中国科学院大学
University of Chinese Academy of Sciences

Introduction to MPI

Part*1. MPI Basics*

人工智能学院
缪青海
miaoqh@ucas.ac.cn

Content

- **The Message-Passing Model**
- What is MPI
- Work with MPICH
- Write a simple program in MPI

Sample Parallel Programming Models

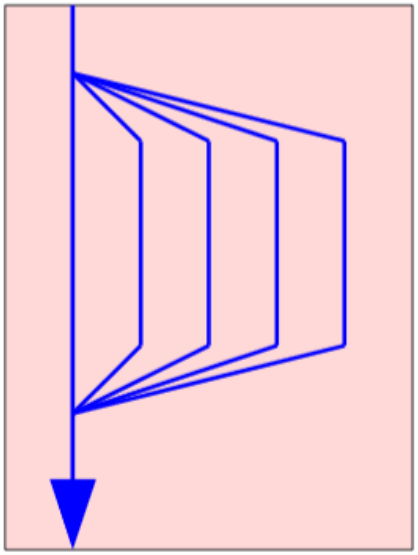
- Shared Memory Programming Model
  - Transparent Parallelization
    - Compiler works magic on sequential programs
  - Directive-based Parallelization
    - Compiler needs help (e.g., OpenMP)

- **Message Passing Model**
  - Explicit communication between processes (like sending and receiving emails)

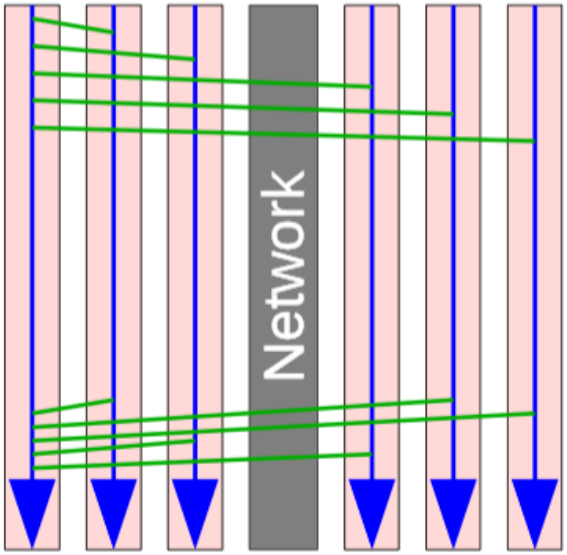
Memory models

Shared Memory

Distributed Memory

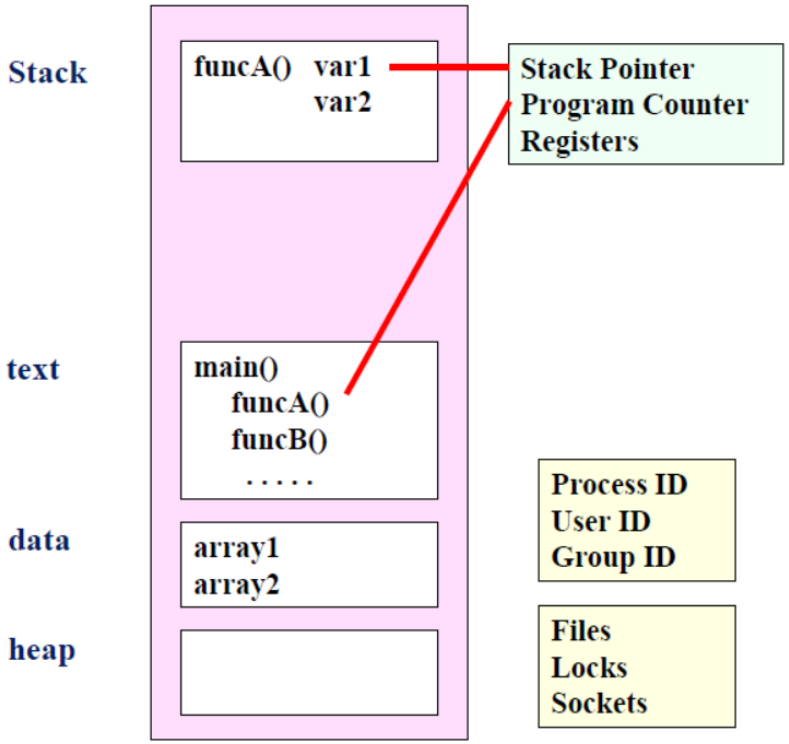


one process,
many threads

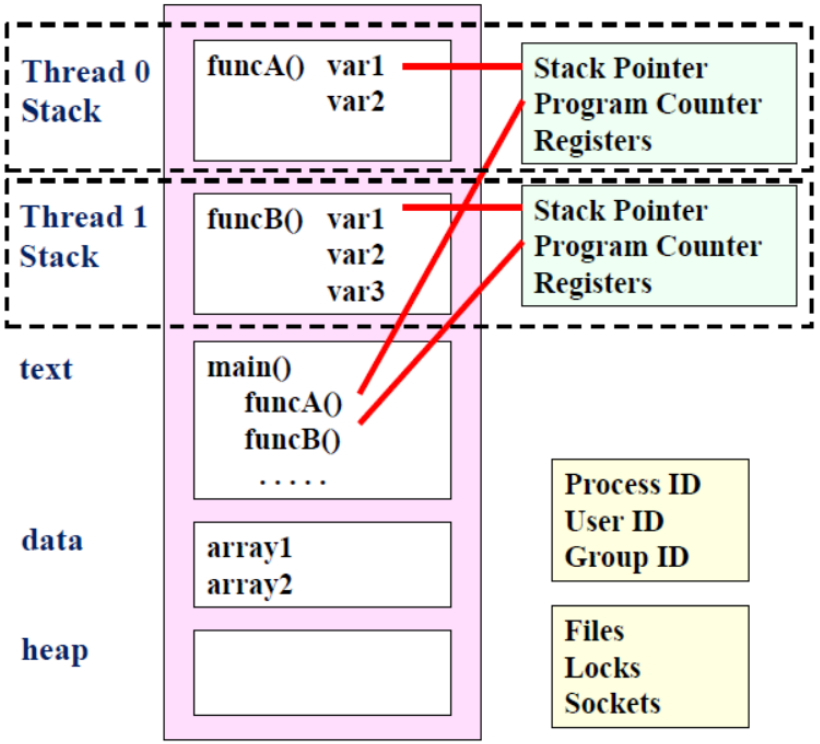


multiple processes,
multiple nodes

Process & Threads



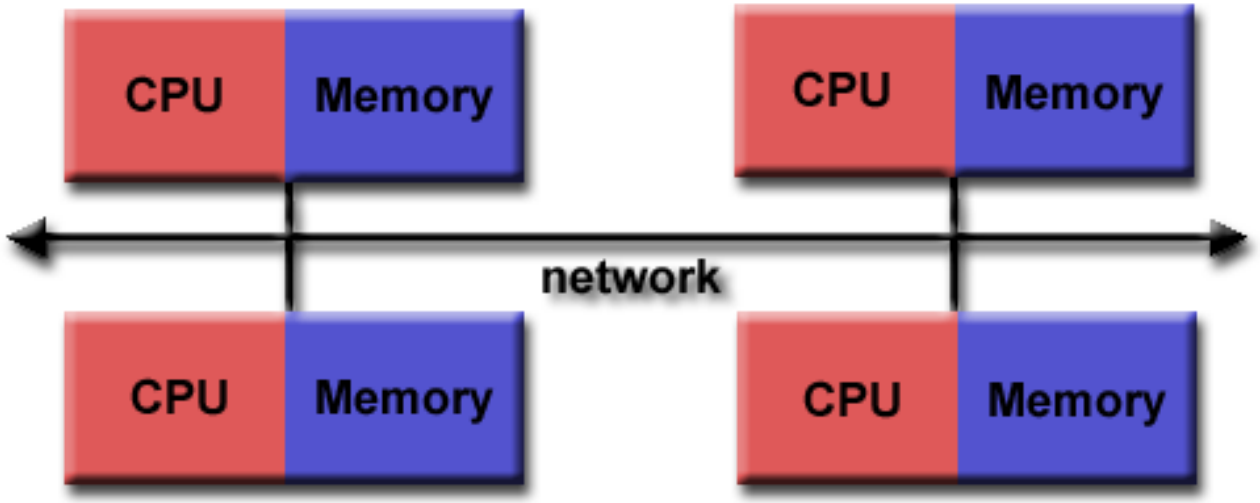
Process



Threads

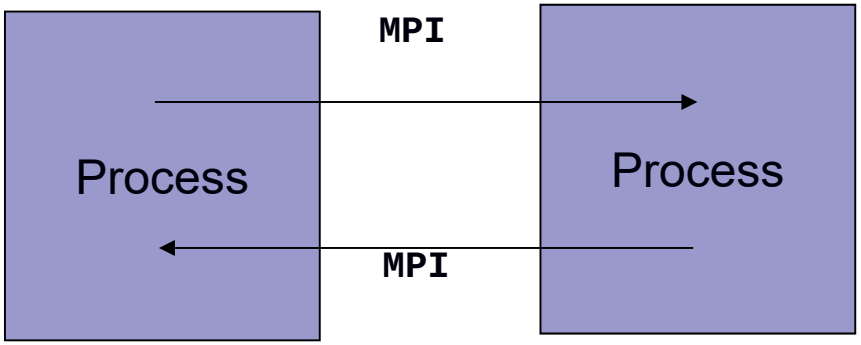
The Message-Passing Model

- **Distributed Memory: 分布式内存**
  - 分布式内存架构需要通讯网络（*communication network*）链接各个CPU专属内存。



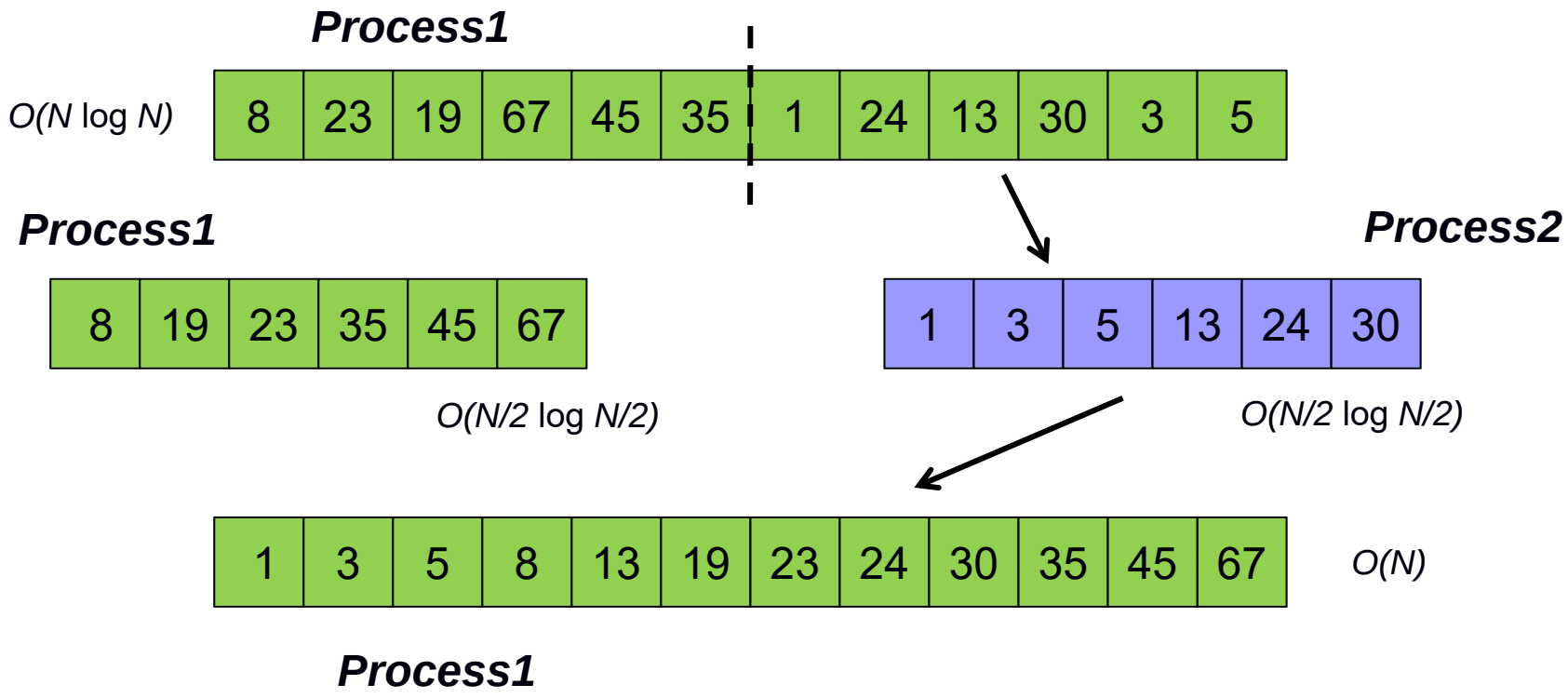
The Message-Passing Model

- **Inter-process communication consists of**
  - synchronization
  - movement of data from one process's address space to another's.



The Message-Passing Model

- Each process has to send/receive data to/from other processes. **Example: Sorting Integers**



Standardizing with MPI

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were **not portable** (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The **MPI Forum** was a collection of vendors, portability writers and users that wanted to **standardize** all these efforts.

Content

- The Message-Passing Model
- **What is MPI**
- Work with MPICH
- Write a simple program in MPI

What is MPI?

- **MPI**: **M**essage **P**assing **I**nterface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers

What is MPI?

- MPI: Message Passing Interface
  - Incorporates the best ideas in a "**standard**" way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match

What is MPI?

- MPI is **not**…
  - a language or compiler specification;
  - a specific implementation or product;

MPI-1

- MPI-1 was defined (1994) by a broadly based group:
  - parallel computer vendors
  - computer scientists
  - applications developers.
- MPI-1 supports the classical message-passing programming model:
  - basic point-to-point communication,
  - collectives,
  - datatypes, etc

MPI-1

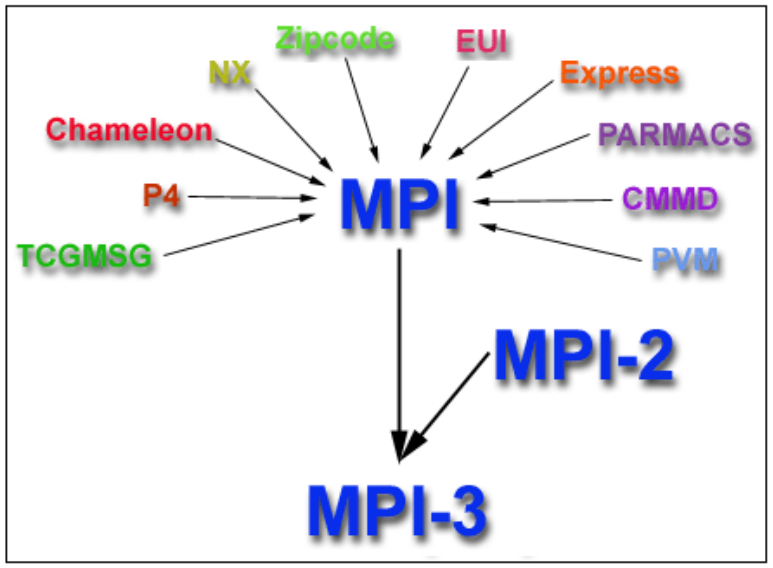
- Implementations:
  - appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments:
  - **MPICH**
  - **OpenMPI**

MPI-2

- MPI-2 was released in 1997
  - MPI-2.1 (2008) and MPI-2.2 (2009)
  - Extends the message-passing model.
    - Parallel I/O
    - Remote memory operations (one-sided)
    - Dynamic process management
  - Adds other functionality
    - C++ and Fortran 90 bindings
    - External interfaces
    - Language interoperability
    - MPI interaction with threads

MPI-3

- MPI-3 (2012) added several new features to MPI
- Major new features
  - Nonblocking collectives
  - Neighborhood collectives
  - Improved one-sided communication interface
  - Tools interface
  - Fortran 2008 bindings



MPI-3

- Other new features
  - Matching Probe and Recv for thread-safe probe and receive
  - Noncollective communicator creation function
  - “const” correct C bindings
  - Comm_split_type function
  - Nonblocking Comm_dup
  - Type_create_hindexed_block function
- C++ bindings removed

MPI-3

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of material including tutorials, a FAQ, other MPI pages
- MPI-4
  - 2020~2021

Reasons for Using MPI

- **Standardization**
  - - MPI is the only standard.
  - It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability**
  - - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

Reasons for Using MPI

- **Performance Opportunities**
  - - Vendor implementations should be able to exploit native hardware features to optimize performance
- **Functionality**
  - – Rich set of features
- **Availability**
  - - A variety of implementations are available, both vendor and public domain
  - Mpich → Intel MPI

Important considerations

- When using MPI, all parallelism is explicit.

- The programmer is responsible for correctly:
  - **identifying** parallelism;
  - **implementing** parallel algorithms;
  - using MPI constructs.

Content

- The Message-Passing Model
- What is MPI?
- **Work with MPICH**
- Write a simple program in MPI

Status of MPI-3 Implementations

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI [1]	IBM PE MPICH [2]	IBM Platform	SGI MPI	Fujitsu MPI	Microsoft MPI
NB collectives	✔	✔	✔	✔	✔	✔	✔	Q4 '14	✔	✔		
Neighborhood collectives	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15			
RMA	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15			
Shared memory	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15			
Tools Interface	✔	✔	✔				(✔)	Q4 '14	Q3 '15			
Non-collective comm. create	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15	✔		
F08 Bindings	✔		✔				✔	Q4 '14	Q3 '15			
New Datatypes	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15	✔		
Large Counts	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15			
Matched Probe	✔	✔	✔	✔	✔	✔	✔	Q4 '14	Q3 '15	✔		

Release dates are estimates and are subject to change at any time.

Empty cells indicate no *publicly announced* plan to implement/support that feature.

What is MPICH

- MPICH is a high-performance and widely portable open-source **implementation of MPI**.
- It provides **all features of MPI** that have been defined so far (including MPI-1, MPI-2.0, MPI-2.1, MPI-2.2, and MPI-3.0)
- MPICH is a popular **open-source and free** implementation of MPI.
  - Vendors and other collaborators take MPICH and add support for their systems:
  - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH, MPICH-MX

What is MPICH

- Active development lead by:
  - **Argonne National Laboratory**
  - **University of Illinois at Urbana-Champaign**
  - Several close collaborators who contribute many features, bug fixes, testing for quality assurance, etc.
    - IBM, Microsoft, Cray, Intel
    - Ohio State University, Queen's University
    - and many others
- Current release is MPICH-3.4

Getting Started with MPICH

- Download MPICH
  - Go to http://www.mpich.org and downloads link
  - The download will be a zipped tarball
- Build MPICH
  - Unzip/untar the tarball: **`tar -xzvf mpich-3.2.1.tar.gz`**
  - **`cd mpich-3.2.1`**
  - **`./configure –-prefix=/where/to/install/mpich |& tee c.log`**
  - **`make |& tee m.log`**
  - **`make install |& tee mi.log`**
  - **`Add /where/to/install/mpich/bin to your PATH`**
- 或者在Ubuntu下：
  - sudo apt install mpich

Compiling MPI programs with MPICH

- You can just assume that "mpicc" and friends have replaced your regular compilers (gcc, gfortran, etc.)
- Compilation Wrappers
  - For C programs: `mpicc test.c –o test`
  - For C++ programs: `mpicxx test.cpp –o test`
- You can link other libraries are required too
  - To link to a math library:
  - `mpicc test.c –o test –lm`

Running MPI programs with MPICH

- Launch 16 processes on the local node:
  - `mpiexec -n 16 ./test`

Running MPI programs with MPICH

- Launch 16 processes on 4 nodes (each has 4 cores)
  - **`mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test`**
    - Runs the first four processes on h1, the next four on h2, etc.
  - **`mpiexec -hosts h1,h2,h3,h4 -n 16 ./test`**
    - Runs the first process on h1, the second on h2, etc., and wraps around
    - So, h1 will have the 1st, 5th, 9th and 13th processes

Running MPI programs with MPICH

- If there are many nodes, it might be easier to create a host file:
  - **`cat hf`**

    `h1:4`

    `h2:2`
  - **`mpiexec -hostfile hf -n 16 ./test`**

Trying some example programs

- MPICH comes packaged with several example programs using almost ALL of MPICH's functionality

- A simple program to try out is the **PI example** written in C (cpi.c)
  - – calculates the value of PI in parallel (available in the examples directory when you build MPICH)
  - **`mpiexec -n 16 ./examples/cpi`**

Trying some example programs

- The output will show how many processes are running, and the error in calculating PI
- Next, try it with multiple hosts
  - **`mpiexec –hosts h1:2,h2:4 –n 16 ./examples/cpi`**

Interaction with Resource Managers

- **On clusters**
  - Resource managers such as SGE, PBS, SLURM or Loadleveler;
  - MPICH automatically detects them and interoperates with them

- For example with PBS, you can create a script such as:

```
#! /bin/bash
cd $PBS_O_WORKDIR
# No need to provide –np or –hostfile options
mpiexec ./test
```

Interaction with Resource Managers

- Job can be submitted as:
  - **qsub –l nodes=2:ppn=2 test.sub**
  - "mpiexec" will automatically know that the system has PBS, and ask PBS for the number of cores allocated (4 in this case), and which nodes have been allocated
- The usage is similar for other resource managers

Debugging MPI programs

- Parallel debugging is trickier than debugging serial programs
  - Many processes computing;
  - getting the state of one failed process is usually hard.

- MPICH provides in-built support for debugging
  - It natively interoperates with commercial parallel debuggers such as
    - Totalview
    - DDT

Debugging MPI programs

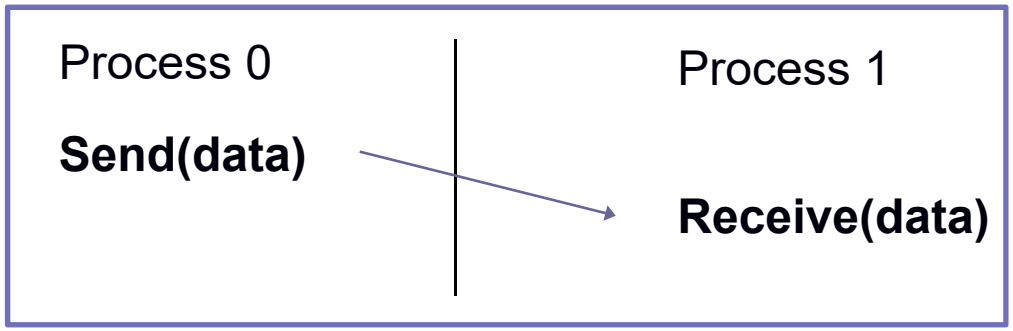
- Using MPICH with **Totalview**:
  - `totalview -a mpiexec -n 6 ./test`
- Using MPICH with **ddd (or gdb)** on one process:
  - `mpiexec -n 4 ./test : -n 1 ddd ./test : -n 1 ./test`
  - Launches the 5th process under "ddd" and all other processes normally

Content

- The Message-Passing Model
- What is MPI?
- Work with MPICH
- **Write a simple program in MPI**

MPI Basic Send/Receive

- Simple communication model



- Application needs to specify to the MPI implementation:
  1. How do you compile and run an MPI application?
  2. **How will processes be identified?**
  3. **How will "data" be described?**

Process Identification

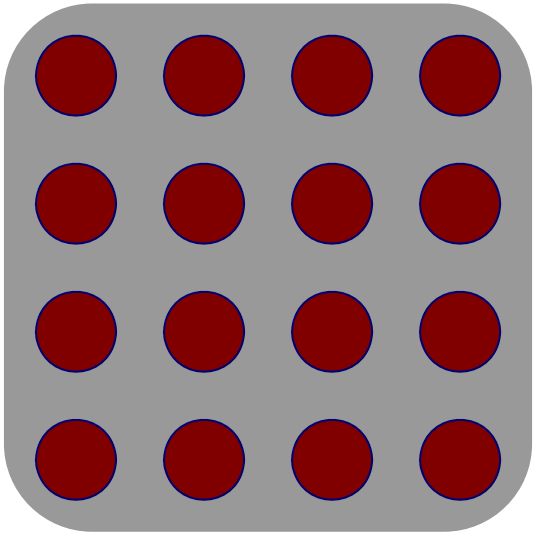
- MPI processes can be collected into groups
  - Each group can have multiple colors (some times called context)
  - *Group + color == communicator (it is like a name for the group)*
  - When an MPI application starts, the group of all processes is initially given a predefined name called **MPI_COMM_WORLD**
    - The same group can have many names, but simple programs do not have to worry about multiple names

Process Identification

- A process is identified by **a unique number** within each communicator, called ***rank***
  - Rank is local to communicator.
  - For two different communicators, the same process can have two different ranks;
  - So the meaning of a "rank" is only defined when you specify the communicator.

Communicators

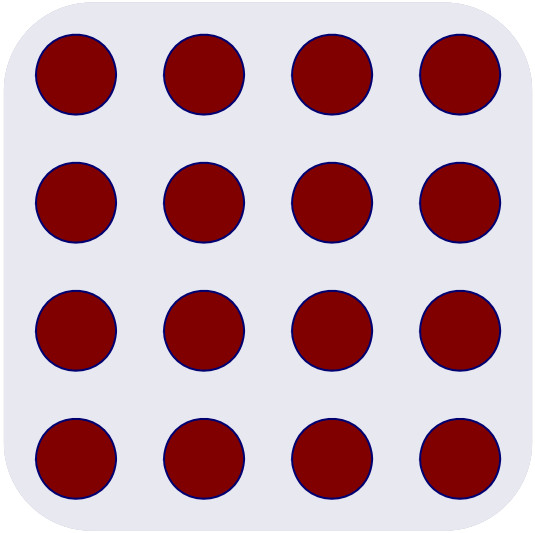
```
mpiexec  -n  16  ./test
```



When you start an MPI program, there is one predefined communicator
MPI_COMM_WORLD

Communicators

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



When you start an MPI program, there is one predefined communicator
MPI_COMM_WORLD

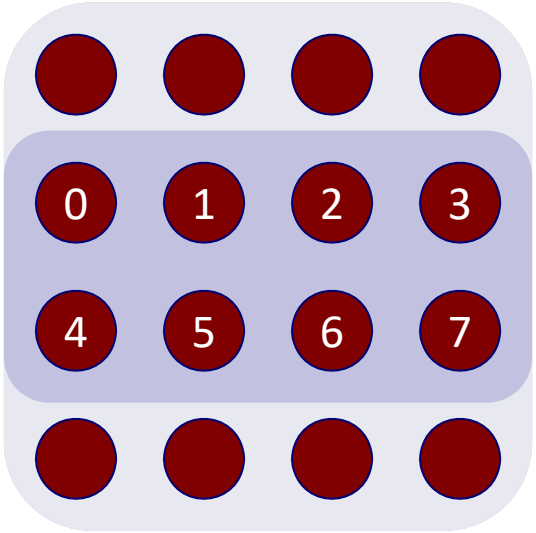
Can make copies of this communicator (same group of processes, but different "aliases")

Communicators

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

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as “rank”



When you start an MPI program, there is one predefined communicator
`MPI_COMM_WORLD`

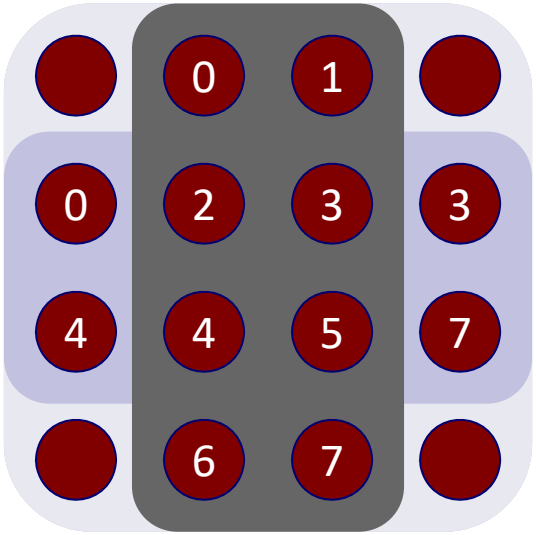
Can make copies of this communicator (same group of processes, but different “aliases”)

Communicators

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

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as “rank”



When you start an MPI program, there is one predefined communicator **MPI_COMM_WORLD**

Can make copies of this communicator (same group of processes, but different “aliases”)

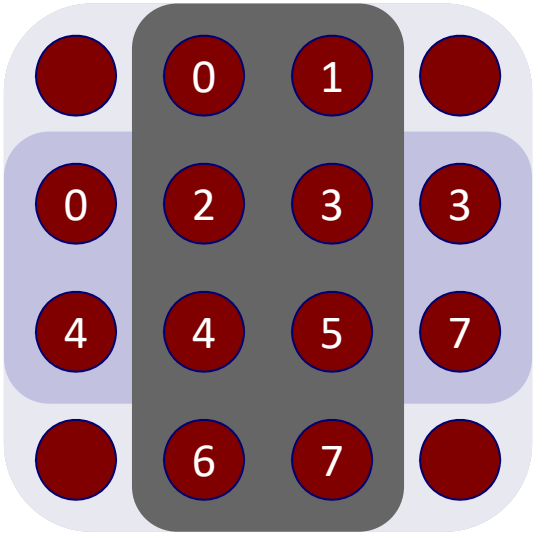
The same process might have different ranks in different communicators

Communicators

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

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as “rank”



When you start an MPI program, there is one predefined communicator
`MPI_COMM_WORLD`

Can make copies of this communicator (same group of processes, but different “aliases”)

The same process might have different ranks in different communicators

Communicators can be created “by hand” or using tools provided by MPI (not discussed in this tutorial)

Simple programs typically only use the predefined communicator
`MPI_COMM_WORLD`

Simple MPI Program

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

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Finalize();
    return 0;
}
```

Basic requirements for an MPI program

Simple MPI Program

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

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank + 1, size);

    MPI_Finalize();
    return 0;
}
```

Basic requirements for an MPI program

Code Example - 1

- *code/test1/mpi_hello.c*
  - *mpicc mpi_hello.c –o hello*
  - *mpiexec –n 2 ./hello*

Data Communication

- Data communication in MPI is:
  - like **email exchange**
  - One process sends a copy of the data to another process (or a group of processes);
  - and the other process receives it.
- Communication requires the following:
  - Information that **Sender** has to know.
  - Information that **Receiver** "might" have to know.

Data Communication

- Information that **Sender** has to know :
  - Whom to send the data to
    - receiver's process rank
  - What kind of data to send
    - 100 integers or 200 characters, etc
  - A user-defined "tag" for the message
    - think of it as an email subject;
    - allows the receiver to understand what type of data is being received

Data Communication

- **Receiver** "might" have to know :
  - Who is sending the data
    - OK if the receiver does not know;
    - in this case sender rank will be **MPI_ANY_SOURCE**, meaning anyone can send
  - What kind of data is being received
    - partial information is OK: I might receive up to 1000 integers
  - What the user-defined "tag" of the message is
    - OK if the receiver does not know
    - in this case tag will be **MPI_ANY_TAG**

More Details on Describing Data

- MPI Datatype is very similar to a C or Fortran datatype
  - `int` → `MPI_INT`
  - `double` → `MPI_DOUBLE`
  - `char` → `MPI_CHAR`
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises of other datatypes → (a char, an int and a double).
  - Or, a vector datatype for the columns of a matrix
- The "count" in **`MPI_SEND`** and **`MPI_RECV`** refers to how many datatype elements should be communicated

MPI Basic (Blocking) Send

```
MPI_SEND(buf, count, datatype,
         dest, tag, comm)
```

- The message buffer is described by (**`buf, count, datatype`**).
- The target process is specified by **`dest`** and **`comm`**.
  - **`dest`** is the rank of the target process in the communicator specified by **`comm`**.
- **`tag`** is a user-defined "type" for the message

- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Basic (Blocking) Receive

```
MPI_RECV(buf, count, datatype,
         source, tag, comm, status)
```

- Waits until a matching (on **source, tag, comm**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator **comm**, or **MPI_ANY_SOURCE**.
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.

MPI Basic (Blocking) Receive

```
MPI_RECV(buf, count, datatype,
         source, tag, comm, status)
```

- `status` contains further information:
  - Who sent the message (can be used if you used **`MPI_ANY_SOURCE`**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **`MPI_ANY_TAG`**)
  - **`MPI_STATUS_IGNORE`** can be used if we don't need any additional information

Simple Communication in MPI

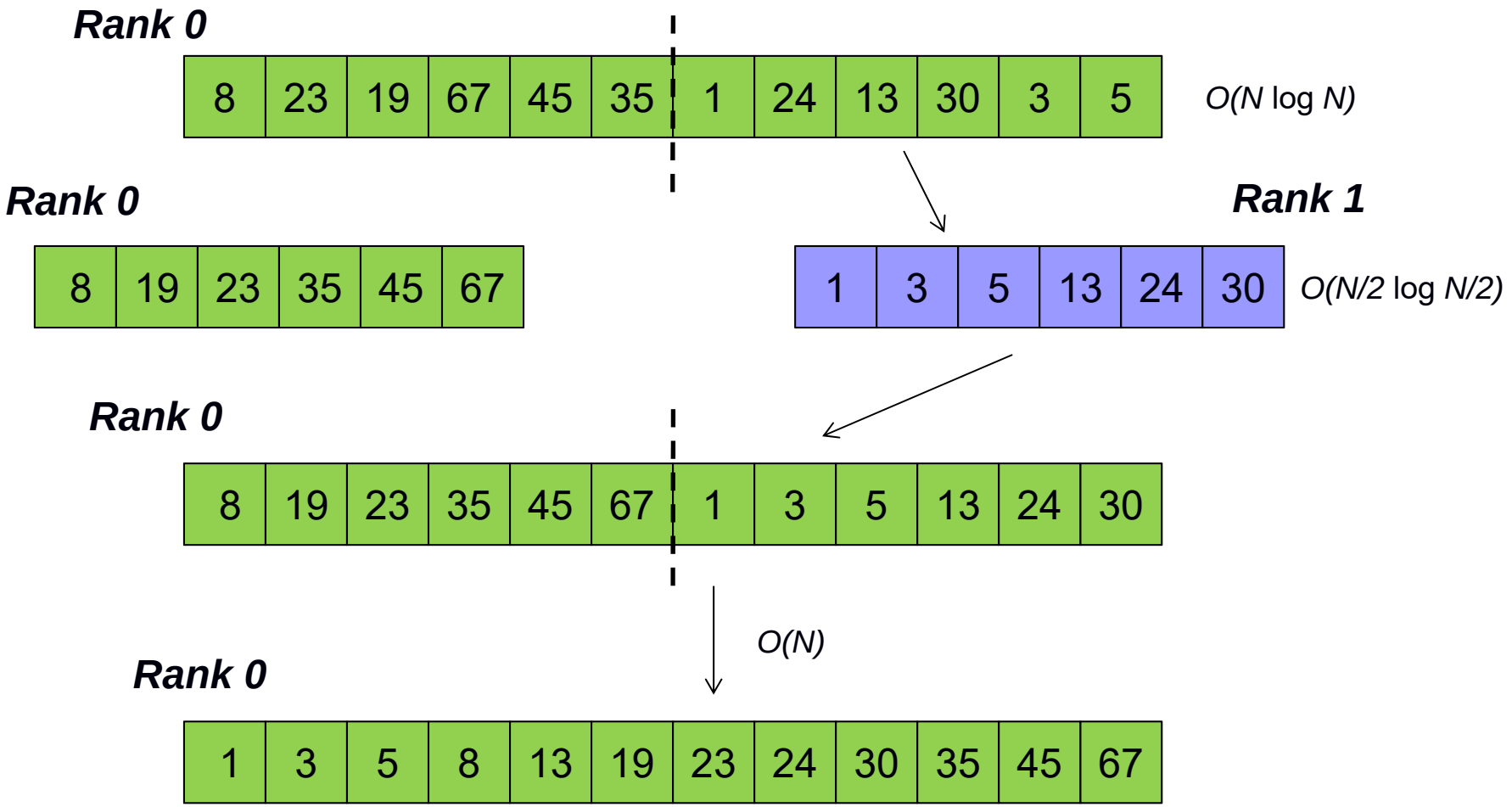
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

Code Example - 1

- *Code/test1/mpi_send_recv.c*
- *Or (mpi_helloBsend.c)*

  - *mpicc mpi_send_recv.c –o test*
  - *mpiexec –n 2 ./test*

Parallel Sort using MPI Send/Recv



Parallel Sort using MPI Send/Recv

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank, a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD,
                                    MPI_STATUS_IGNORE);

        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD,
                                    MPI_STATUS_IGNORE);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

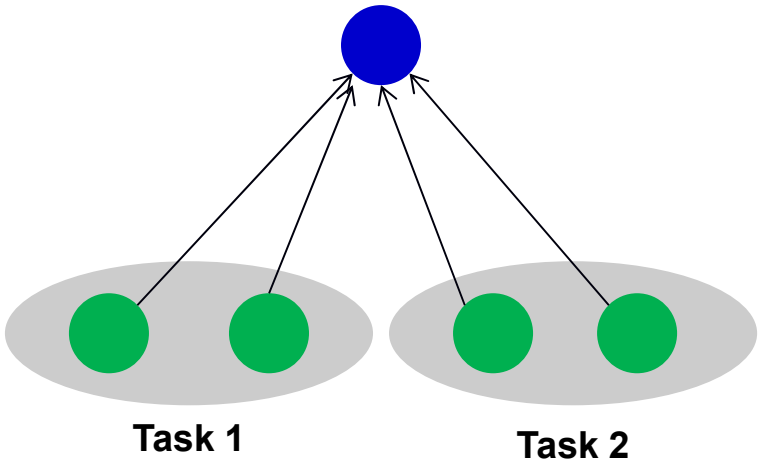
Status Object

- The status object is used after completion of a receive to find:
  - The source process for the message (**`status.MPI_SOURCE`**)
  - The message tag (**`status.MPI_TAG`**)
  - Error status (**`status.MPI_ERROR`**)
- The number of elements received is given by:

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

`status`	return status of receive operation (status)
`datatype`	datatype of each receive buffer element (handle)
`count`	number of received elements (integer)(OUT)

Using the "status" field



- Each "worker process" → "master" process together with its group number:
  - the "tag" field can be used to represent the task
  - Data count is not fixed (maximum 100 elements)
  - Order in which workers send output to master is not fixed (different workers = different source ranks, and different tasks = different tags)

Using the “status” field (contd.)

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

int main(int argc, char ** argv)
{
    [...snip...]

    if (rank != 0)  /* worker process */
        MPI_Send(data, rand() % 100, MPI_INT, 0, group_id,
                 MPI_COMM_WORLD);
    else {  /* master process */
        for (i = 0; i < size – 1; i++) {
            MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE,
                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            printf(“worker ID: %d; task ID: %d; count: %d\n”,
                   status.MPI_SOURCE, status.MPI_TAG, count);
        }
    }

    [...snip...]
}
```

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_INIT – initialize the MPI library (must be the first routine called)**
  - **MPI_COMM_SIZE - get the size of a communicator**
  - **MPI_COMM_RANK – get the rank of the calling process in the communicator**
  - **MPI_SEND – send a message to another process**
  - **MPI_RECV – send a message to another process**
  - **MPI_FINALIZE – clean up all MPI state (must be the last MPI function called by a process)**
- For performance, however, you need to use other MPI features

Web Pointers

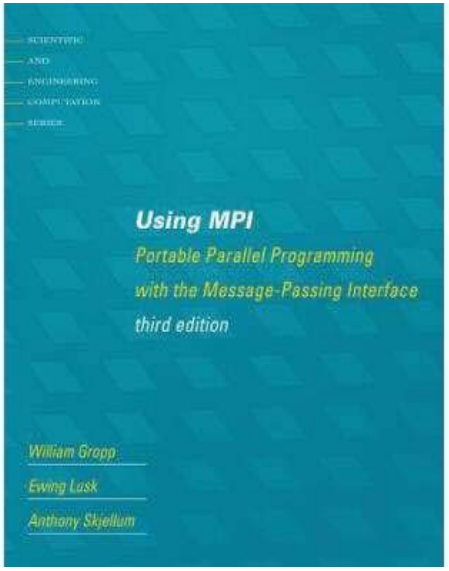
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPICH : http://www.mpich.org
- MPICH mailing list: discuss@mpich.org
- MPI Forum : http://www.mpi-forum.org/

- Other MPI implementations:
  - MVAPICH (MPICH on InfiniBand) : http://mvapich.cse.ohio-state.edu/
  - Intel MPI (MPICH derivative): http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI (MPICH derivative)
  - Open MPI : http://www.open-mpi.org/

- Several MPI tutorials can be found on the web

books

- Using MPI and Using Advanced MPI



references

- Parallel Programming with MPI, Argonne National Laboratory, http://www.anl.gov/events/parallel-programming-mpi.
- A Comprehensive MPI Tutorial Resource, https://github.com/wesleykendall/mpitutorial.

- 张武生等，MPI并行程序设计实例教程，清华大学出版社。
- 迟学斌等，并行计算与实现技术，科学出版社。