MAE 5032 High Performance Computing: Methods and Practices

Lecture 15: Message Passing Interface

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

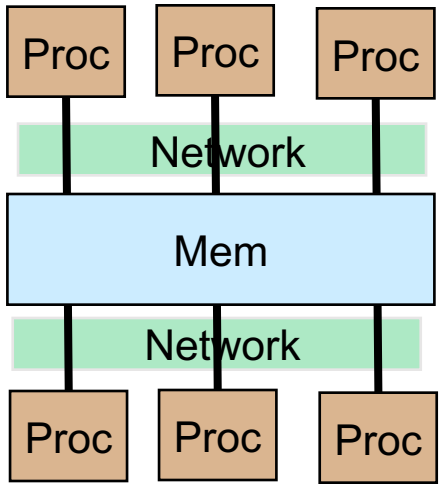
Why MPI

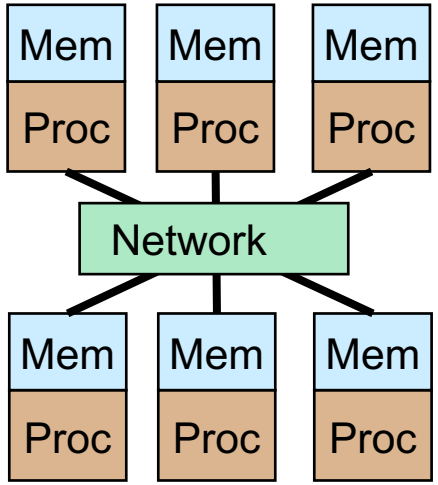
Sunway TaihuLight

Tianhe-2A

Allow solving large scientific/engineering problems

Recall: shared vs distributed memory





Shared memory system: each core has access to a common memory

Distributed memory system: each core has access to a private memory

Recall: shared vs distributed memory

Shared memory parallelism: (not covered)

- enabled by multi-threading
- easy to access data from any threads
- must handle cache coherence, NUMA architecture, etc.

➢ looks simple, but there is a lot implicit complexity

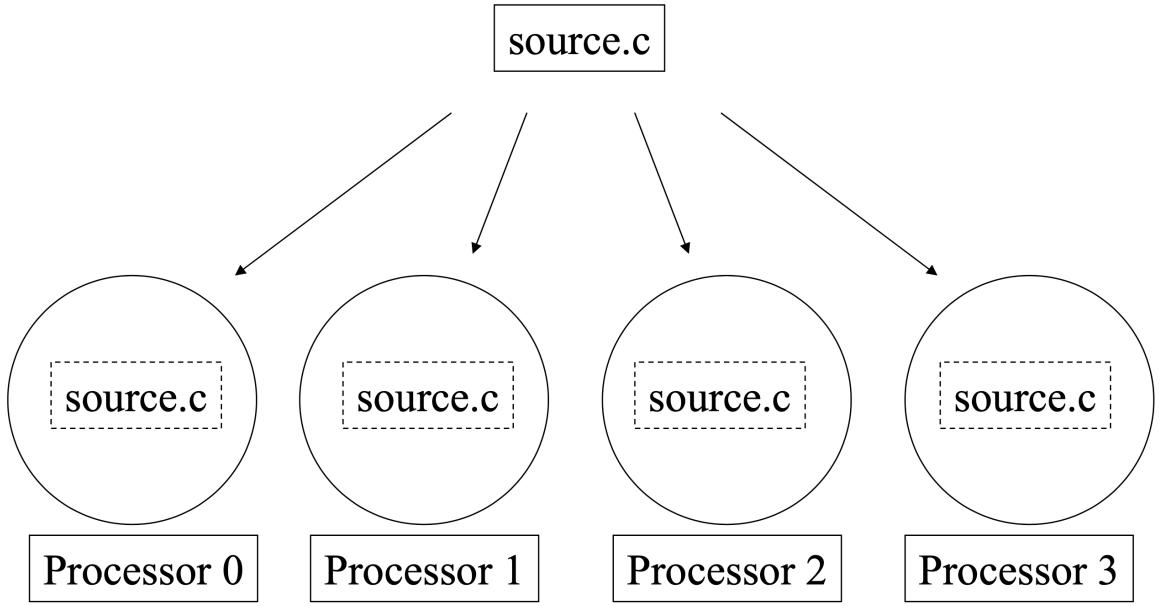
Distributed memory parallelism:

- only possible through multi-processing
- each process has its own private memory
- must explicitly exchange data across processes
- can scale beyond a single computer to solve much larger problems

SPMD programming model

SPMD: single program multiple data

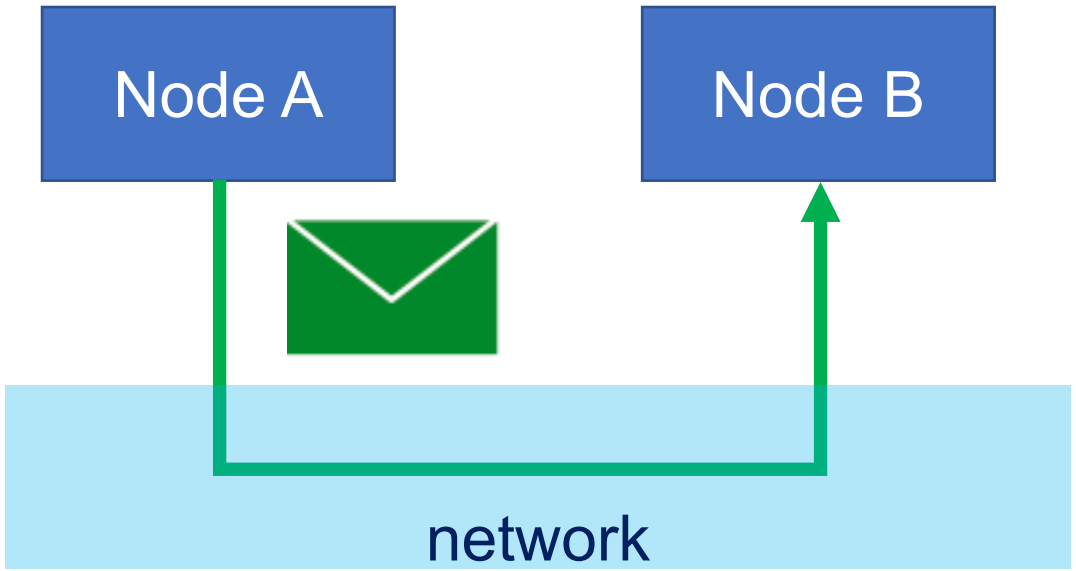
- A programming model for distributed memory parallelism
- Parallelism is achieved by creating multiple instances (processes) of the same program
- Each process operates on its own set of data
- Each process might follow a different execution path



SPMD programming model

SPMD: single program multiple data

- processes cannot communicate or synchronize implicitly through the shared memory
- we need explicit communication mechanism: message passing
- message can be classified in terms of participants:
  - **point-to-point communication**
  - **collective communication**

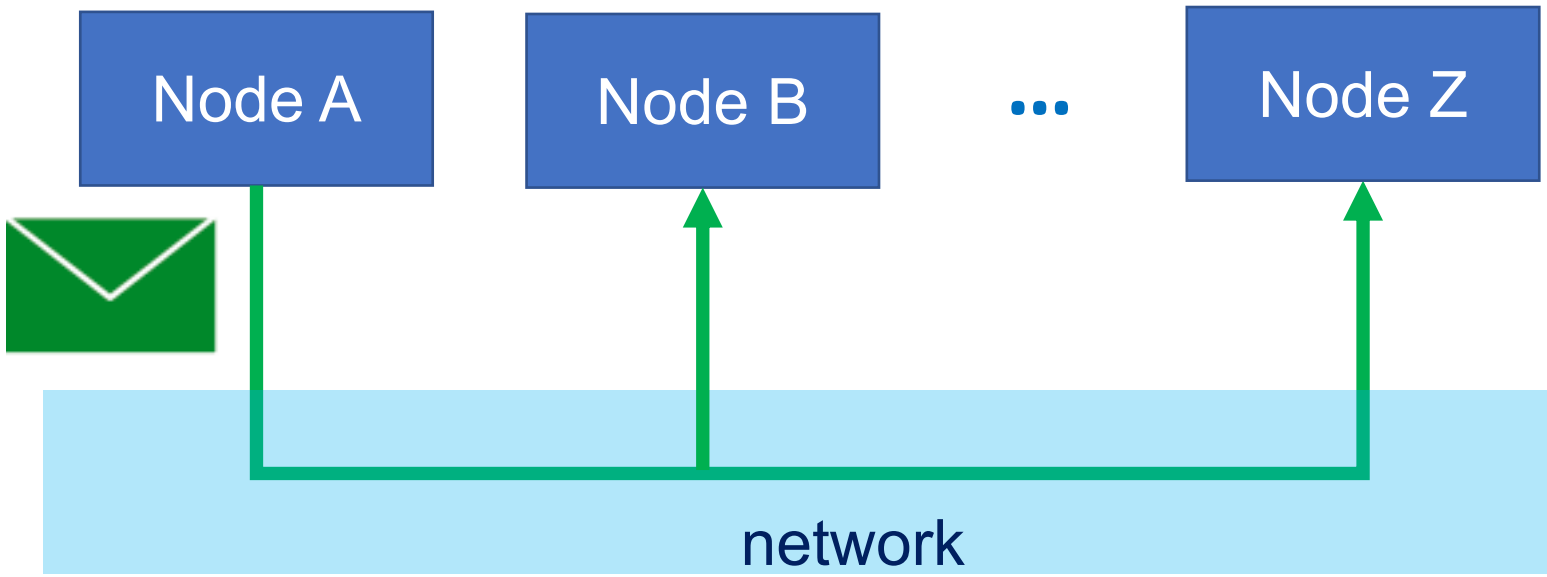


Point-to-point: processor A sends a message to another processor B

SPMD programming model

SPMD: single program multiple data

- processes cannot communicate or synchronize implicitly through the shared memory
- we need explicit communication mechanism: message passing
- message can be classified in terms of participants:
  - **point-to-point communication**
  - **collective communication**



Collective communication:

- involving multiple processes
- can be expressed in terms of point-to-point communications

MPI: Message Passing Interface

- MPI defines a standard API for message passing in SPMD applications
- MPI allows exploiting distributed memory systems
- There are several implementations
  - MPICH https://www.mpich.org
  - OpenMP https://www.open-mpi.org
  - MVAPICH http://mvapich.cse.ohio-state.edu
  - DeinoMPI https://mpi.deino.net
- Official documentation
  - https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

History of the MPI standard

- MPI-1
  - 1992 standardization of the message passing protocol into MPI . More than 60 people from 40 organizations involved. Point-to-point, block collective communications, one-sided communication, I/O, …
- MPI-2
  - Fortran and C++ wrappers
- MPI-3
  - Non-blocking collective communications
- The MPI standard API goals:
  - Work on distributed memory, shared memory, and hybrid systems
  - Portable, efficient, easy to use

Compile and execute MPI programs

- MPI is simply a C library, we only need to link it to the main program
- MPI usually provide a wrapper mpicc/mpicxx/mpiifort

```
-> mpicc -show
gcc -Wl,-flat_namespace -Wl,-commons,use_dylibs -I/Users/juliu/lib/mpich-3.3.2/include
 -L/Users/juliu/lib/mpich-3.3.2/lib -lmpi -lpmpi
```

- Wrapper to launch multiple processes at once: mpiexec/mpirun

```
-> mpicc main.c -o hello
juliu::Kolmogorov {~/MAE5032
-> mpirun -np 2 ./hello
Hello MAE5032!
Hello MAE5032!
```

- The number of processors does NOT need to be passed to mpirun in LSF/bsub
- Some clusters have their own wrapper of mpirun (e.g. srun in SLURM)

Hello world with MPI

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

int main( int argc, char **argv )
{
  MPI_Init(&argc, &argv);

  printf("Hello MAE5032!\n");

  MPI_Finalize();
  return 0;
}
```

- MPI is simply a C library, we only need to link it to the main program
- MPI usually provide a wrapper mpicc/mpicxx
- Wrapper to launch multiple processes at once: mpiexec, mpirun

Hello world with MPI

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

int main( int argc, char **argv )
{
  MPI_Init(&argc, &argv);

  printf("Hello MAE5032!\n");

  MPI_Finalize();
  return 0;
}
```

Include header

Cannot call MPI routines

Initialize MPI environment

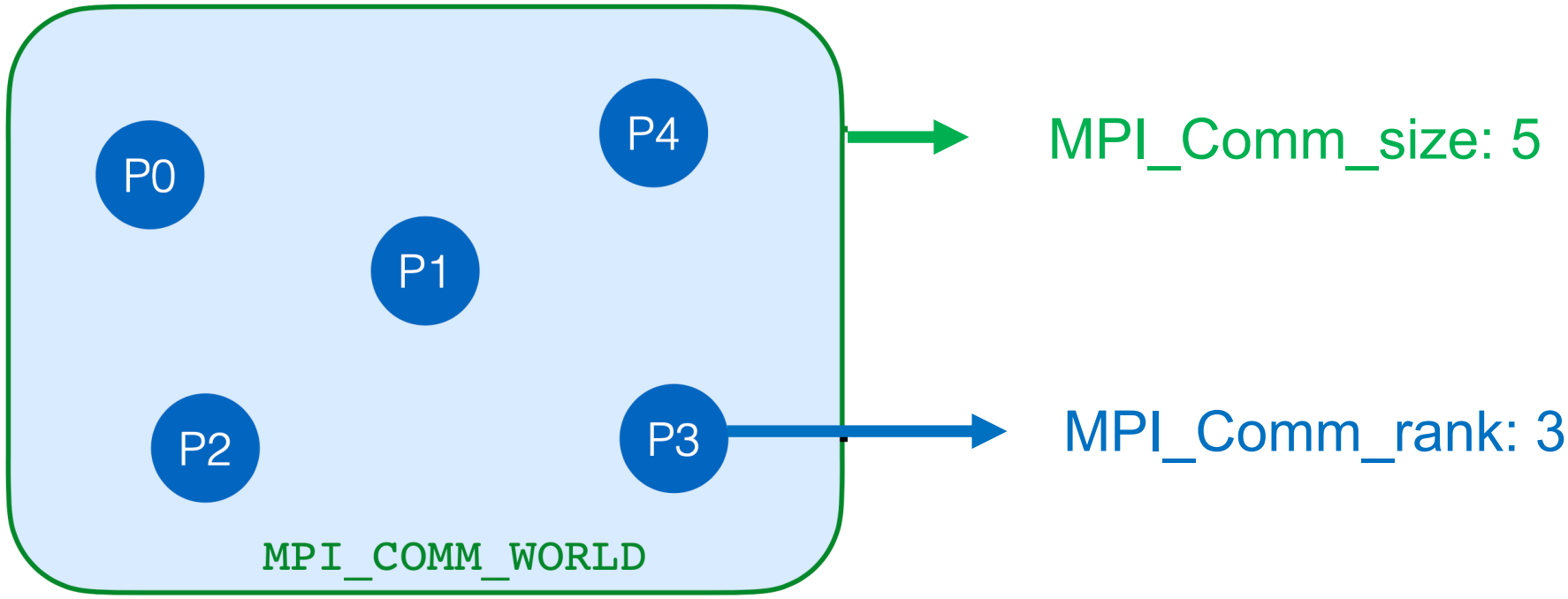
Can call MPI routines

Finalize MPI environment

Cannot call MPI routines

MPI communicators

- A communicator owns a group of processes that can communicate information among them. Each process has a unique rank within the communicator.
- **MPI_COMM_WORLD** is the communicator for all processes; defined after MPI_Init is called
- All MPI routines involving communications require a communicator object of tyep MPI_Comm



MPI rank and size

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

int main( int argc, char **argv )
{
  MPI_Init(&argc, &argv);

  int rank, size;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Hello MAE5032 from rank %d out of %d!\n",
      rank, size);

  MPI_Finalize();
  return 0;
}
```

Communicator of ALL processes

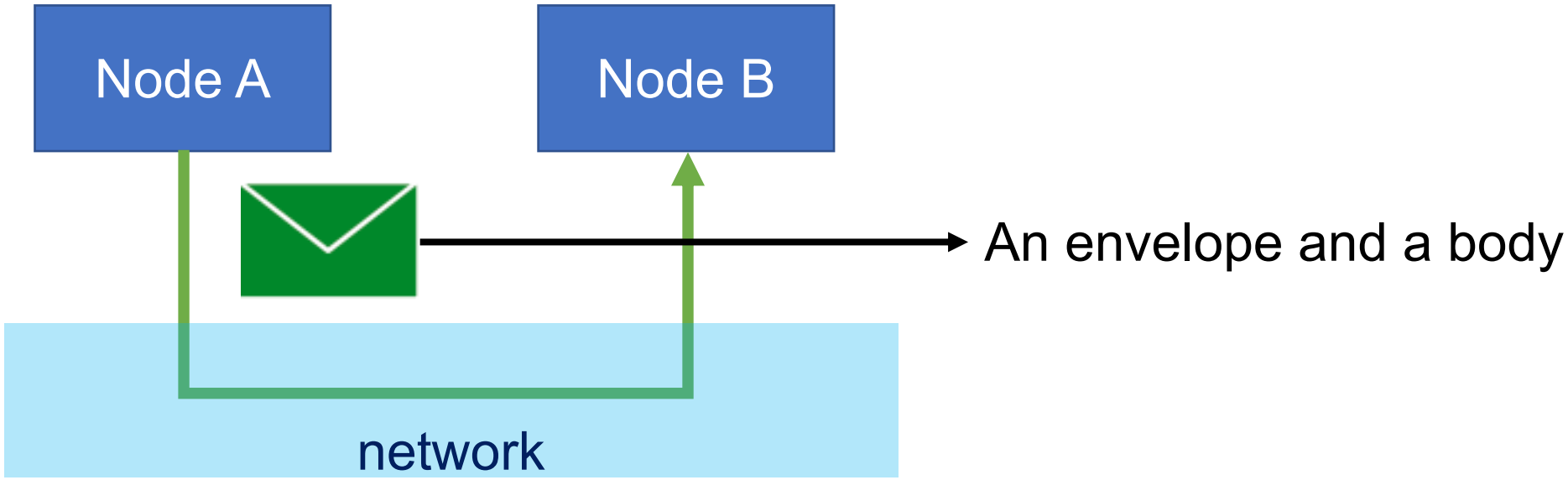
Get process ID

Get number of processes

Asynchronous execution: order is not determinstic

```
-> mpirun -np 3 ./a.out
Hello MAE5032 from rank 2 out of 3!
Hello MAE5032 from rank 0 out of 3!
Hello MAE5032 from rank 1 out of 3!
```

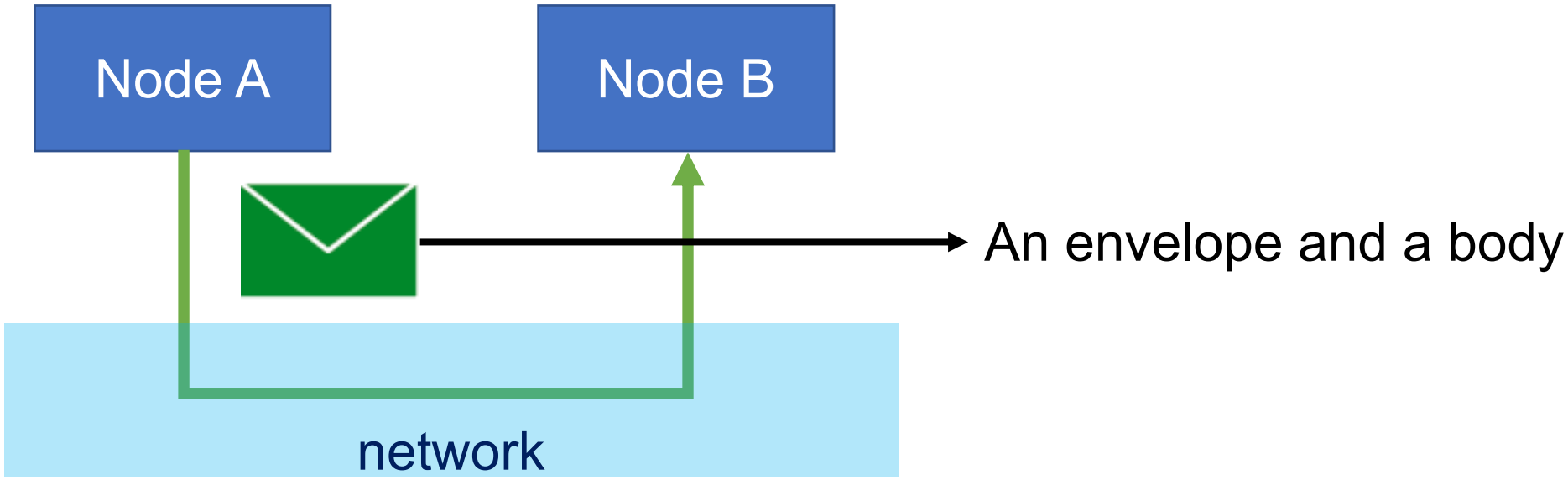
Point-to-point communication



Envelope				Body		
source	destination	communicator	tag	count	datatype	buffer
Rank 0	**Rank 1**	**MPI_COMM_WORLD**	**mytag**	**10**	**MPI_DOUBLE.**	**mydata**

Source, destination: Rank IDs within the communicator
Tag: user-defined message numeric label

Point-to-point communication



Envelope				Body		
source	destination	communicator	tag	count	datatype	buffer

count: number of elements
datatype: element data type
buffer: a pointer to the data itself

Point-to-point communication

```
int MPI_Send(const 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_Recv **matches** a message sent by MPI_Send only if comm, tag, source, and dest match.
- The count variable is the maximum size that can be received, message can be smaller.
- The tag can be set to the wildcard MPI_ANY_TAG
- Source can be set to the wildcard MPI_ANY_SOURC
- MPI_STATUS_IGNORE can be used if the status is n
- MPI_STATUS is a struct containing at least MPI_SOU MPI_ERROR

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

Point-to-point communication

MPI_Datatype	C type
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG_INT	long long int
MPI_LONG_LONG	long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Point-to-point communication

```
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);

  int tag = 42;
  int message;
  if(rank == 0)
  {
    message = 7;
    MPI_Send(&message, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  }
  else if(rank == 1)
  {
    MPI_Recv(&message, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Rank %d received the following integer: %d\n", rank, message);
  }

  MPI_Finalize();
  return 0;
}
```

```
-> mpirun -np 2 ./a.out
Rank 1 received the following integer: 7
```

Send one integer to rank 1

Receive one integer from rank 0

Point-to-point communication

```
int tag = 42;
int message;
if(rank == 0)
{
  for(int i=1; i<size; ++i)
  {
    message = i * i;
    MPI_Send(&message, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
  }
}
else
{
  MPI_Recv(&message, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Rank %d received the following integer: %d\n", rank, message);
}

MPI_Finalize();
return 0;
```

```
-> mpirun -np 5 ./a.out
Rank 2 received the following integer: 4
Rank 1 received the following integer: 1
Rank 3 received the following integer: 9
Rank 4 received the following integer: 16
```

Send one integer to all ranks !- 0

Receive one integer from rank 0

MPI_Send: communication modes

- MPI_Send is blocking: the function returns when the buffer becomes available
- MPI_Ssend: Synchronous send. Will return only once a matching recv has been posted and started receiving the message.
- MPI_Bsend: buffered send. Stores the message in a buffer and returns immediately. Users need to manually allocate the memory space.

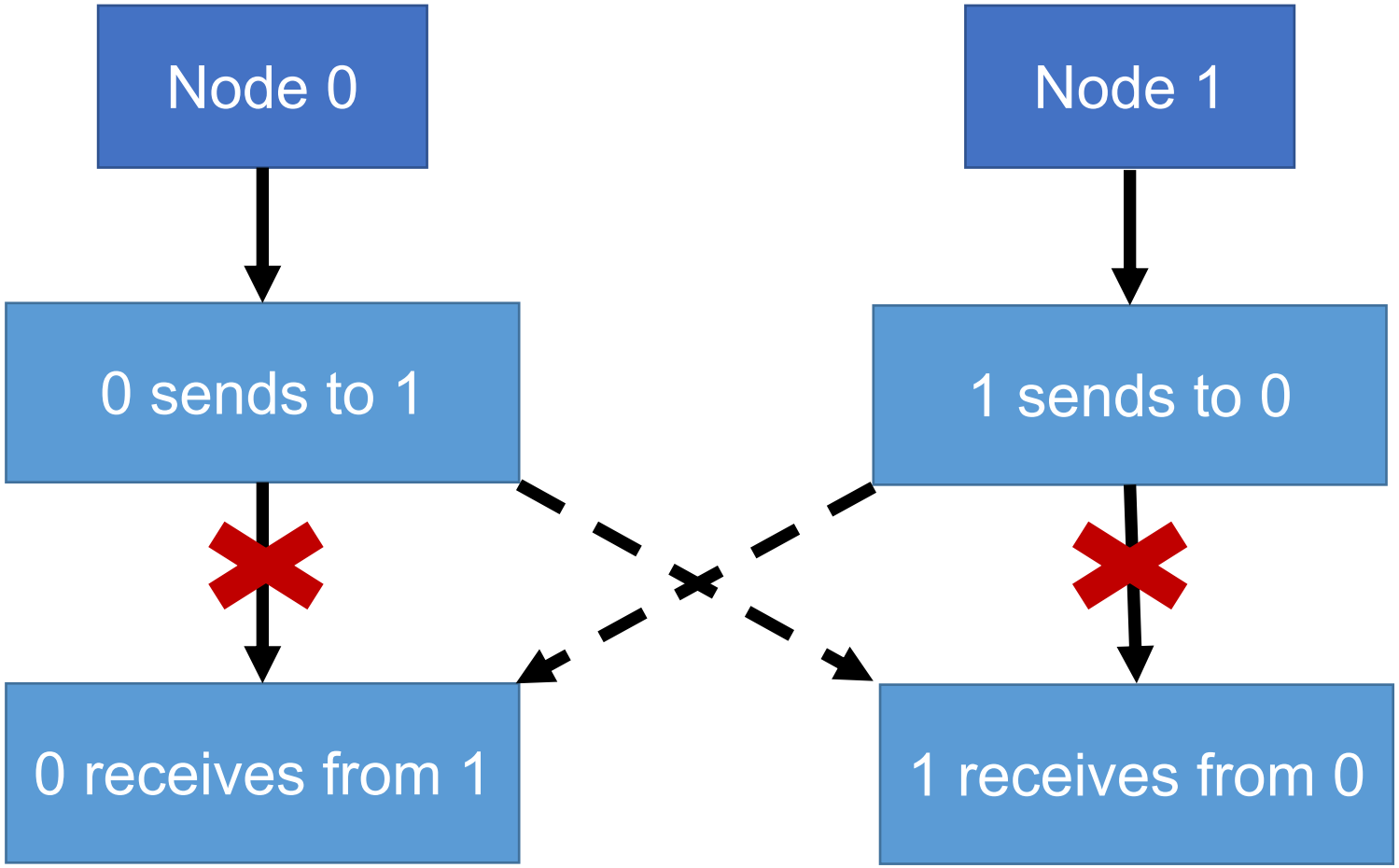
Prefer non-buffered sends to avoid unnecessary copies.

Deadlock

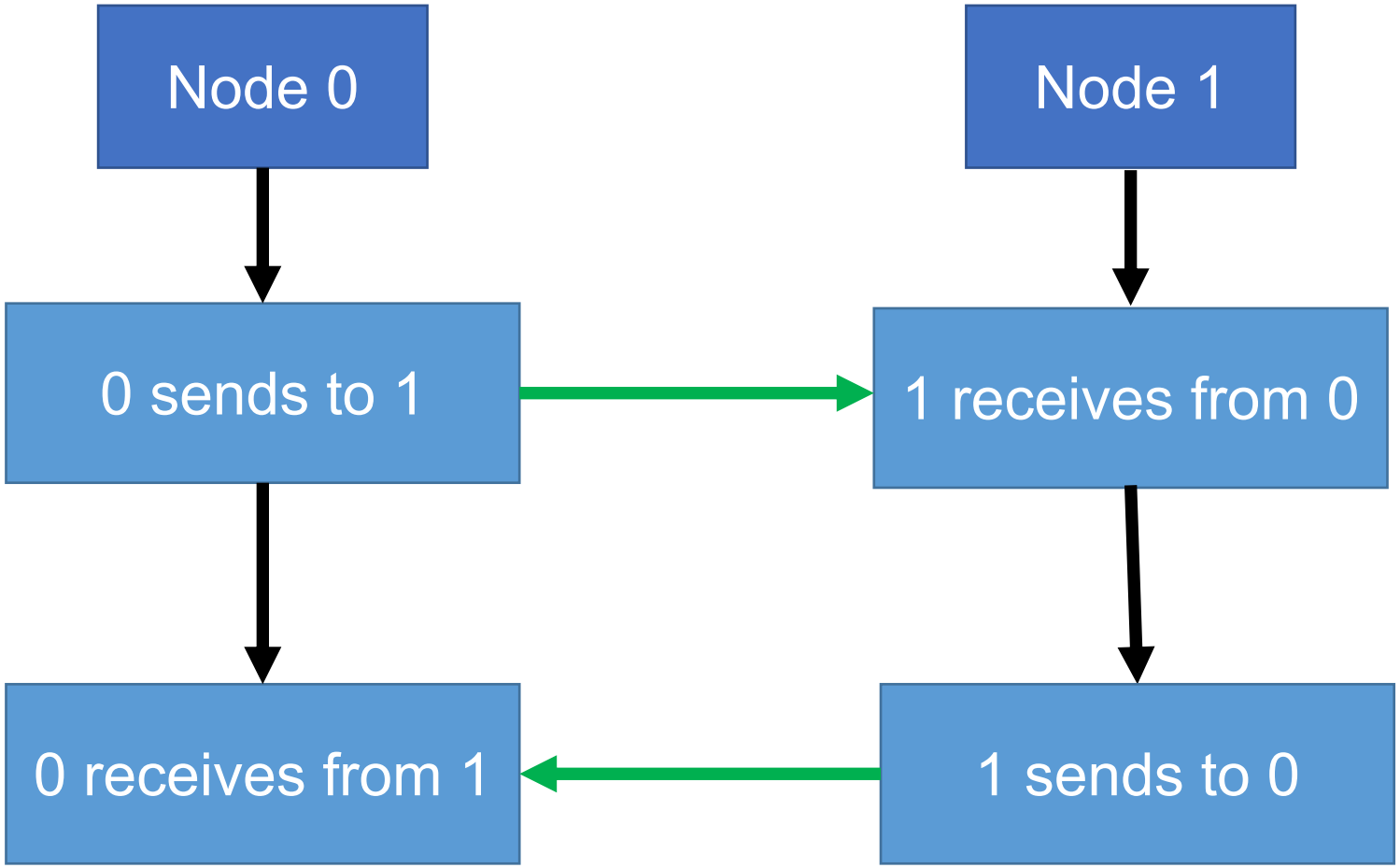
```
int tag1 = 42, tag2 = 43;
int sendMessage, recvMessage;
if(rank == 0)
{
  sendMessage = 7;
  MPI_Ssend(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if(rank == 1)
{
  sendMessage = 14;
  MPI_Ssend(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
  MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

printf("Rank %d received the following integer: %d\n", rank, recvMessage);
```

```
-> mpirun -np 2 ./a.out
```

Deadlock


Node 0
Node 1
0 sends to 1
1 sends to 0
0 receives from 1
1 receives from 0

Deadlock fix 1: switch the order



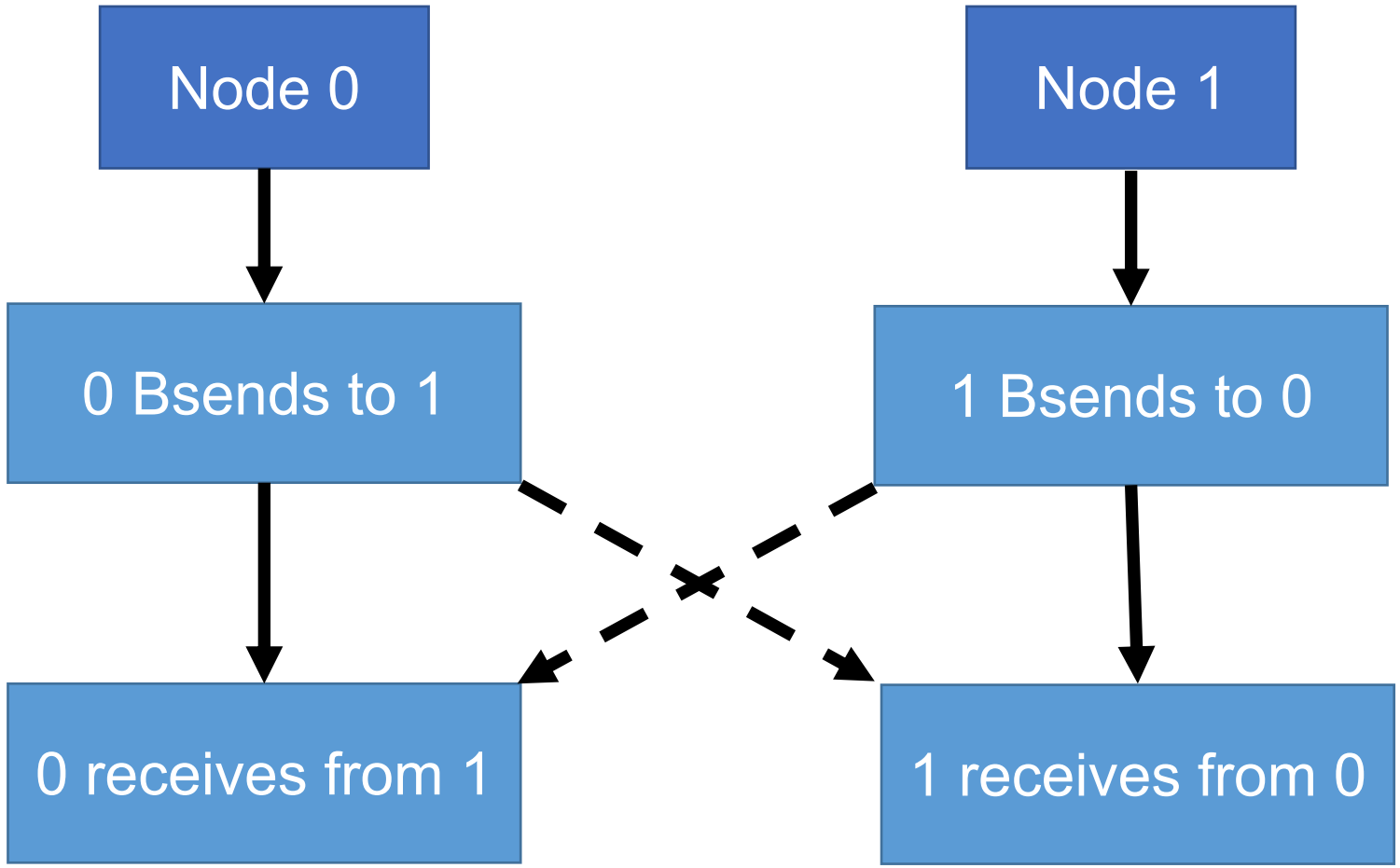
Deadlock fix 1: switch the order

```
int tag1 = 42, tag2 = 43;
int sendMessage, recvMessage;
if(rank == 0)
{
  sendMessage = 7;
  MPI_Ssend(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if(rank == 1)
{
  sendMessage = 14;
  MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Ssend(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
}

printf("Rank %d received the following integer: %d\n", rank, recvMessage);
```

```
-> mpirun -np 2 ./a.out
Rank 0 received the following integer: 14
Rank 1 received the following integer: 7
```

Deadlock fix 2: buffered send

Buffered send can return before the message is received.



Deadlock fix 3: non-blocking communication

Computation proceeds concurrently with send and receive

- Non-blocking send
  - Start send operation
  - Return before data copied out of buffer
  - Must wait until data sent to overwrite the buffer

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

- Non-blocking receive
  - Start receive operation
  - Return before data received and copied into buffer
  - Must wait until data received to use the data in buffer

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

Deadlock fix 3: non-blocking communication

- Test whether request complete:

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

- Wait for request to complete:

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

- request: handle on the non-blocking routine to wait on
- flag: the variable in which store the check result, true if the underlying non-blocking is compleet, false otherwise
- status: the struct containing the MPI_SOURCE, MPI_TAG, MPI_ERROR, etc.

Deadlock fix 3: non-blocking communication

```
int tag1 = 42, tag2 = 43;
int sendMessage, recvMessage;
MPI_Request sendReq;
MPI_Status status;

if(rank == 0)
{
  sendMessage = 7;
  MPI_Isend(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD, &sendReq);
  MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if(rank == 1)
{
  sendMessage = 14;
  MPI_Isend(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &sendReq);
  MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

MPI_Wait(&sendReq, &status);

printf("Rank %d received the following integer: %d\n", rank, recvMessage);
```

Non-blocking send returns immediately.

Send is completed here and the buffer can be used now.

Unknown message size

What if we don’t know the size of the message on the receive side in advance?

Solution 1: send two messages, one for size first, then another for content.

Solution 2: Use MPI_Probe to obtain the message info

```
if (rank == 0) {
  message.resize(rand() % 2048, 42.0);
  MPI_Send(message.data(), message.size(), MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
}
else if (rank == 1) {
  int size;
  MPI_Status status;
  MPI_Probe(0, tag, MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_DOUBLE, &size);
  message.resize(size);
  MPI_Recv(message.data(), message.size(), MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE); printf("Rank %d received %d doubles\n", rank, size);
}
```

Example: computing pi

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  long nsteps = 1000000000;
  double sum  = 0.0;

  for(long ii=0; ii<nsteps; ++ii)
    sum += (1.0 - 2.0 *(ii%2)) / (2.0 * ii + 1.0);

  printf("pi is %.12g \n", sum * 4.0);

  return 0;
}
```

```
-> ./serial
pi is 3.14159265259
```

How to parallelize this with MPI?

- Each rank performs its own chnk of the sum
- All ranks need to communicate and sum up its results to one rank
- This rank should report the result

Example: computing pi

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  long nsteps = 1000000000;
  double sum  = 0.0;

  for(long ii=0; ii<nsteps; ++ii)
    sum += (1.0 - 2.0 *(ii%2)) / (2.0 * ii + 1.0);

  printf("pi is %.12g \n", sum * 4.0);

  return 0;
}
```

```
-> ./serial
pi is 3.14159265259
```

How to parallelize this with MPI?

- Each rank performs its own chnk of the sum
- All ranks need to communicate and sum up its results to one rank
- This rank should report the result

Example: computing pi

```
MPI_Init(&argc, &argv);

int size, rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

long nsteps = 1000000000;
double sum  = 0.0;

long chunk = (nsteps + size - 1) / size;
long start = chunk * rank;
long end   = (chunk * (rank+1) > nsteps) ? nsteps : chunk*(rank+1);

double t1 = MPI_Wtime();

for(long ii=start; ii<end; ++ii)
  sum += (1.0 - 2.0 *(ii%2)) / (2.0 * ii + 1.0);

MPI_Barrier(MPI_COMM_WORLD);
double t2 = MPI_Wtime();
if(rank == 0) printf("Time taken is %f. \n", t2 - t1);
```

Split the loop

MPI_Wtime: return the wall-clock time in seconds.

- gives timing per rank
- can reduce it to get the max, mean, …

MPI_Barrier: block untill all processes in the communicator has reached this routine

- all processes need to call it in the communicator

Example: computing pi

```
const int tag = 123;
if(rank == 0)
{
  double total_sum = sum;
  double other;
  for(int ii=1; ii<size; ++ii)
  {
    MPI_Recv(&other, 1, MPI_DOUBLE, ii, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    total_sum += other;
  }
  printf("pi is %.12g \n", sum * 4.0);
}
else
  MPI_Ssend(&sum, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);

MPI_Finalize();
return 0;
```

Collect and addd the sum from other ranks

Send partial sums to rank 0

Communication takes O(size) steps! Not ideal. HPC network is slower than main memory.

Blocking collective communications

Some very common collective operations have been implemented in MPI

These operations include:

- Reduce
- AllReduce
- Broadcast
- Gather
- AllGather
- Scatter
- Barrier
- AllToAll
- Scan
- ExScan
- Reduce_Scatter

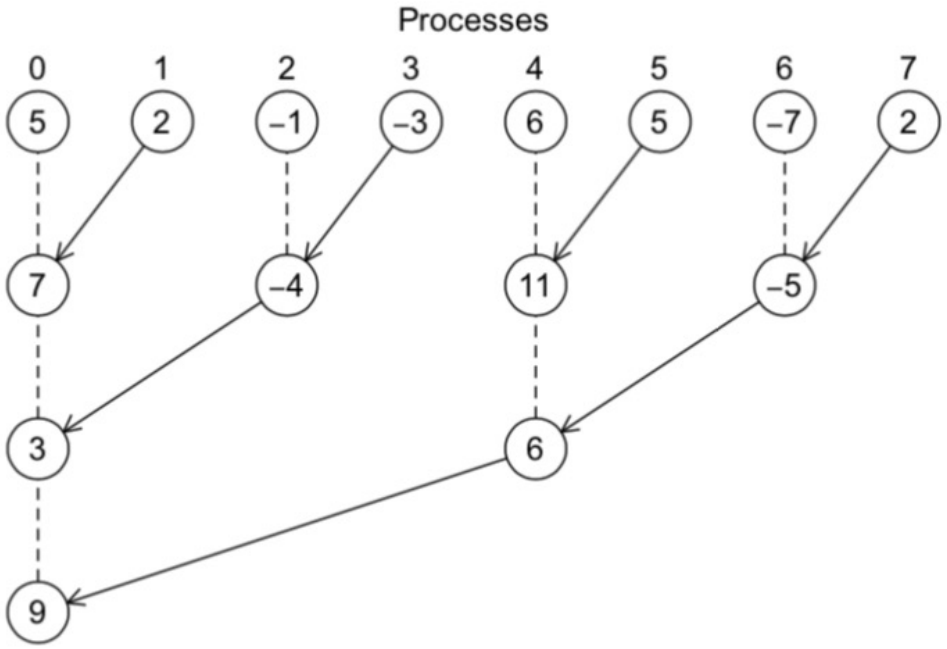
All to One
One to All
All to All
Other

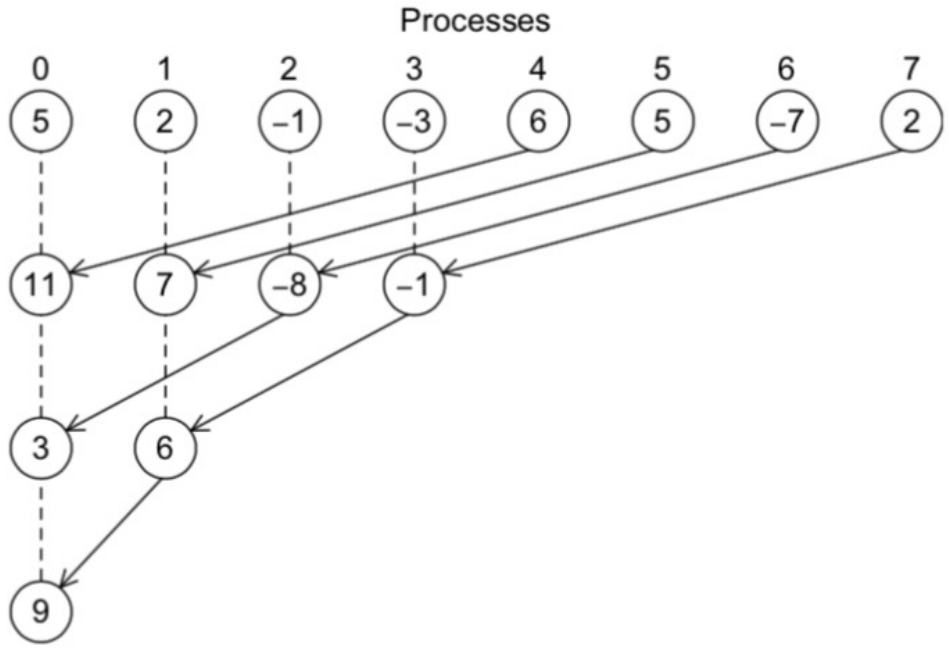
They involve **all ranks of a same communicator** and therefore must be called by all the ranks of the concerned communicator

Blocking: the operation is completed for the process once the function returns.

Reduction

There are better ways to do reduction, based on point-to-point communication





O(log(size)) rather than O(size)

More intermediate ranks are involved in the send and recv operations

Reduction

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

- sendbuf: input data to reduce
- recvbuf: output result (only for root)
- count: number of elements (per process) to reduce
- datatype: type of element to reduce
- op: operation to operate
- root: the rank id to which the result is output
- comm: communicator

This is a **collective** operation, meaning **ALL** processes in the communicator must call it

Data can be reduced in place to save memory, in which case recvbuf is input-output and sendbuf should be set to MPI_IN_PLACE

Reduction

MPI implements a few basic operations:

MPI_Op	Operation
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 exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

You can define a custom operation using MPI_OP_CREATE

The operation must be associative

Reduction

Reduction is called by all processes and send the result to rank 0

```
double total_sum = 0.0;

MPI_Reduce(&sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if(rank == 0)
  printf("pi is %.12g \n", sum * 4.0);

MPI_Finalize();
return 0;
```

only rank 0 has the result

Reduction

In place reduction

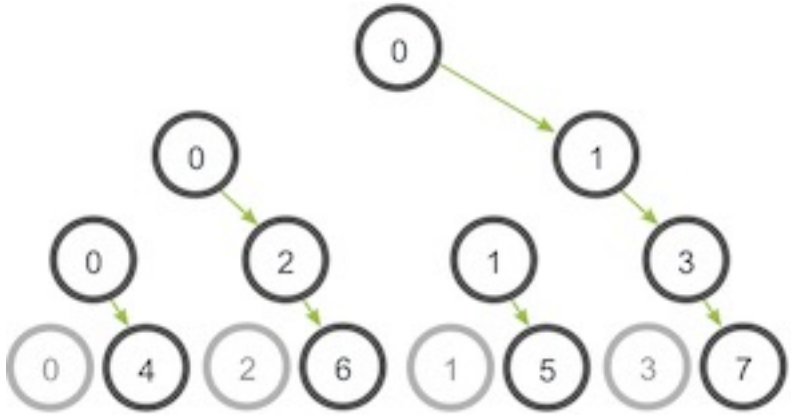
```
double total_sum = 0.0;

MPI_Reduce(rank ? &sum : MPI_IN_PLACE, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if(rank == 0) printf("pi is %.12g \n", sum * 4.0);

MPI_Finalize();
return 0;
```

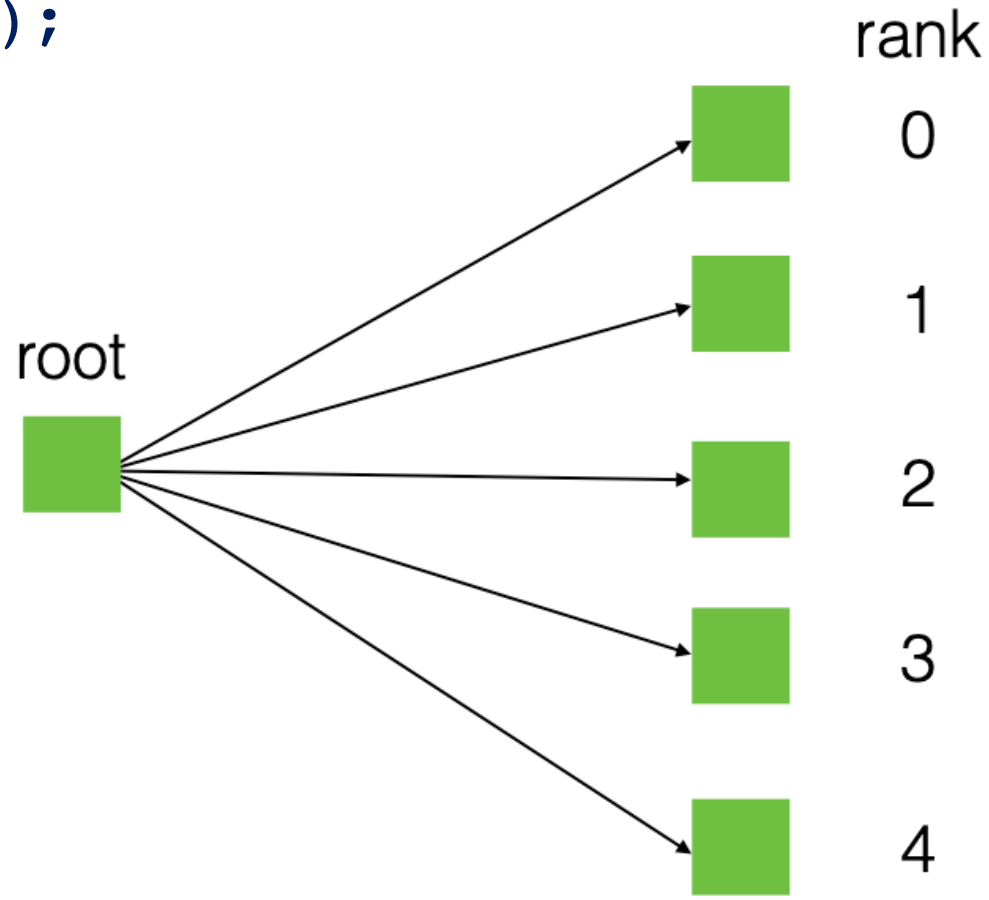
Broadcast

A broadcast sends data from the root rank to al ranks in the communicator

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm );
```

- Can be naively implemented by send and recv
- MPI makes use of a tree structure to obtain better performance



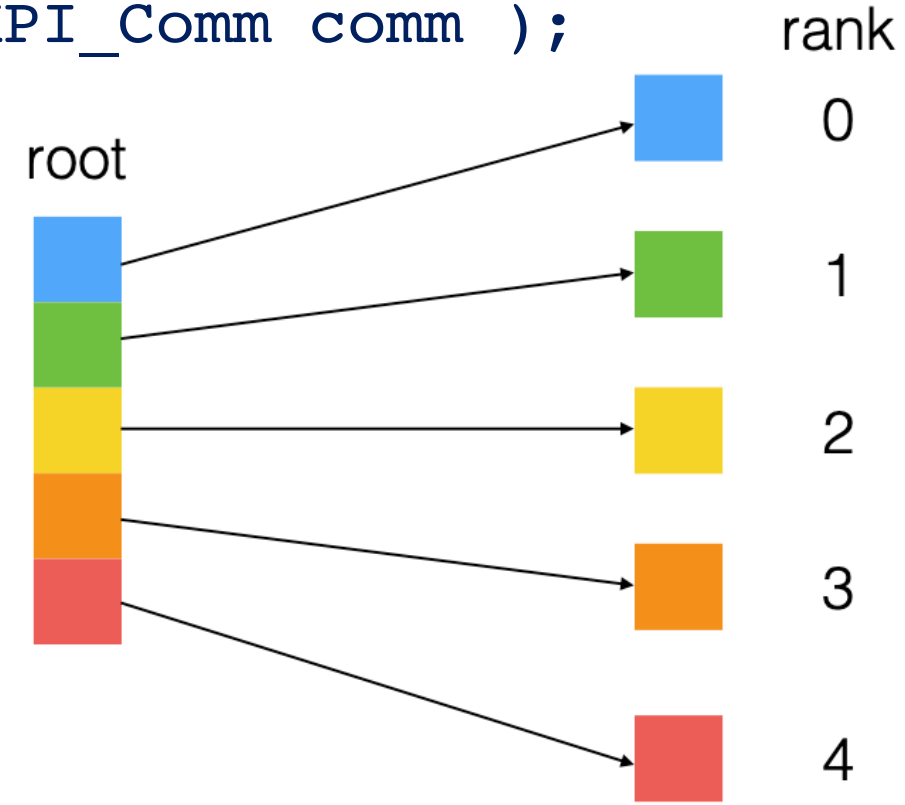


Scatter

Send data in separate chunks from the root rank to all ranks in the communicator

```
int MPI_Scatter( void *sendbuffer, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPIDatatype recvtype, int root, MPI_Comm comm );
```

- sendbuf, sendcount, sendtype only significant to root
- sendcount: number of elements sent to each process

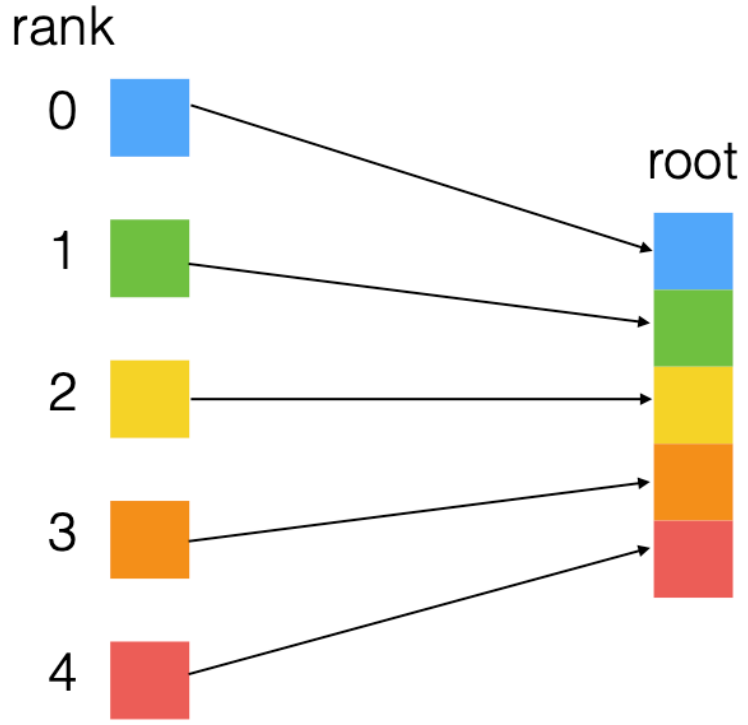


Gather

Collect chunks of data from all ranks in the communicator to the root rank (inverse of scatter)

```
int MPI_Gather( void *sendbuffer, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPIDatatype recvtype, int root, MPI_Comm comm );
```

- recvbuf, recvcount, recvtype: only significant to root
- sendcount: number of elements for a receive

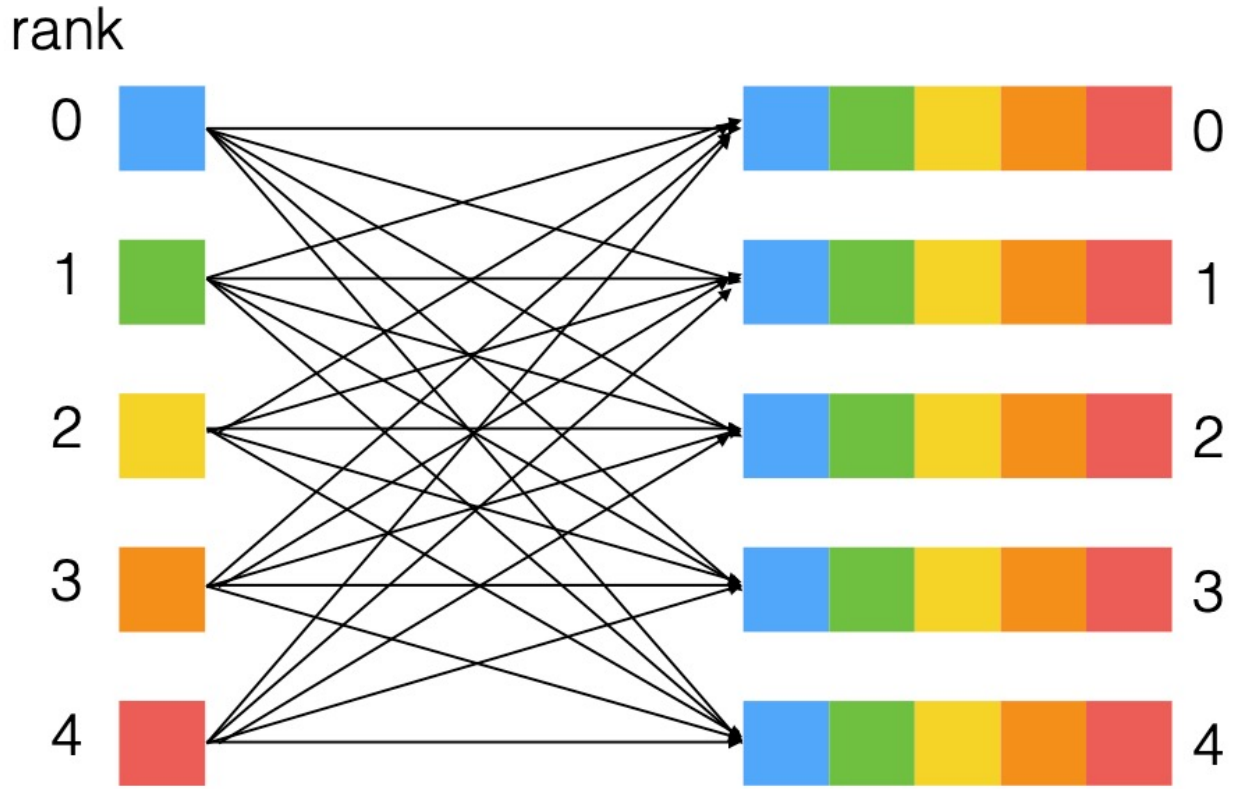


AllGather

Same as Gather but all ranks get the result

```
int MPI_AllGather( void *sendbuffer, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPIDatatype recvtype, MPI_Comm comm );
```

- recvcount: number of elements for a single receive

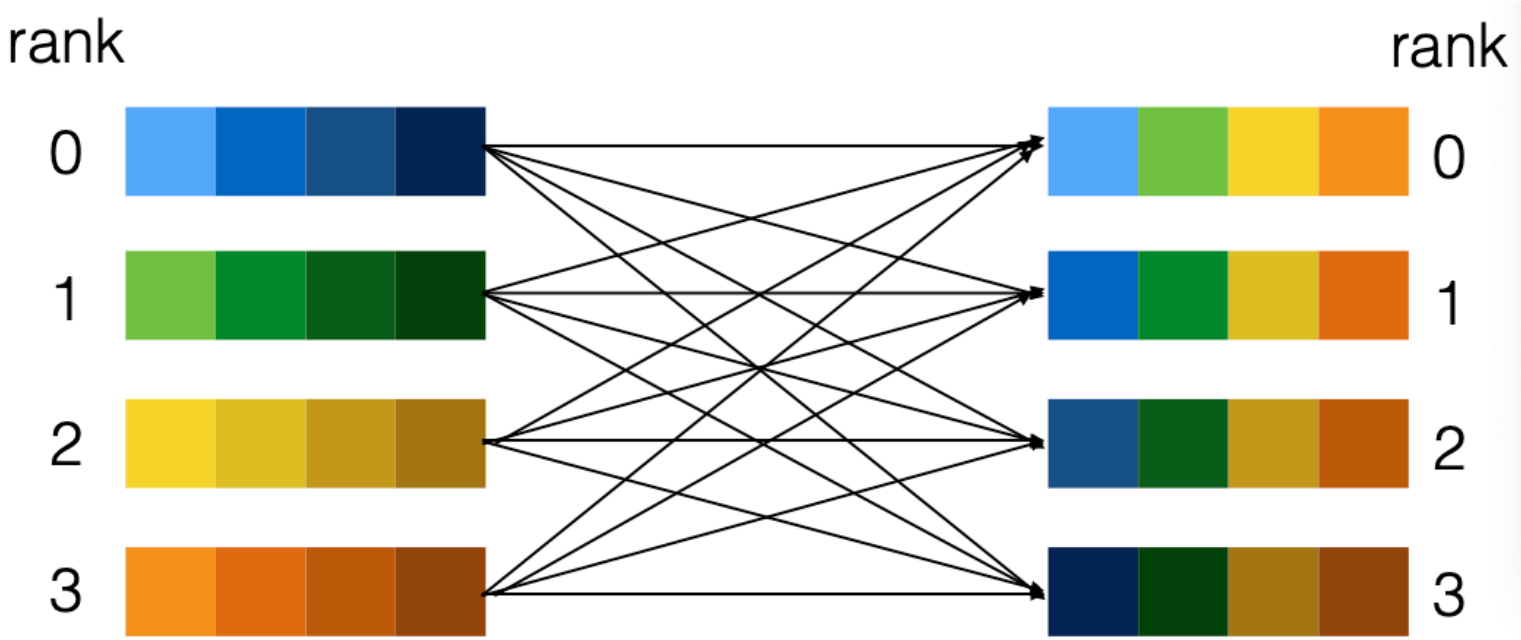


AllToAll

Shuffle the data between ranks: acts like a transpose

```
int MPI_Alltoall( void *sendbuffer, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPIDatatype recvtype, MPI_Comm comm );
```

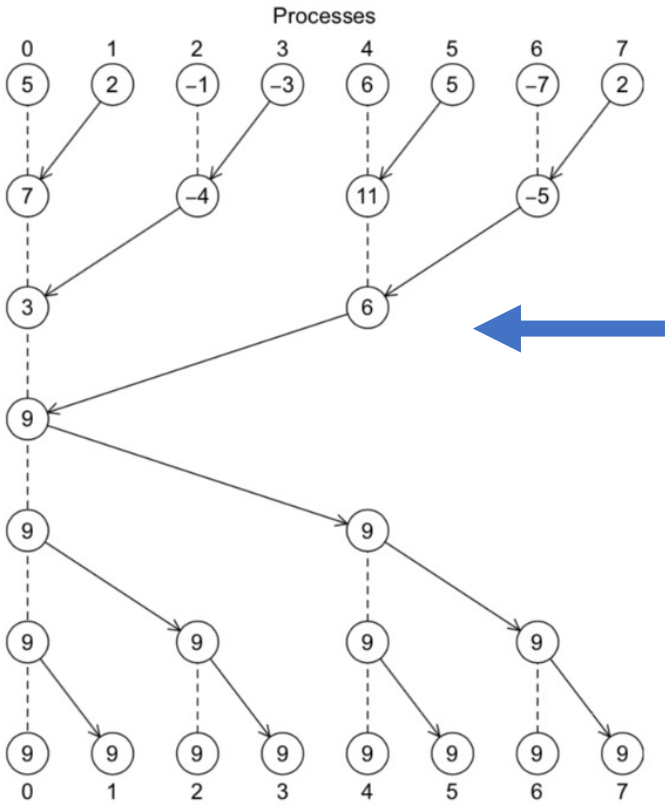
- Can be seen as an extension of AllGather, except that each rank has different data



AllReduce

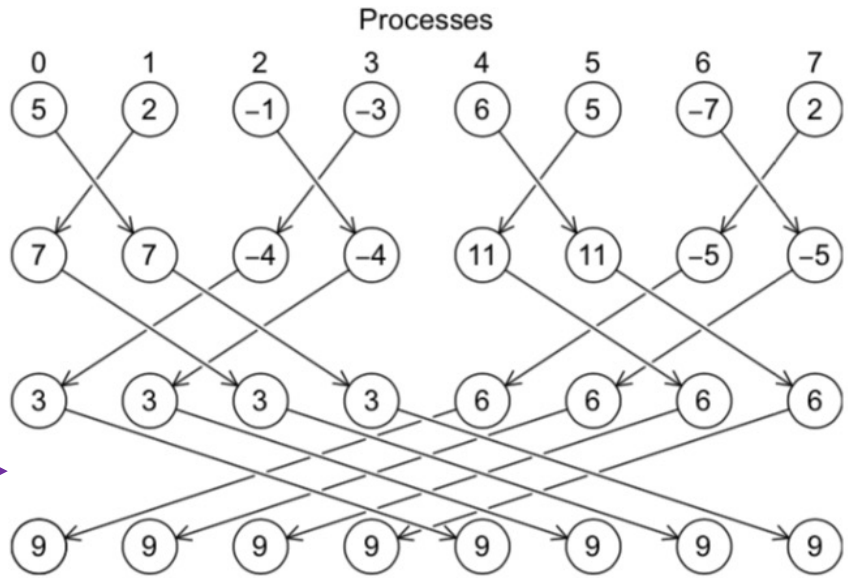
Reduce the data, and broadcast the result to all ranks

```
int MPI_Allreduce( const void *sendbuffer, void *
recvbuffer, int count, MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm );
```



Conceptually

Actually



Non-blocking collective

Non-blocking collective:

- the data buffer cannot be used right after the function returns
- the call returns directly but creates a MPI_Request object
- MPI_Wait and MPI_Test can be used to wait or check the completion

```
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Request
*request);

int MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request);

int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
MPI_Request *request);
……
```

Same names as blocking with a “I” in front of the operation.
Extra output argument of type MPI_Request at the end.

Non-blocking collective

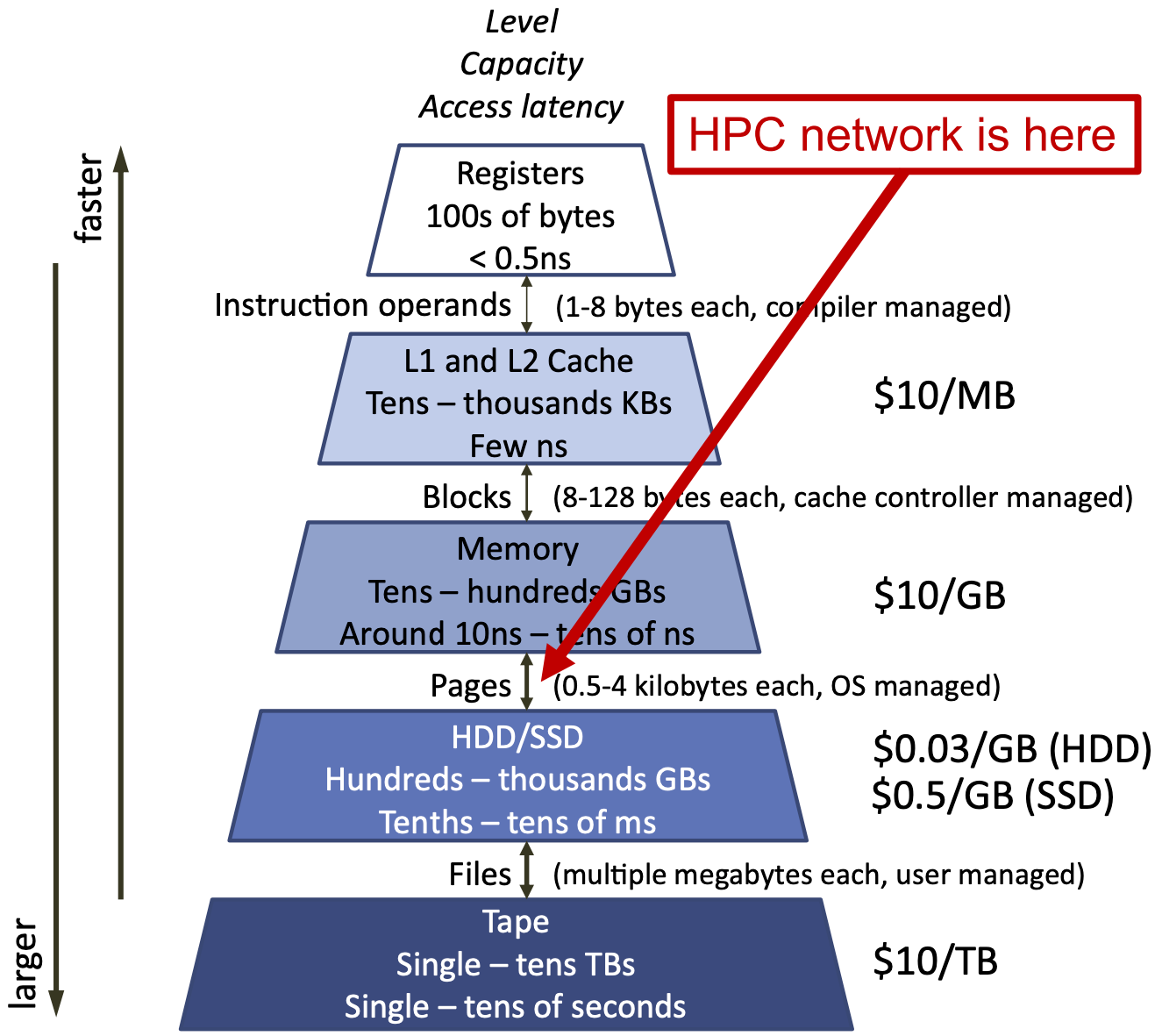
```
double val = (double) (rank * rank);
double sumVals = 0;

MPI_Request request;
MPI_Iallreduce(&val, &sumVals, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD,
&request);

doUnrelatedWork(); // this function overlaps with the reduction

MPI_Wait(&request, MPI_STATUS_IGNORE); // Need to wait for the result
if (rank == 0)
    printf("total sum is %g\n", sumVals);
```

MPI I/O

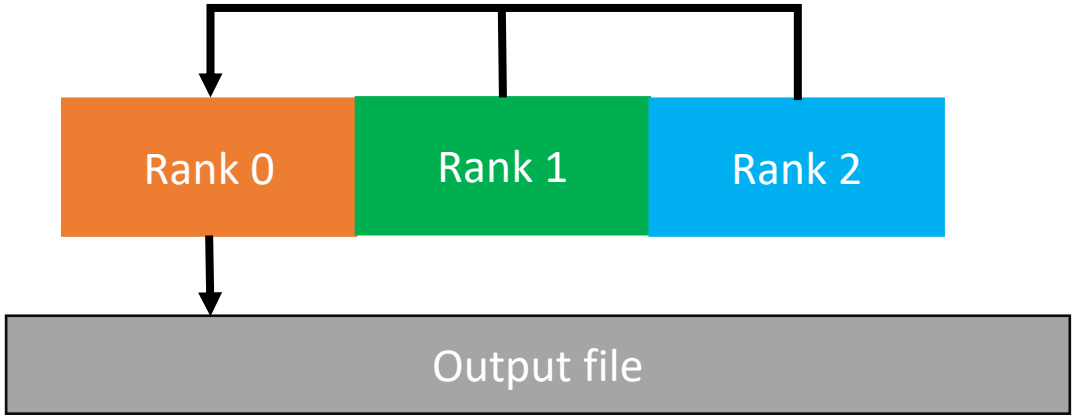
You will need to do Input and Output when

- postprocessing
- visualization
- checkpoint/restart

Regular I/O operations allow writing or reading files sequentially (one rank at a time)
Larger problems require dumping possibly very large files
This can be a limiting factor for performance



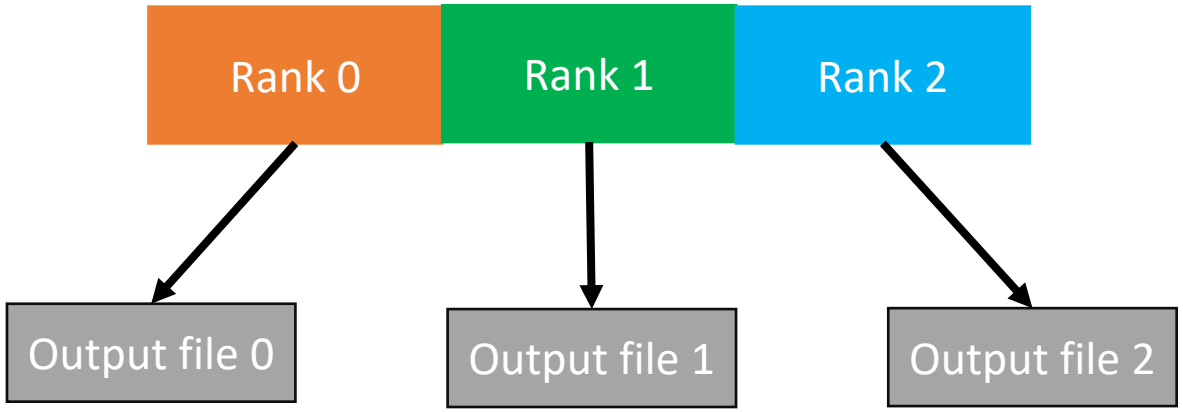
MPI I/O



Dump the data to the root processor through the HPC network

- A single file
- Not scalable ✖

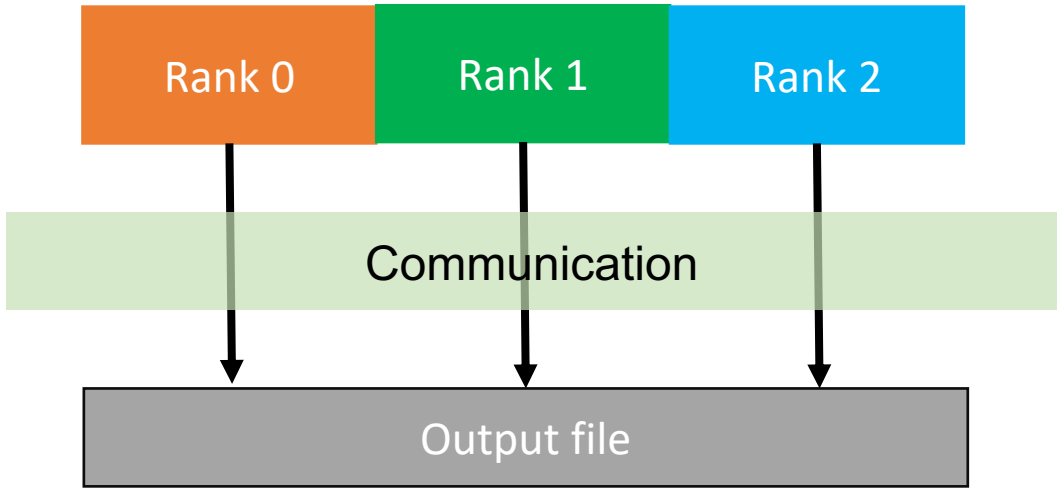
MPI I/O



Dump one file per rank

- Scalable
- Many small files, bad for HPC filesystem ✖
- Difficult for postprocessing ✖

MPI I/O



Dump a common file from many ranks

- Scalable
- Single common file

MPI I/O

MPI I/O is used in many mature libraries: HDF5, NETCDF, etc.

Logic is the same as the usual file management:

- Open/create a file handle

```
    int MPI_File_open(MPI_Comm comm, const char *filename, int amode,
MPI_Info info, MPI_File *fh);
```

- Read/write data

- Close the file

```
    int MPI_File_close(MPI_File *fh);
```

The opening and closing of the file are collective operations.

MPI I/O

amode: access mode, possible values:

- MPI_MODE_APPEND Set initial position of all file pointers to end of file
- MPI_MODE_CREATE Create the file if it does not exist
- MPI_MODE_DELETE_ON_CLOSE Delete file on close
- MPI_MODE_EXCL Error creating a file that already exists
- MPI_MODE_RDONLY Read only
- MPI_MODE_RDWR Reading and writing
- MPI_MODE_SEQUENTIAL File will only be accessed sequentially
- MPI_MODE_WRONLY Write only
- MPI_MODE_UNIQUE_OPEN File will not be concurrently opened elsewhere

Summary

- MPI is used to create parallel programs based on message passing
- We use SPMD model for most of the time
- There are 6 basic calls: `MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, MPI_Finalize`
- There are other point-to-point communication functions: buffered, non-blocking, etc.
- There are many collective communications that are handy and efficient
- There are MPI I/O functions, but we recommend using HDF5
- There are some topics not covered:
  - MPI communicator management by groups
  - one-sided communication
  - MPI derived data types
  - …

References: MPI standard document http://www.mpi-forum.org/docs/

Using MPI by Gropp, Lusk, and Skjellum, MIT Press, 1994