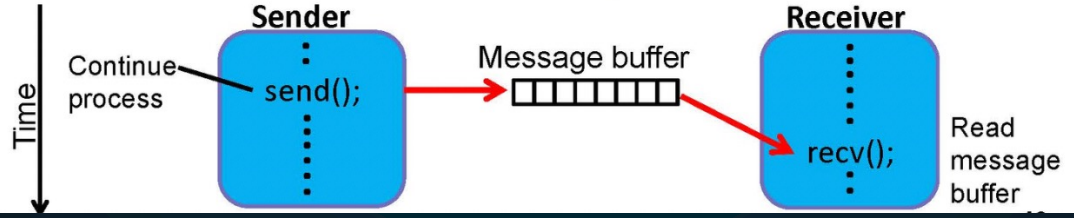
- **Receiver**: wait until the message it is expecting arrives



https://www.youtube.com/watch?v=iF-oQIznTGY&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=15

ASynchronous/Non-Blocking Messaging Passing

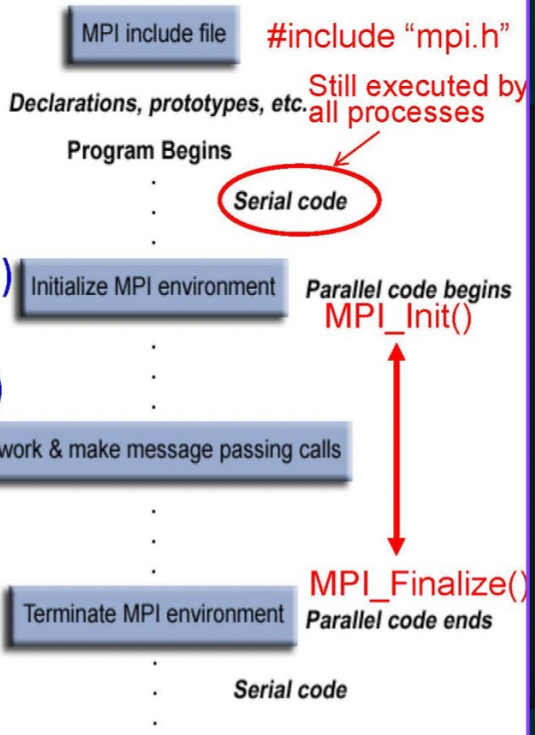
- How message-passing routines can return before the message transfer has been completed?
  - Generally, a ***message buffer*** needed between source and destination to hold message
  - Message buffer is a memory space at the sender and/or receiver side
  - For send routine, once the local actions have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



https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

Getting Start

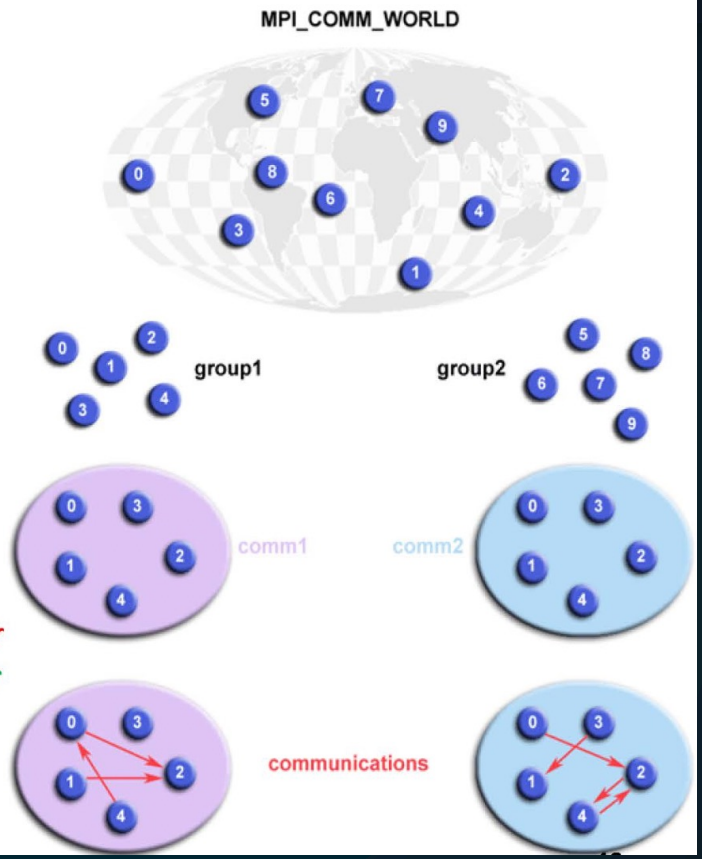
- Header file: **"mpi.h"**
  - Required for all programs that make MPI library call
- MPI calls:
  - Format: rc = MPI_Xxx(parameter, ...)
  - Example: rc = MPI_Bcast (&buffer,count,datatype,root,comm)
  - Error code: return as "rc"; rc=MPI_SUCCESS if successful
- General MPI program structure:



https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

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



https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

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)** rank_id = task_id
  - Determines the rank of the calling process within the communicator
  - This rank is often referred to as a task ID

https://www.youtube.com/watch?v=9XMobnlnDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

```
#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 ();
}
```

https://www.youtube.com/watch?v=9XMobnlnDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

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

buffer 大小 = type*count(長度)

- comm: indicates the communication context

comm 預設 COMM_WORLD

- 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

https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

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);
}
```

https://www.youtube.com/watch?v=9XMobnlnDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

Non-Blocking Example

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

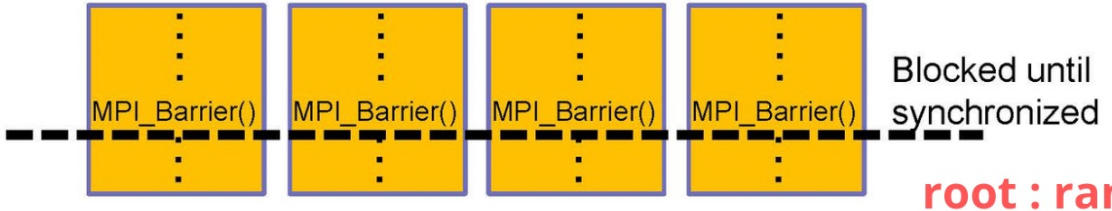
```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {
   int x=10;
   MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req1);
   compute();
} else if (myrank == 1) {
   int x;
   MPI_Irecv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, req1);
}
MPI_Wait(req1, status);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

- MPI_Wait() **blocks** until the operation has actually completed

https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

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

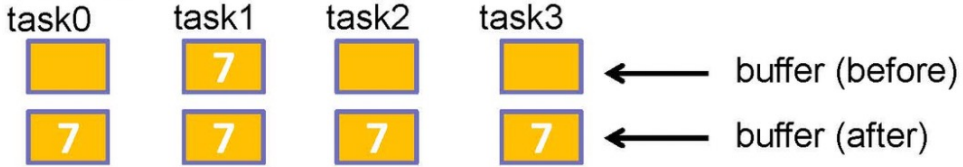


root : rank ID

- 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: 呼叫broadcast的process

root=1; count=1;



21

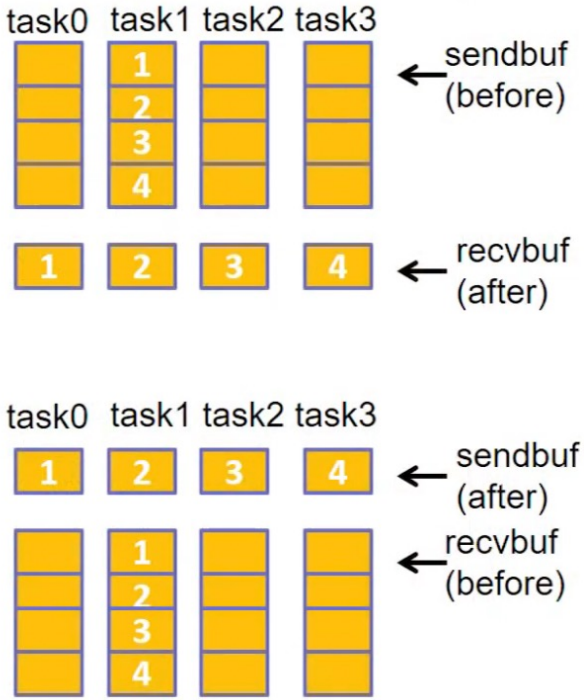
https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

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

sendcnt=recvcnt 是決定要送幾個值

root=1; sendcnt=recvcnt=1;



https://www.youtube.com/watch?v=9XMobnInDkg&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=14

03 Hands-on time !

```
mpicc -o hello_world hello_world.c

mpirun -np 4 ./hello_world
```

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the name of the processor

    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    MPI_Finalize(); // Finalize the MPI environment
    return 0;
}
```

```
mpicc -o hello_world hello_world.c

mpirun -np 4 ./hello_world
```

```
atseng@TsengdeMacBook-Pro GPUtest % mpicc -o hello_world hello_world.c

atseng@TsengdeMacBook-Pro GPUtest % mpirun -np 4 ./hello_world

Hello world from processor TsengdeMacBook-Pro.local, rank 2 out of 4 processors
Hello world from processor TsengdeMacBook-Pro.local, rank 0 out of 4 processors
Hello world from processor TsengdeMacBook-Pro.local, rank 1 out of 4 processors
Hello world from processor TsengdeMacBook-Pro.local, rank 3 out of 4 processors
```

It’s your turn!

Course Reference

https://www.youtube.com/watch?v=5IjXq_fcsM4&list=PLSOSUwlYe8cxqw7OUHOE5n4Lm-mXFXbZT&index=5

https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html