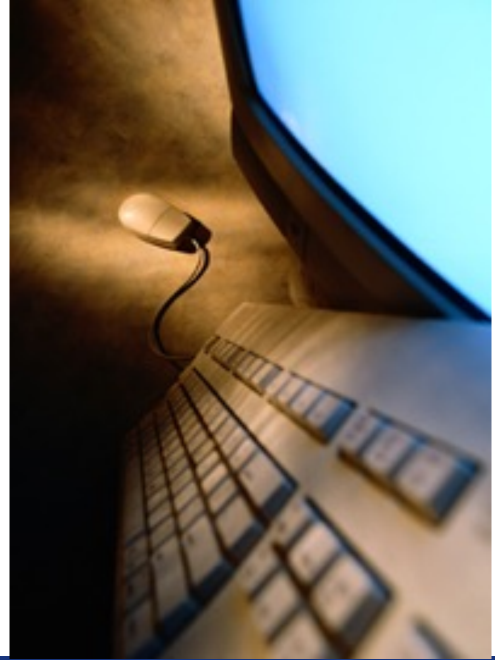
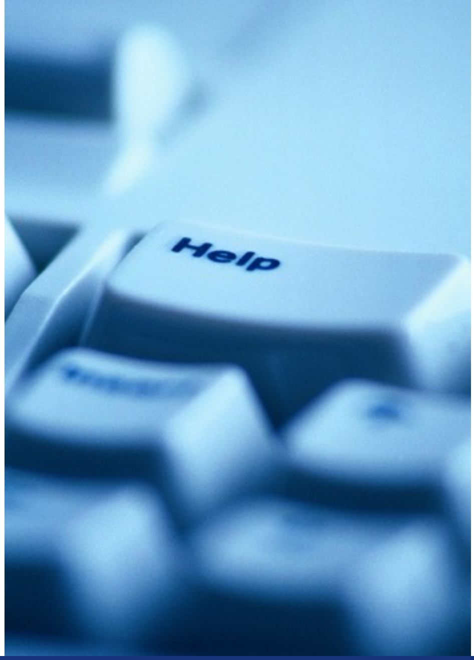




并行与分布式计算
Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
2020-06-12

Lecture 8 — Programming with MPI

Pengfei Chen

School of Data and Computer Science

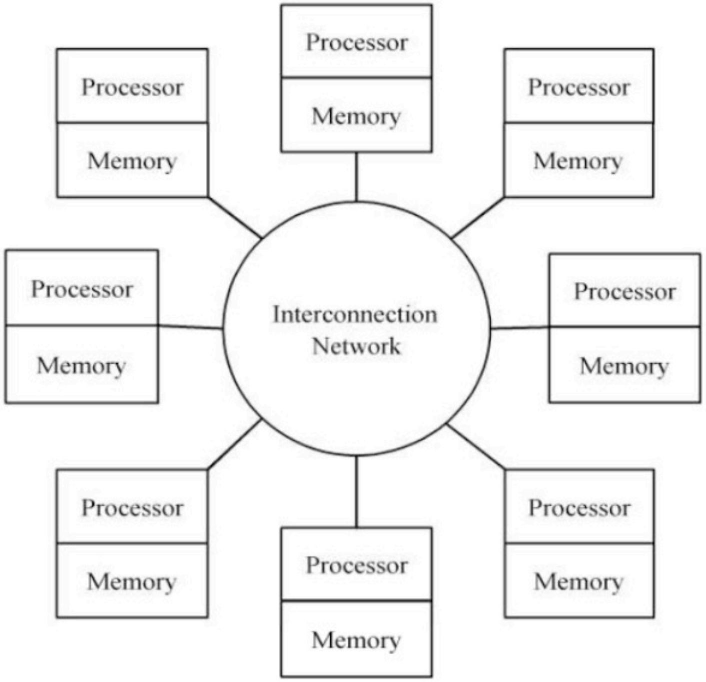
Jun 12, 2020

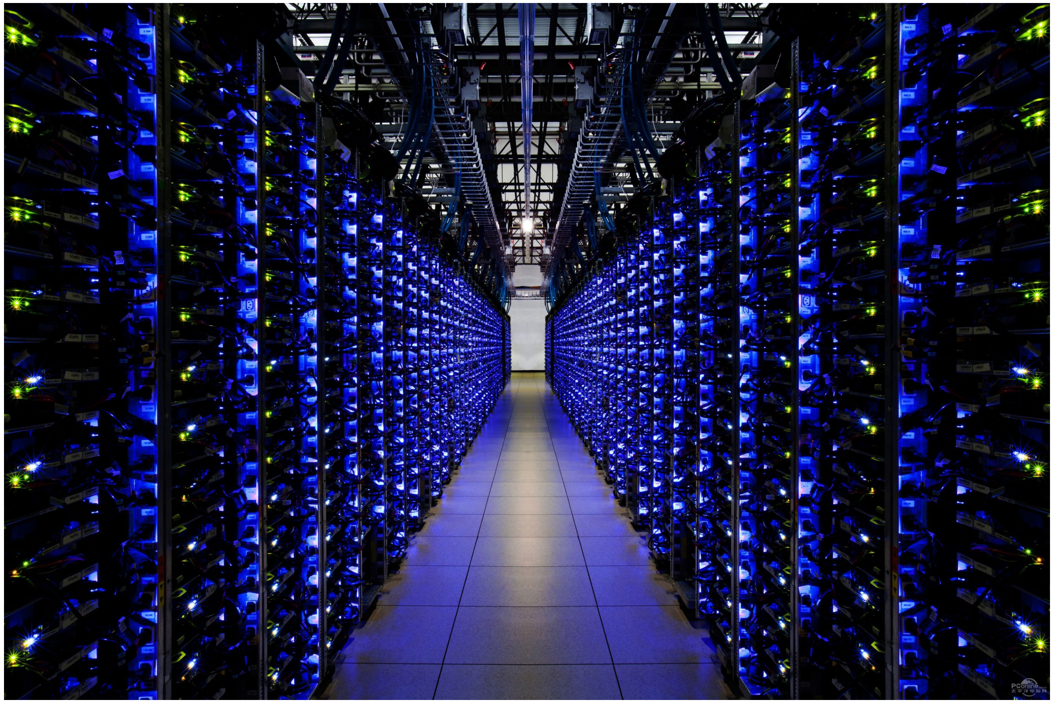
Outline:

- **Principles of Message-Passing Programming**
- **The Building Blocks: Send and Receive Operations**
- **MPI: the Message Passing Interface**
- **Overlapping Communication with Computation (Key)**
- **Groups and Communicators**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Message-Passing Model





Principles of Message-Passing Programming

- **The logical view of a machine supporting the message passing paradigm consists of *p* processes, each with its own exclusive address space**
- **CONSTRAINTS （限制）…**
  - **Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed**
  - **All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data**
- **These two constraints, while onerous (繁重), make underlying costs very explicit to the programmer**

Principles of Message-Passing Programming

- **Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms**
  - **In the asynchronous paradigm, all concurrent tasks execute asynchronously**
  - **In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously**
- **Most message-passing programs are written using the *single program multiple data* (SPMD) model**

The Building Blocks: Send and Receive Operations

- **The prototypes of these operations are as follows:**

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- **Consider the following code segments:**

```
P0                          P1
a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a = 0;
```

  - **The semantics of the send operation require that the value received by process P1 must be *100* as opposed to *0***

- **This motivates the design of the send and receive protocols**

Send and Receive Operations

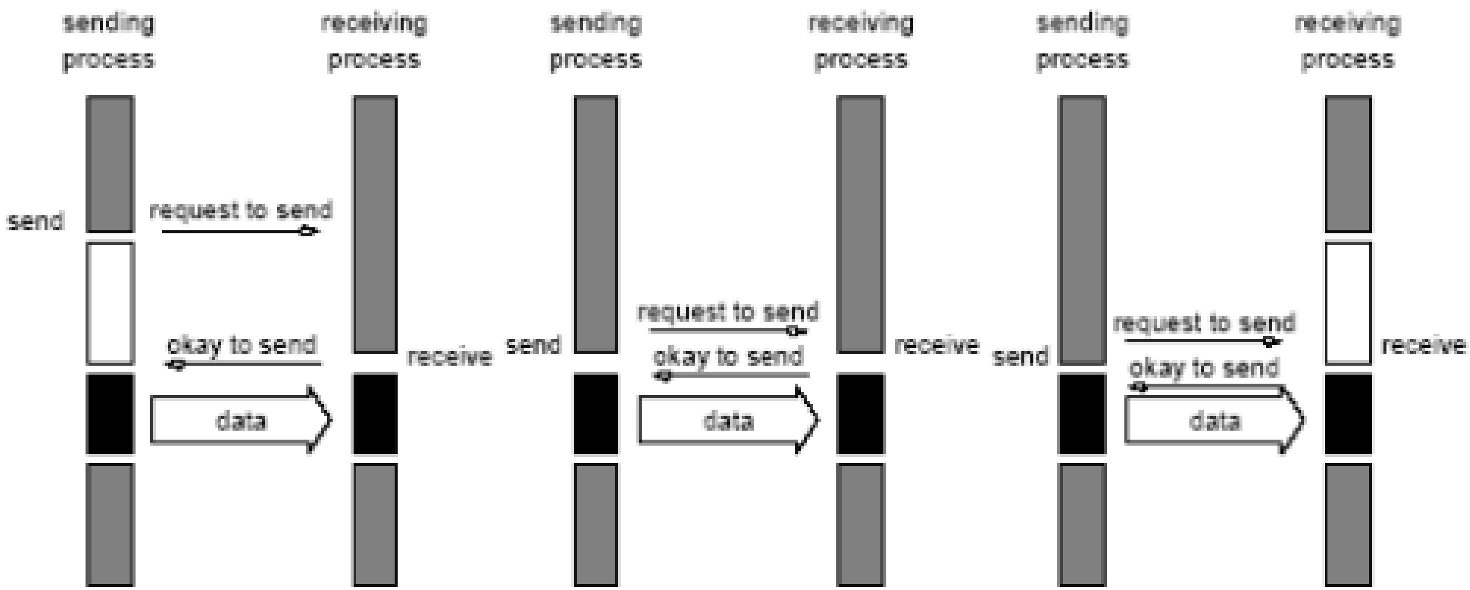
- **Non-buffered blocking**
- **Buffered blocking**
- **Non-blocking (非阻塞)**

Non-Buffered Blocking Message Passing Operations

- **A simple method for forcing （强制） send/receive semantics is for the send operation to return only when it is safe to do so**
- **In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process**
- **Idling and deadlocks are major issues with nonbuffered blocking sends**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Handshake for a blocking non-buffered send/receive operation.
It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Non-Buffered Blocking ⇒ Buffered Blocking

- **In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well**
- **Buffering alleviates idling at the expense of copying overheads**

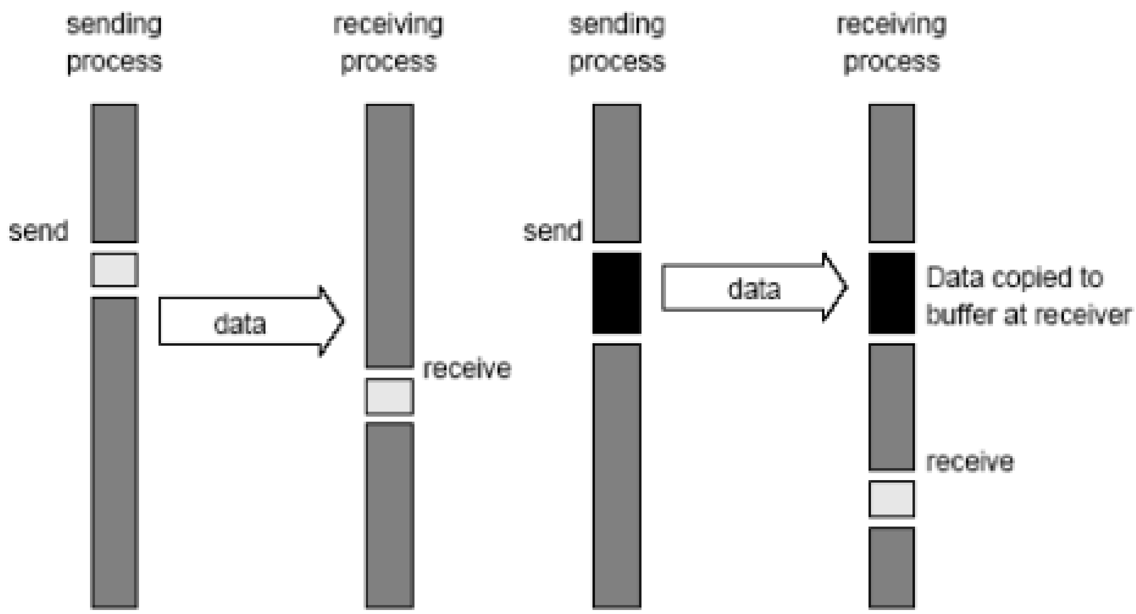
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Buffered Blocking Message Passing Operations

- **In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed.**
- **The data is copied at a buffer at the receiving end as well**
- **Buffering trades off idling overhead for buffer copying overhead**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols: (a) in the ***presence*** of communication hardware with buffers at send and receive ends; and (b) in the ***absence*** of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Buffered Blocking Message Passing Operations

- Bounded buffer sizes can have significant impact on performance

P0

```
for (i = 0; i < 1000; i++){
    produce_data(&a);
    send(&a, 1, 1);
}
```

P1

```
for (i = 0; i < 1000; i++){
    receive(&a, 1, 0);
    consume_data(&a);
}
```

What if consumer was much slower than producer?

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Buffered Blocking Message Passing Operations

- **Bounded buffer sizes can have significant impact on performance**

P0

```
for (i = 0; i < 1000; i++){
   produce_data(&a);
   send(&a, 1, 1);
}
```

P1

```
for (i = 0; i < 1000; i++){
   receive(&a, 1, 0);
   consume_data(&a);
}
```

What if consumer was much slower than producer?

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Buffered Blocking Message Passing Operations

- **Bounded buffer sizes can have significant impact on performance**

```
P0                                P1
for (i = 0; i < 1000; i++){       for (i = 0; i < 1000; i++){
   produce_data(&a);                 receive(&a, 1, 0);
   send(&a, 1, 1);                   consume_data(&a);
}                                 }
```

What if consumer was much slower than producer?

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Buffered Blocking Message Passing Operations

- Deadlocks are still possible with buffering since receive operations block.

```
P0                         P1

receive(&b, 1, 1);         receive(&a, 1, 0);

send(&a, 1, 1);            send(&b, 1, 0);
```

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Buffered Blocking Message Passing Operations

- **Deadlocks are still possible with buffering since receive operations block.**

P0	P1
`receive(&b, 1, 1);`	`receive(&a, 1, 0);`
`send(&a, 1, 1);`	`send(&b, 1, 0);`

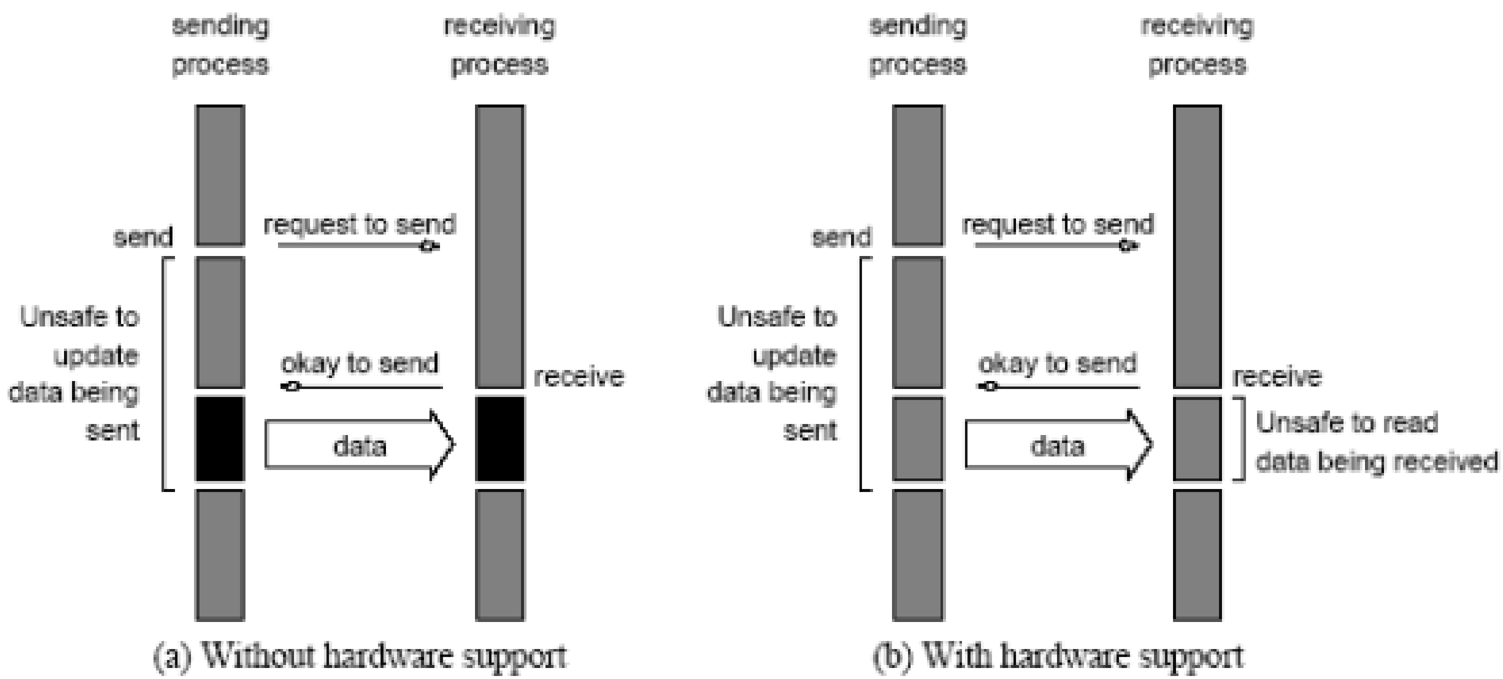
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Non-Blocking Message Passing Operations

- **The programmer must ensure semantics of the send and receive.**
- **This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.**
- **Non-blocking operations are generally accompanied by a check-status operation.**
- **When used correctly, these primitives are capable of overlapping communication overheads with useful computations.**
- **Message passing libraries typically provide both blocking and non-blocking primitives.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

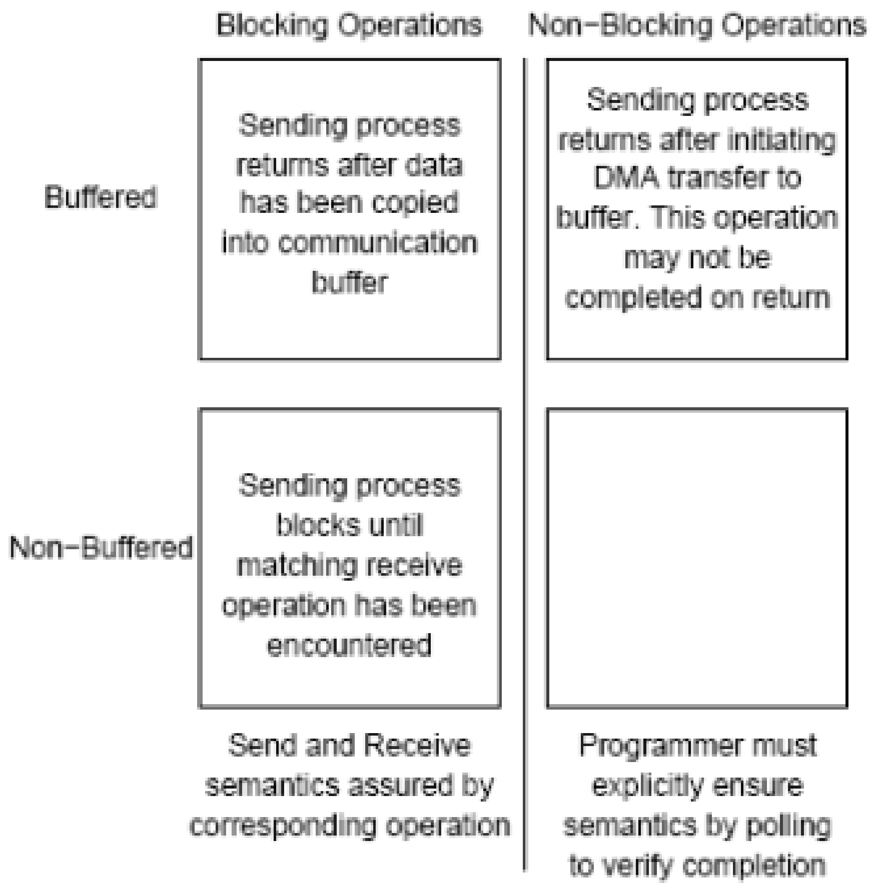
Non-Blocking Message Passing Operations



Non-blocking non-buffered send and receive operations (a) in ***absence*** of communication hardware; (b) in ***presence*** of communication hardware.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Send and Receive Protocols



Space of possible protocols for send and receive operations.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

The Message Passing Interface

- **Late 1980s: vendors had unique libraries**
- **1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab**
- **1992: Work on MPI standard begun**
- **1994: Version 1.0 of MPI standard**
- **1997: Version 2.0 of MPI standard**
- **Today: MPI is dominant massage passing library standard**

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

MPI: the Message Passing Interface

- **MPI defines a standard *library* for message-passing that can be used to develop portable message-passing programs using either C or Fortran.**
- **The MPI standard defines both the syntax as well as the semantics of a core set of library routines.**
- **Vendor implementations of MPI are available on almost all commercial parallel computers.**
- **It is possible to write fully-functional message-passing programs by using only the six routines.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

MPI Implementations and Tutorials

- **Standard**
  - **http://www.mpi-forum.org**
- **Implementations**
  - **MPICH2 http://www.mcs.anl.gov/research/projects/mpich2/**
  - **Open MPI http://www.open-mpi.org/**
- **Tutorials**
  - **https://computing.llnl.gov/tutorials/mpi/**
  - **http://www.mcs.anl.gov/research/projects/mpi/**

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Starting and Terminating the MPI Library

- *`MPI_Init`* **is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment**
- *`MPI_Finalize`* **is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment**
- **The prototypes of these two functions are:**

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- *`MPI_Init`* **also strips off any MPI related command-line arguments.**
- **All MPI routines, data-types, and constants are prefixed by "*`MPI_`*". The return code for successful completion is *`MPI_SUCCESS`***

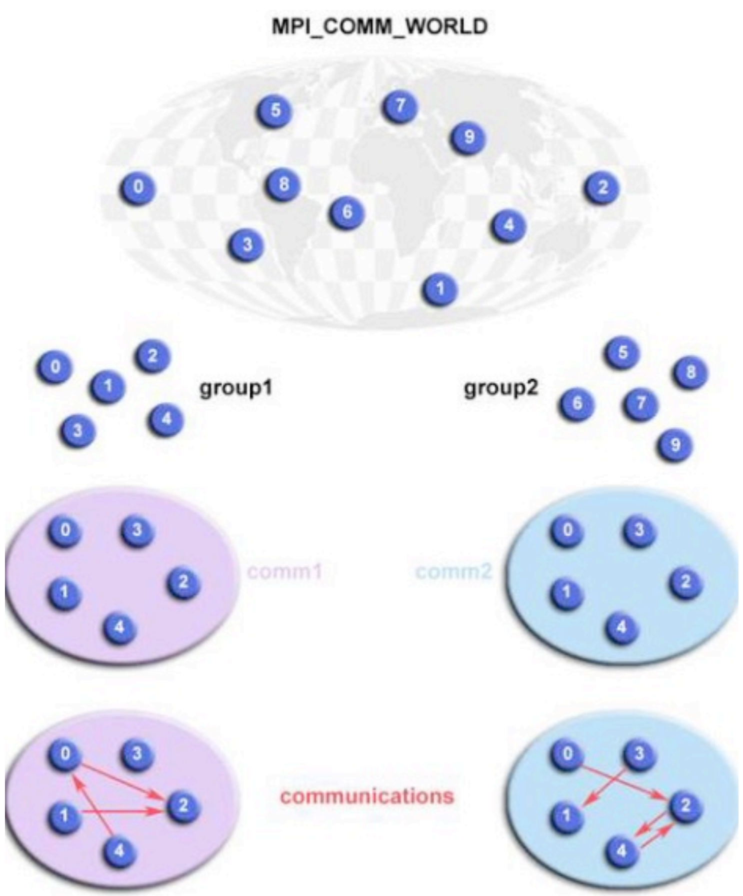
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Some Basic Concepts

- **Processes can be collected into groups**
- **Each message is sent in a context, and must be received in the same context**
  - **Provides necessary support for libraries**
- **A group and context together form a communicator**
- **A process is identified by its rank in the group associated with a communicator**
- **There is a default communicator whose group contains all initial processes, called** *MPI_COMM_WORLD.*

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Groups and Communicators



- **Related MPI functions**
  - **Form new group as a subset of global group using** *MPI_Group_incl*
  - **Create new communicator for new group using** *MPI_Comm_create*
  - **Determine new rank in new communicator using** *MPI_Comm_rank*
  - **Conduct communications using any MPI message passing routine**
  - **When finished, free up new communicator and group (optional) using** *MPI_Comm_free* **and** *MPI_Group_free*

B. Barney, “Message Passing Interface (MPI)”, LLNL.

Communicators

- **A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other**
- **Information about communication domains is stored in variables of type** *`MPI_Comm`*
- **Communicators are used as arguments to all message transfer MPI routines**
- **A process can belong to many different (possibly overlapping) communication domains**
- **MPI defines a default communicator called** *`MPI_COMM_WORLD`* **which includes all the processes**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Querying Information

- The *MPI_Comm_size* and *MPI_Comm_rank* functions are used to determine the number of processes and the label of the calling process, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

MPI: Hello World!

```
#include <mpi.h>

main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello World!\n",
                myrank, npes);
        MPI_Finalize();
}
```

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Compiling and Running MPI Programs

- Compiling examples
  - mpicc –o foo foo.c
  - mpic++ -o bar bar.cpp
- Running examples
  - mpirun –np 4 foo
  - mpirun –np 2 foo : -np 4 bar
- Specifying host processors
  - see "--hostfile" and "--host" options

MPI Messages

- **data : (address, count, datatype)**
  - **Datatype is either a predefined type, or a custom type**
- **message : (data, tag)**
  - **Tag is an integer to assist the receiving process in identifying the message;**

Sending and Receiving Messages

- **The basic functions for sending and receiving messages in MPI are the** *`MPI_Send`* **and** *`MPI_Recv`***, respectively**
- **The calling sequences of these routines are as follows:**

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype
 datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- **MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons**
- **The datatype** *`MPI_BYTE`* **corresponds to a byte (8 bits) and** *`MPI_PACKED`* **corresponds to a collection of data items that has been created by packing non-contiguous data**
- **The message-tag can take values ranging from zero up to the MPI defined constant** *`MPI_TAG_UB`*

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

MPI’s Send Modes (1/2)

- **MPI_Send （标准模式）**
  - **Will not return until you can use the send buffer (Non-local)**
- **MPI_Bsend （缓冲模式）**
  - **Returns immediately and you can use the send buffer**
  - **Related: MPI_buffer_attach(), MPI_buffer_detach()**
- **MPI_Ssend （同步模式）**
  - **Will not return until matching receive posted**
  - **Send + synchronous communication semantics**
- **MPI_Rsend （就绪模式）**
  - **May be used ONLY if matching receive already ready**
  - **The sender provides additional information to the system that can save some overhead**

区别？

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

MPI's Send Modes (2/2)

- **MPI_Isend （非阻塞标准模式）**
  - **Nonblocking send, but you can NOT reuse the send buffer immediately**
  - **Related: MPI_Wait(), MPI_Test()**
- **MPI_Ibsend**
- **MPI_Issend**
- **MPI_Irsend**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

 https://hpc-tutorials.llnl.gov/mpi/exercise_1/

Sending and Receiving Messages

- **MPI allows specification of wildcard arguments for both source and tag.**
- **If source is set to *MPI_ANY_SOURCE*, then any process of the communication domain can be the source of the message.**
- **If tag is set to *MPI_ANY_TAG*, then messages with any tag are accepted.**
- **On the receive side, the message must be of length equal to or less than the length field specified.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the *MPI_Recv* operation
- The corresponding data structure contains:

```
typedef struct MPI_Status {
int MPI_SOURCE;
int MPI_TAG;
int MPI_ERROR; };
```

- The *MPI_Get_count* function returns the precise count of data items received

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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Avoiding Deadlocks

https://github.com/LLNL/HPC-Tutorials/tree/main/mpi/examples

Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If *MPI_Send* is blocking, there is a deadlock.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Some Solutions to the "Unsafe" Problem

- "Unsafe" send/receive

P0	P1
Send(1)	Send(0)
Recv(1)	Recv(0)

- Order the operations more carefully

P0	P1
Send(1)	Recv(0)
Recv(1)	Send(0)

- Supply receive buffer at the same time as send

P0	P1
Sendrecv(1)	Sendrecv(0)

K. Yelick, "Distributed Memory Programming in MPI and UPC", Par Lab Boot Camp 2011.

Some Solutions to the "Unsafe" Problem

- Supply own space as buffer for send

P0	P1
Bsend(1)	Bsend(0)
Recv(1)	Recv(0)

- User non-blocking operations

P0	P1
Isend(1)	Irecv(0)
Irecv(1)	Isend(0)
Waitall	Waitall

K. Yelick, "Distributed Memory Programming in MPI and UPC", Par Lab Boot Camp 2011.

Avoiding Deadlocks

Using non-blocking operations remove most deadlocks. Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Overlapping Communication with Computation

- **In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.**

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- **These operations return before the operations have been completed. Function *`MPI_Test`* tests whether or not the non-blocking send or receive operation identified by its request has finished.**

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

- ***`MPI_Wait`* waits for the operation to complete.**

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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Avoiding Deadlocks

Consider the following piece of code, in which process *i* sends a message to process *i + 1* (module the number of processes) and receives a message from process *i - 1* (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
...
```

Once again, we have a deadlock if *MPI_Send* is blocking.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
}
else {
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
}
...
```

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the following function:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, int dest, int
sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvdatatype, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
int MPI_Sendrecv_replace(void *buf, int count,
MPI_Datatype datatype, int dest, int sendtag,
int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
```

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Collective Communication and Computation Operations （群组通信和计算）

- **MPI provides an extensive set of functions for performing common collective communication operations.**
- **Each of these operations is defined over a group corresponding to the communicator.**
- **All processors in a communicator must call these operations.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Collective Communication Operators (群通信操作子)

- **The barrier synchronization operation is performed in MPI using:**

  `int MPI_Barrier(MPI_Comm comm)` （阻塞到所有进程完成调用）

  The one-to-all broadcast operation is:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
int source, MPI_Comm comm)
```

- **The all-to-one reduction operation is:**

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

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Collective Communication Operations

- The operation *MPI_MAXLOC* combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- *MPI_MINLOC* does the same, except for minimum value of v_i.

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the MPI_MINLOC and MPI_MAXLOC operators.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Collective Communication Operations

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

MPI Datatype	C Datatype
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_INT	long and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Collective Communication Operations

- **If the result of the reduction operation is needed by all processes, MPI provides:**

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm
comm)
```

- **To compute prefix-sums, MPI provides:**

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Collective Communication Operations

- The gather operation is performed in MPI using:

```
int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
int recvcount, MPI_Datatype recvdatatype,
int target, MPI_Comm comm)
```

- MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.

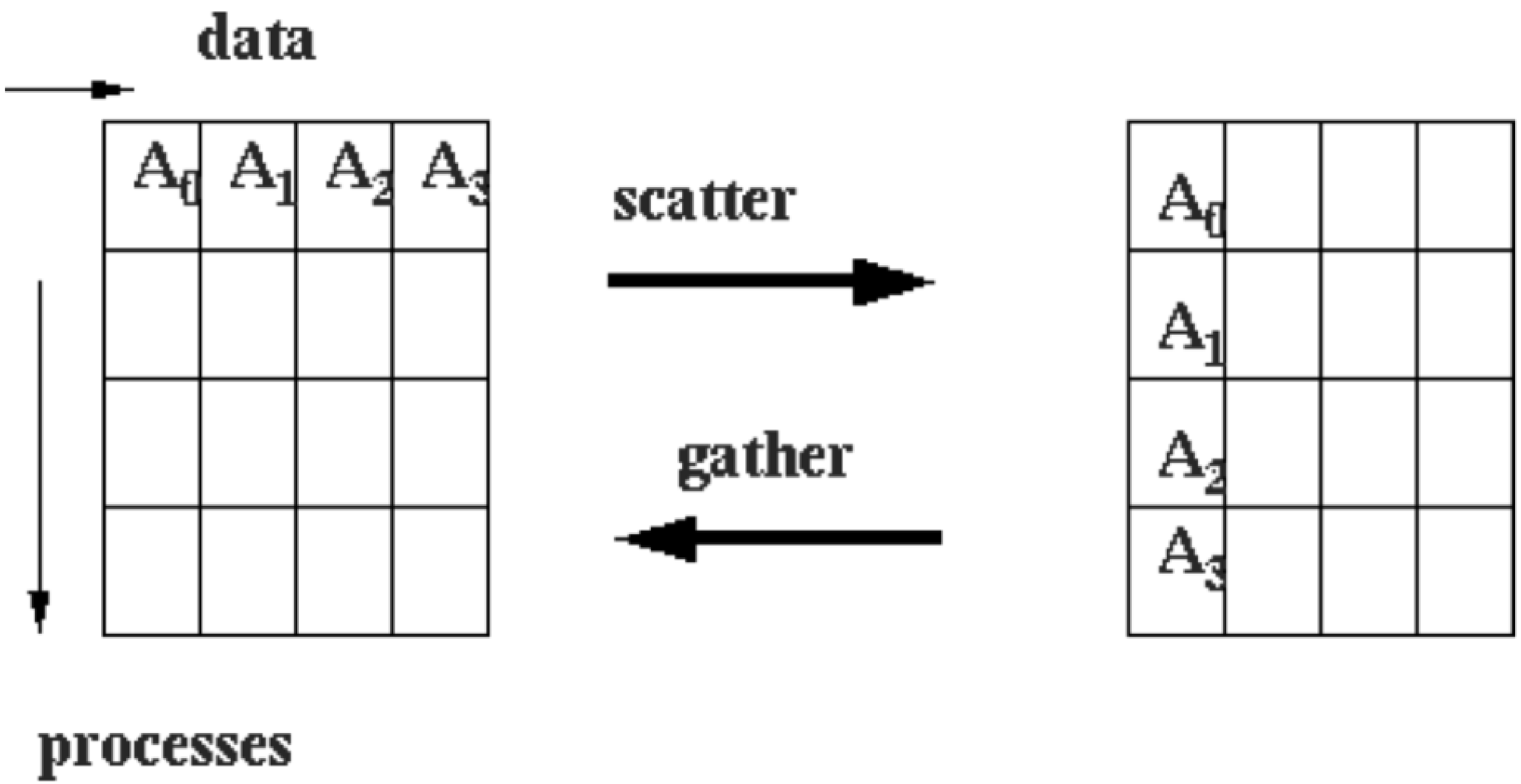
```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
int recvcount, MPI_Datatype recvdatatype,MPI_Comm comm)
```

- The corresponding scatter operation is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
int recvcount, MPI_Datatype recvdatatype,
int source, MPI_Comm comm) 与MPI_Bcast(…)区别？
```

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

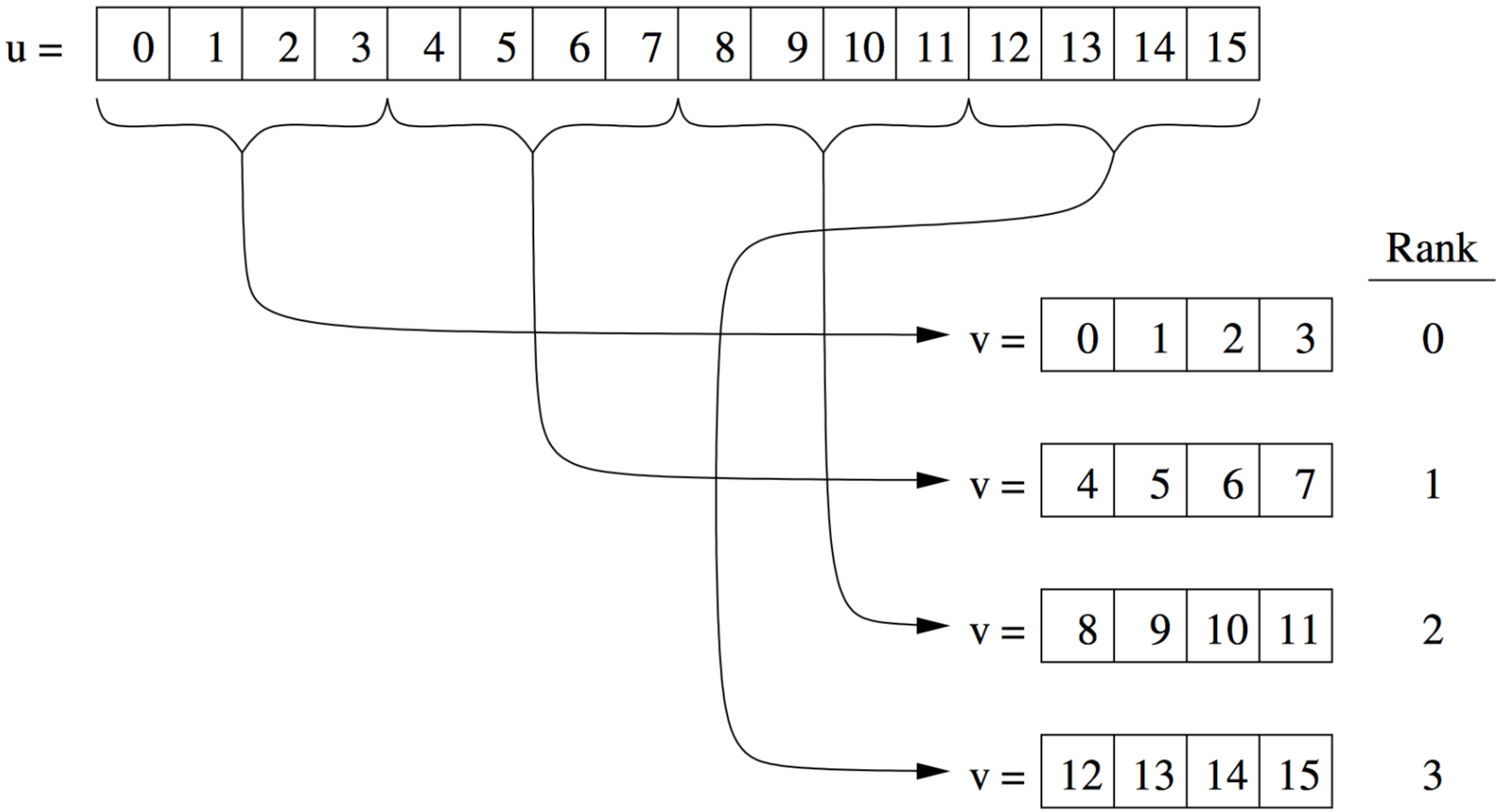
MPI_Gather and MPI_Scatter



A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

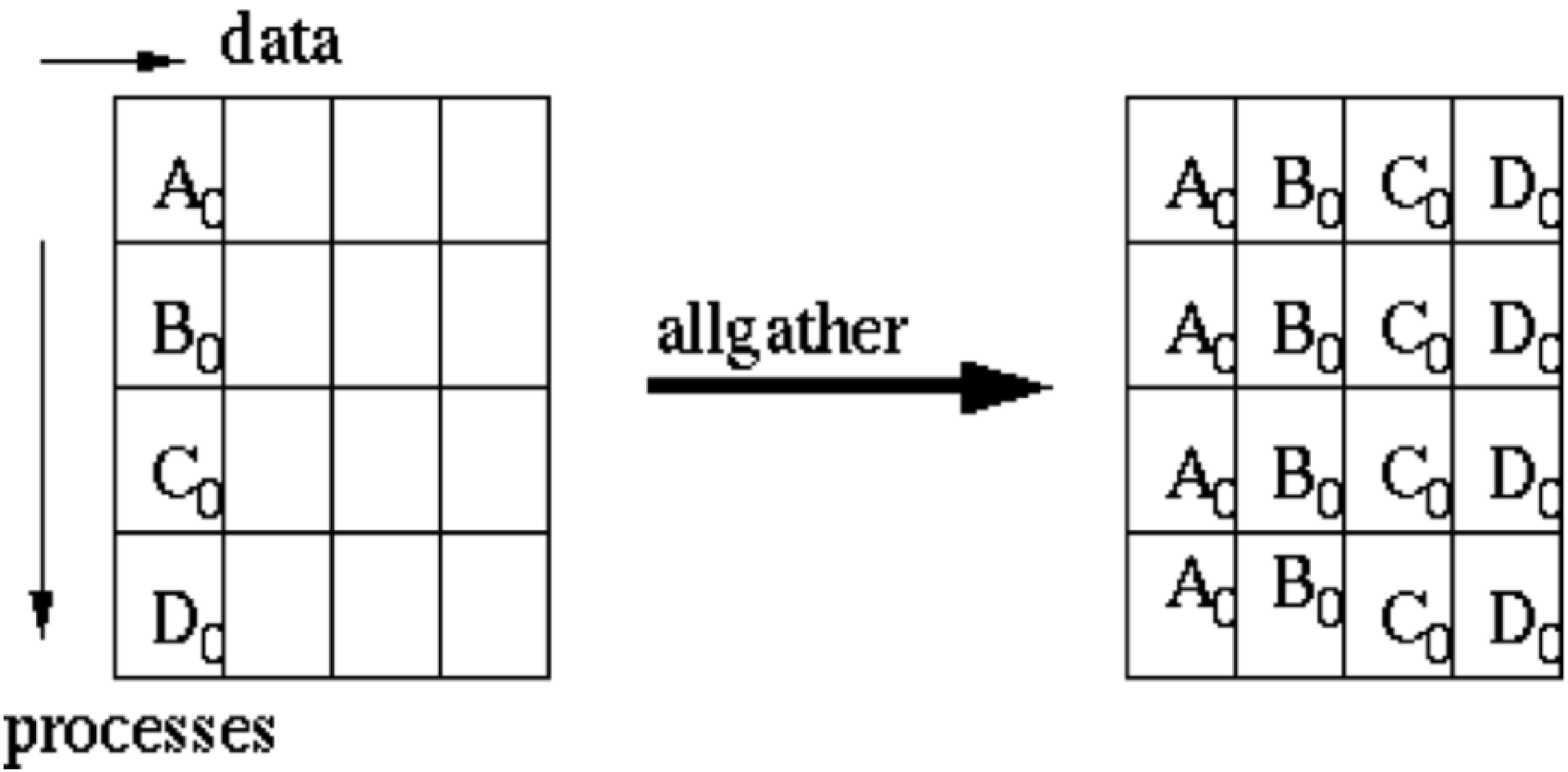
MPI Scatter()

```
MPI_Scatter(u, 4, MPI_INT, v, 4, MPI_INT, 0, MPI_WORLD_COMM);
```



A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

MPI Allgather()



A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Collective Communication Operations

- **The all-to-all personalized communication operation is performed by:**

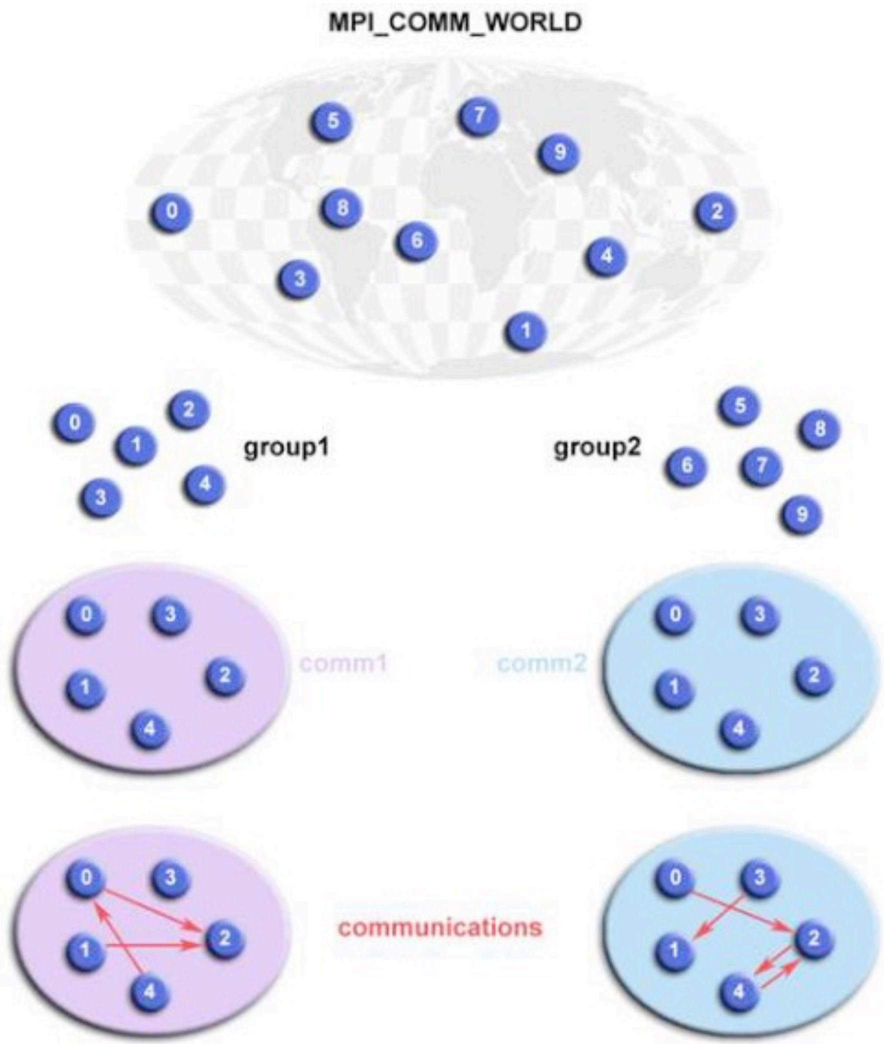
```
int MPI_Alltoall(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
int recvcount, MPI_Datatype recvdatatype, MPI_Comm
comm)
```

- **Using this core set of collective operations, a number of programs can be greatly simplified.**

Challenges?

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Groups and Communicators, Again



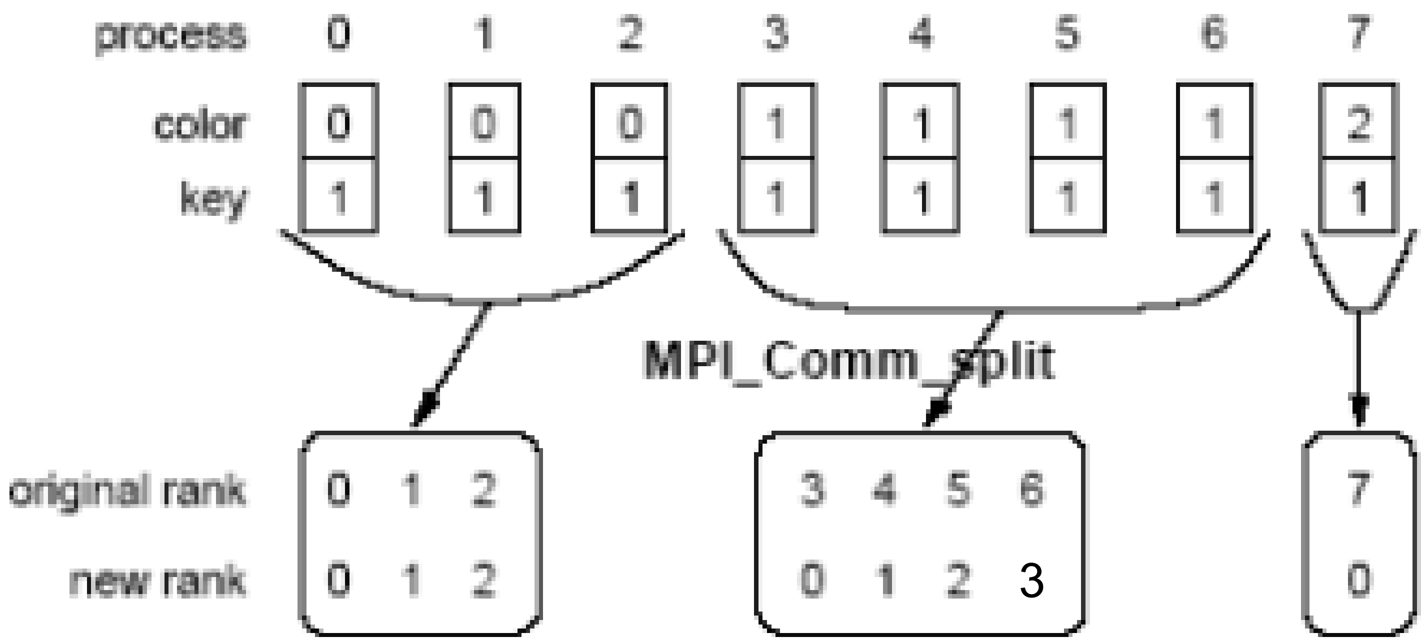
Groups and Communicators

- **In many parallel algorithms, communication operations need to be restricted to certain subsets of processes**
  - **See MPI_Group_*() and MPI_Comm_*() for more APIs**
- **MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator**
  - **The simplest such mechanism is:**

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *newcomm)
```

  - **This operation groups processors by color and sorts resulting groups on the key**

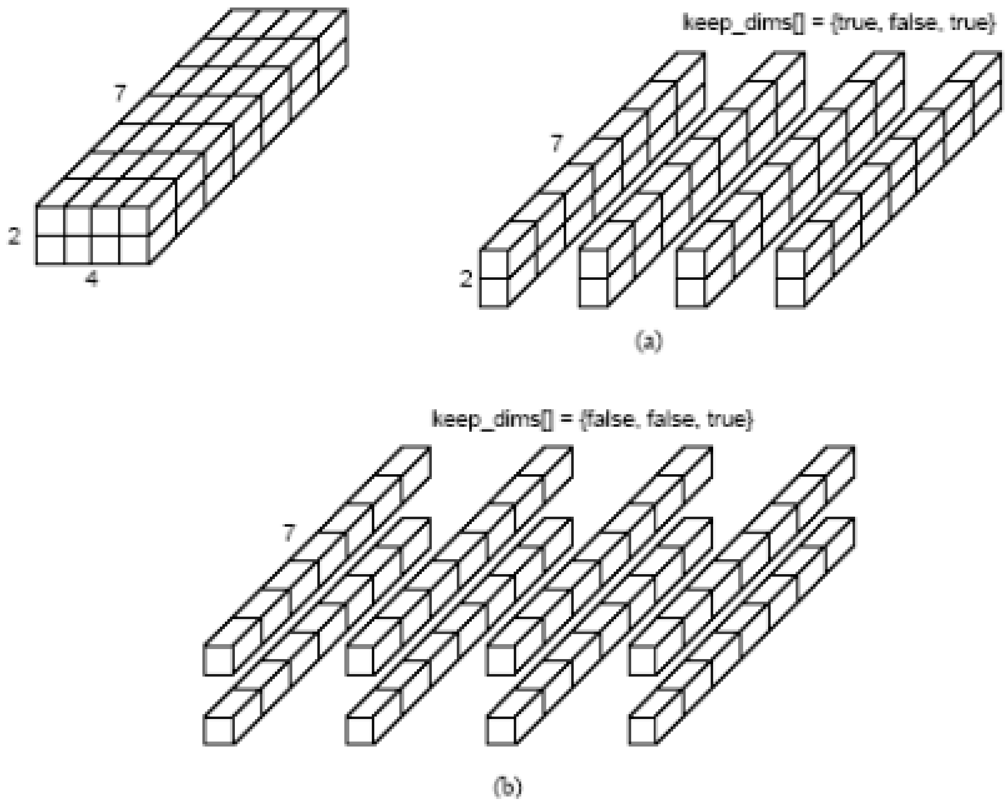
Groups and Communicators



Using *MPI_Comm_split* to split a group of processes in a communicator into subgroups.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

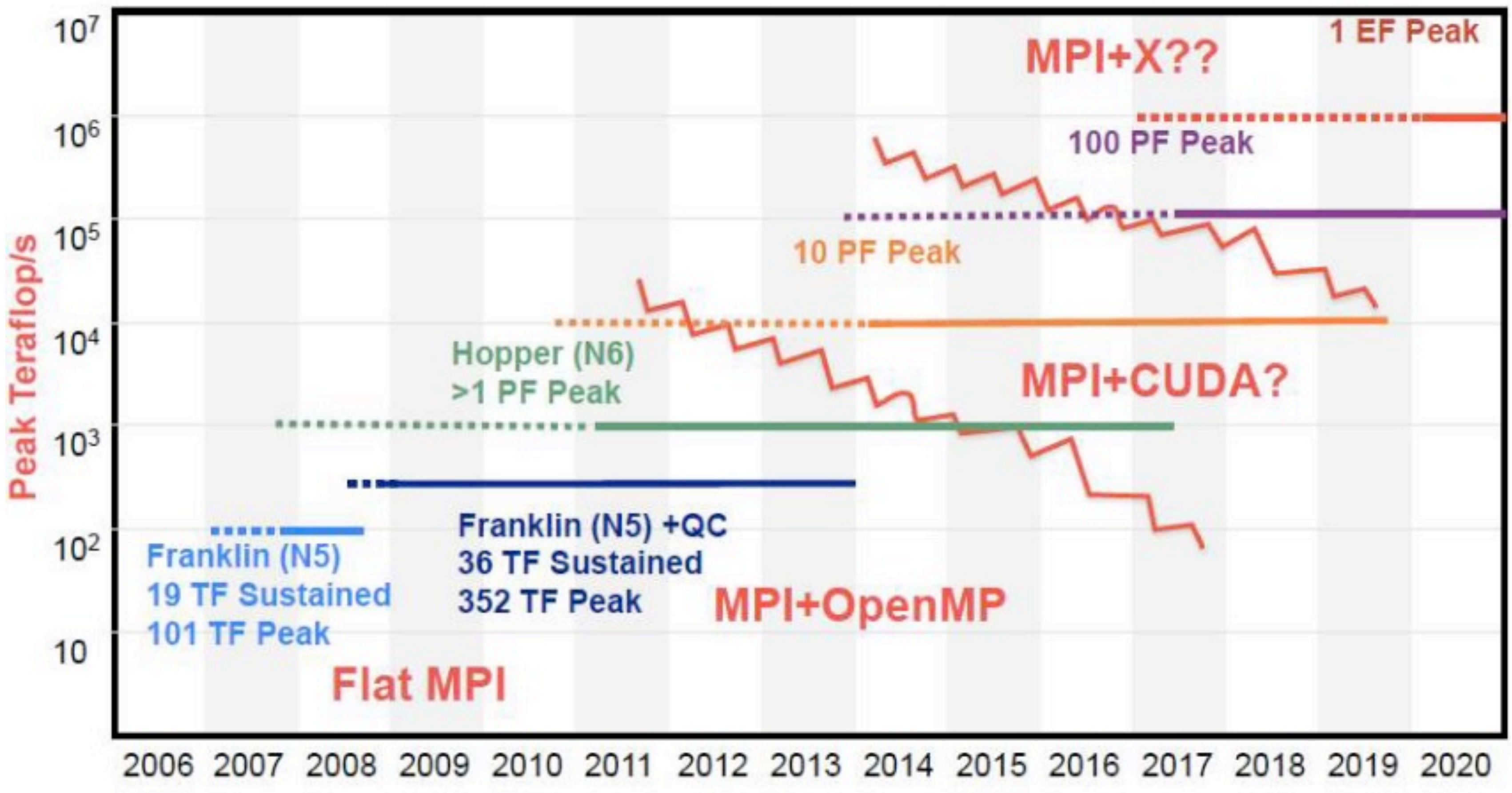
Groups and Communicators



Splitting a Cartesian topology of size 2 x 4 x 7 into (a) four subgroups of size 2 x 1 x 7, and (b) eight subgroups of size 1 x 1 x 7.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

What is the Ecosystem for Exascale?



Want to avoid two paradigm disruptions *on road to Exa-scale*

Thank You！