CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

"Where is the 'any' key?" — Homer Simpson, in response to the message, "Press any key"

Distributed memory computing using MPI

Shared memory is a good model
for a small number of processes.

When dealing with a large number of processors,
we need to view the memory as being distributed.

What this means:

Processors can no longer directly read and write
to another processor's memory

Instead processors exchange messages.

Programmed by the user explicitly.

Send + Receive

This can be done using MPI.

MPI is the standard for distributed memory computing.

Message Passing Interface

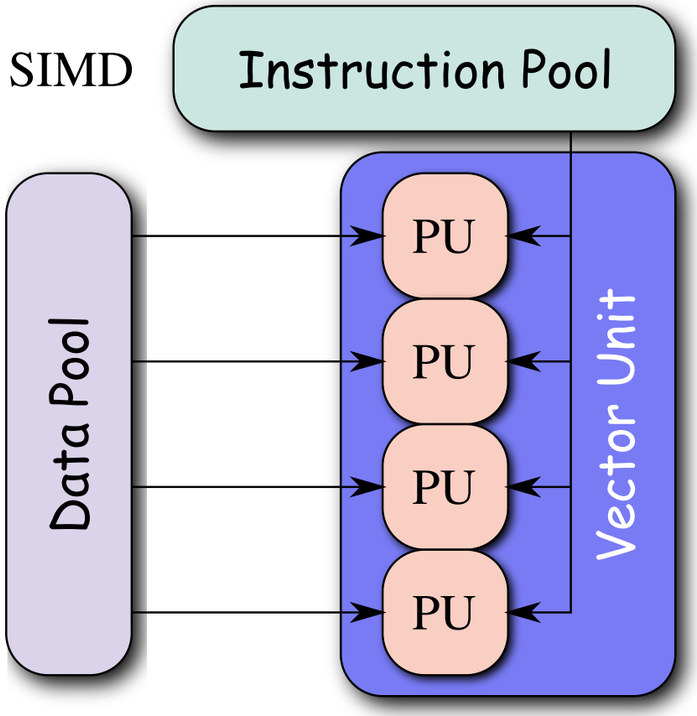
Flynn’s taxonomy

SIMT: one instruction is dispatched to multiple threads.

Warp on a GPU

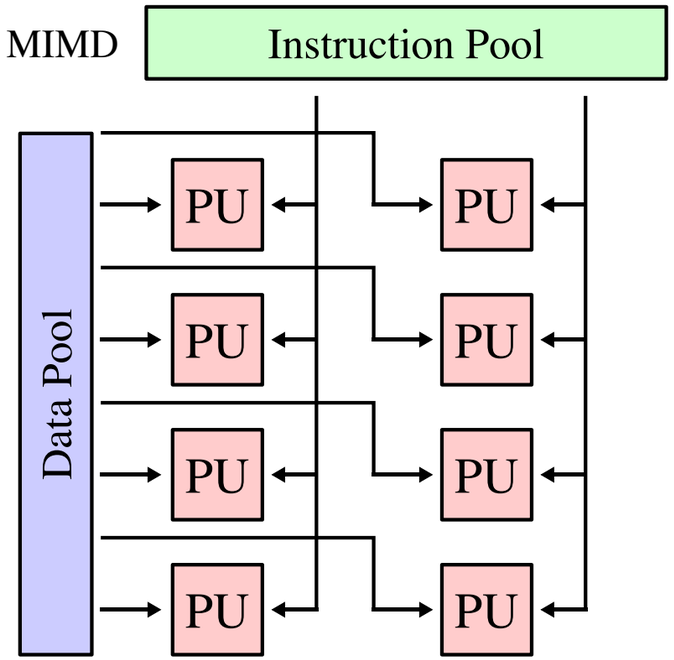
SIMD: same instruction run by different processing units using different data

Vector processing units



MIMD: multiple instructions, multiple data; multiple threads running different functions

Multicore threads

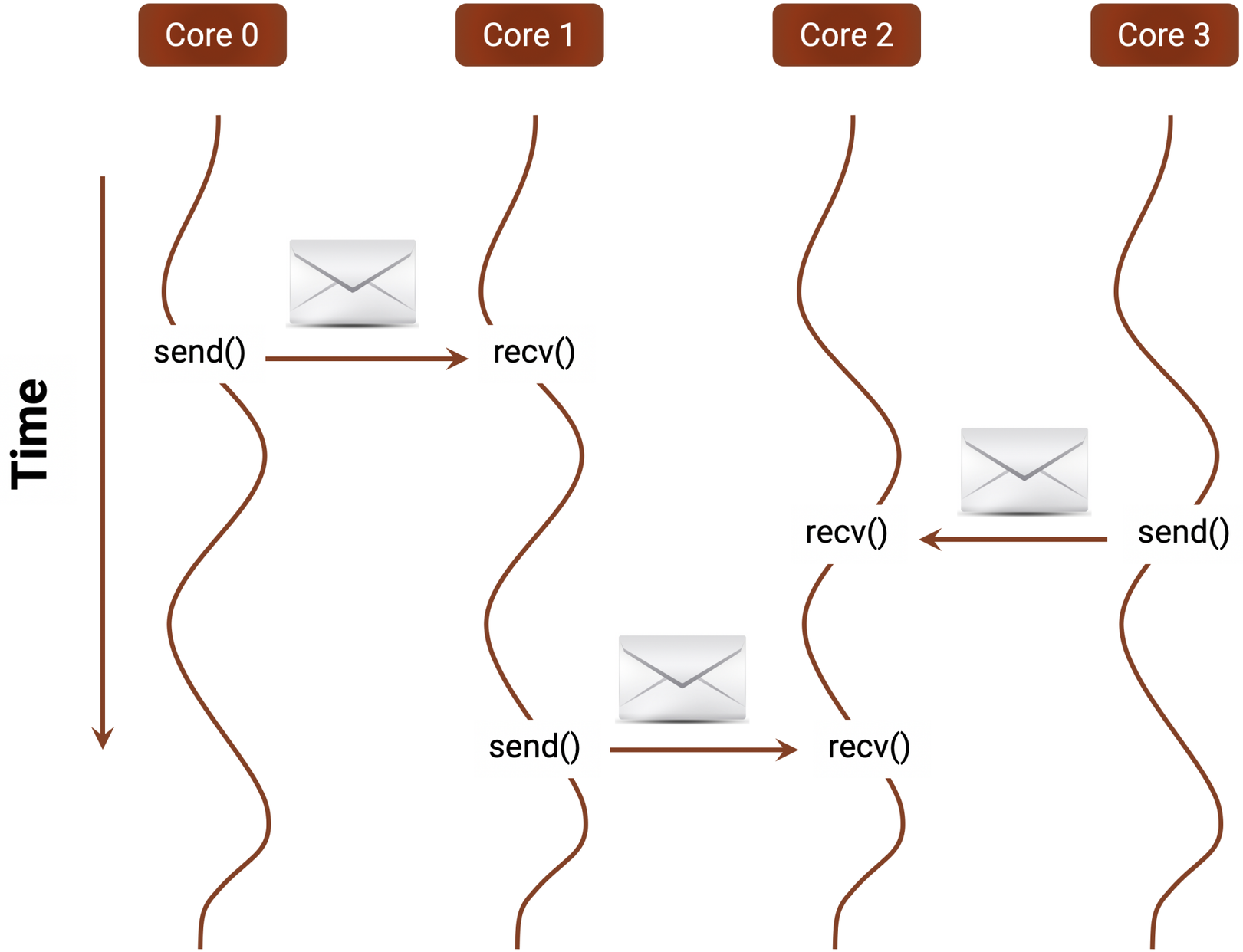


SPMD: this is our topic for today.

The same program runs on different processors.

Processors communicate through a network by exchanging messages or data in an explicit manner.

Each program has a unique ID or rank which is used to determine what computations the program should perform.


Core 0
Core 1
Core 2
Core 3
Time
send()
recv()
recv()
send()
send()
recv()


Let's do this

Where can I get MPI?

- OpenMPI: www.open-mpi.org
  (what we use on `icme-gpu`)
- MVAPICH: mvapich.cse.ohio-state.edu
- MPICH: www.mpich.org

What computer can I use it with?

You can test MPI using a **multicore** computer.

Each process runs on its own core.

You can run this on your **laptop** or **icme-gpu**.

Compiling

Compile with:
`mpic++`

Header:
`mpi.h`

Running is more complicated than usual.

You need to start multiple programs (processes) on multiple computers.

You need to make sure all processes are killed or terminated at the end.

$\Rightarrow$ `sbatch`, `mpirun`

`sbatch`: batch submission

Your job will be placed in a queue and run when resources are available.

The output is written to a file.

```
#!/bin/bash

#SBATCH --time=00:30:00
#SBATCH --partition=CME
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --gres=gpu:4
#SBATCH --job-name=fp
#SBATCH --output=fp-%j.out
#SBATCH --error=fp-%j.err

mpirun ./mpi_hello
```

Or:

```
#!/bin/bash

#SBATCH --time=00:30:00
#SBATCH --partition=CME
#SBATCH --nodes=1
#SBATCH --gres=gpu:4
#SBATCH --job-name=fp
#SBATCH --output=fp-%j.out
#SBATCH --error=fp-%j.err

mpirun -n 4 ./mpi_hello
```

With `mpirun`, we are running the program `mpi_hello` four times.

Example:

```
#!/bin/bash

#SBATCH --partition=CME
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4

mpirun hostname
```

Output:

```
icmet01
icmet01
icmet01
icmet01
```

mpi_hello.cpp

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

// How many processes are running
int numprocs;
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
// What's my rank?
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Which node am I running on?
int len;
char hostname[MPI_MAX_PROCESSOR_NAME];
MPI_Get_processor_name(hostname, &len);

printf("Hello from rank %2d running on node: %s\n", rank, hostname);

if (rank == MASTER)
  printf("MASTER process: the number of MPI processes is: %2d\n", numprocs);

MPI_Finalize();
```

Computing π using MPI

```
$ salloc -N 1 -n 4 --partition=CME mpirun mpi_pi_send
MPI process  0 has started on icmet01 [total number of processors 4]
MPI process  3 has started on icmet01 [total number of processors 4]
MPI process  1 has started on icmet01 [total number of processors 4]
MPI process  2 has started on icmet01 [total number of processors 4]
   After  2000000 throws, average value of pi = 3.14071400
   After  4000000 throws, average value of pi = 3.14121500
   After  6000000 throws, average value of pi = 3.14127600
   After  8000000 throws, average value of pi = 3.14107900
   After 10000000 throws, average value of pi = 3.14110760
   After 12000000 throws, average value of pi = 3.14155867
   After 14000000 throws, average value of pi = 3.14151857
   After 16000000 throws, average value of pi = 3.14160475
   After 18000000 throws, average value of pi = 3.14165844
   After 20000000 throws, average value of pi = 3.14163500

Exact value of pi: 3.1415926535897
```

mpi_pi_send.cpp

```
if (rank != MASTER)
{
    int tag = i;
    int rc = MPI_Send(&my_pi, 1, MPI_DOUBLE,
                      MASTER, tag, MPI_COMM_WORLD);

    if (rc != MPI_SUCCESS)
        printf("%d: Send failure on round %d\n", rank, tag);
}
else
{
    ...
}
```

```
int MPI_Send(void *smessage, int count,
    MPI_Datatype datatype, int dest,
    int tag,
    MPI_Comm comm)
```

`smessage` buffer which contains the data
`count` number of elements to be sent
`datatype` data type of entries
`dest` rank of the target process
`tag` message tag which (used to distinguish messages)
`comm` communicator used for the communication

MPI datatype	C datatype	C++ datatype
MPI::CHAR	`char`	`char`
MPI::SHORT	`signed short`	`signed short`
MPI::INT	`signed int`	`signed int`
MPI::LONG	`signed long`	`signed long`
MPI::LONG_LONG	`signed long long`	`signed long long`
MPI::SIGNED_CHAR	`signed char`	`signed char`
MPI::UNSIGNED_CHAR	`unsigned char`	`unsigned char`
MPI::UNSIGNED_SHORT	`unsigned short`	`unsigned short`
MPI::UNSIGNED	`unsigned int`	`unsigned int`
MPI::UNSIGNED_LONG	`unsigned long`	`unsigned long int`
MPI::UNSIGNED_LONG_LONG	`unsigned long long`	`unsigned long long`
MPI::FLOAT	`float`	`float`
MPI::DOUBLE	`double`	`double`
MPI::LONG_DOUBLE	`long double`	`long double`
MPI::BOOL		`bool`
MPI::COMPLEX		`Complex<float>`
MPI::DOUBLE_COMPLEX		`Complex<double>`
MPI::LONG_DOUBLE_COMPLEX		`Complex<long double>`
MPI::WCHAR	`wchar_t`	`wchar_t`
MPI::BYTE		
MPI::PACKED		

```
if (rank != MASTER) { ... } else {
    int tag = i; double pisum = 0;

    for (int n = 1; n < numprocs; n++) {
        double pirecv; MPI_Status status;
        int rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                          tag, MPI_COMM_WORLD, &status);
        if (rc != MPI_SUCCESS)
            printf("%d: Receive failure on round %d\n", rank, tag);
        /* Running total of pi */
        pisum += pirecv;
    }
}
```

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

Mostly same as before. One new argument:

`status` data structure that contains information about the message that was received

```
if (rank != MASTER) {
    int tag = i;
    int rc = MPI_Send(&my_pi, 1, MPI_DOUBLE,
                      MASTER, tag, MPI_COMM_WORLD);
} else {
    for (int n = 1; n < numprocs; n++) {
        int rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                          tag, MPI_COMM_WORLD, &status);
        pisum += pirecv;
    }
}
```

Rules and order

Each `Send` must be matched with a corresponding `Recv`.

Order: messages are received in the order in which they have been sent.

If a sender sends two messages of the same type one after another to the same receiver, the MPI runtime system ensures that the first message sent is always received first.

Collective communications

What we have discussed so far is **point-to-point communication,** that is one process communicates with another process.

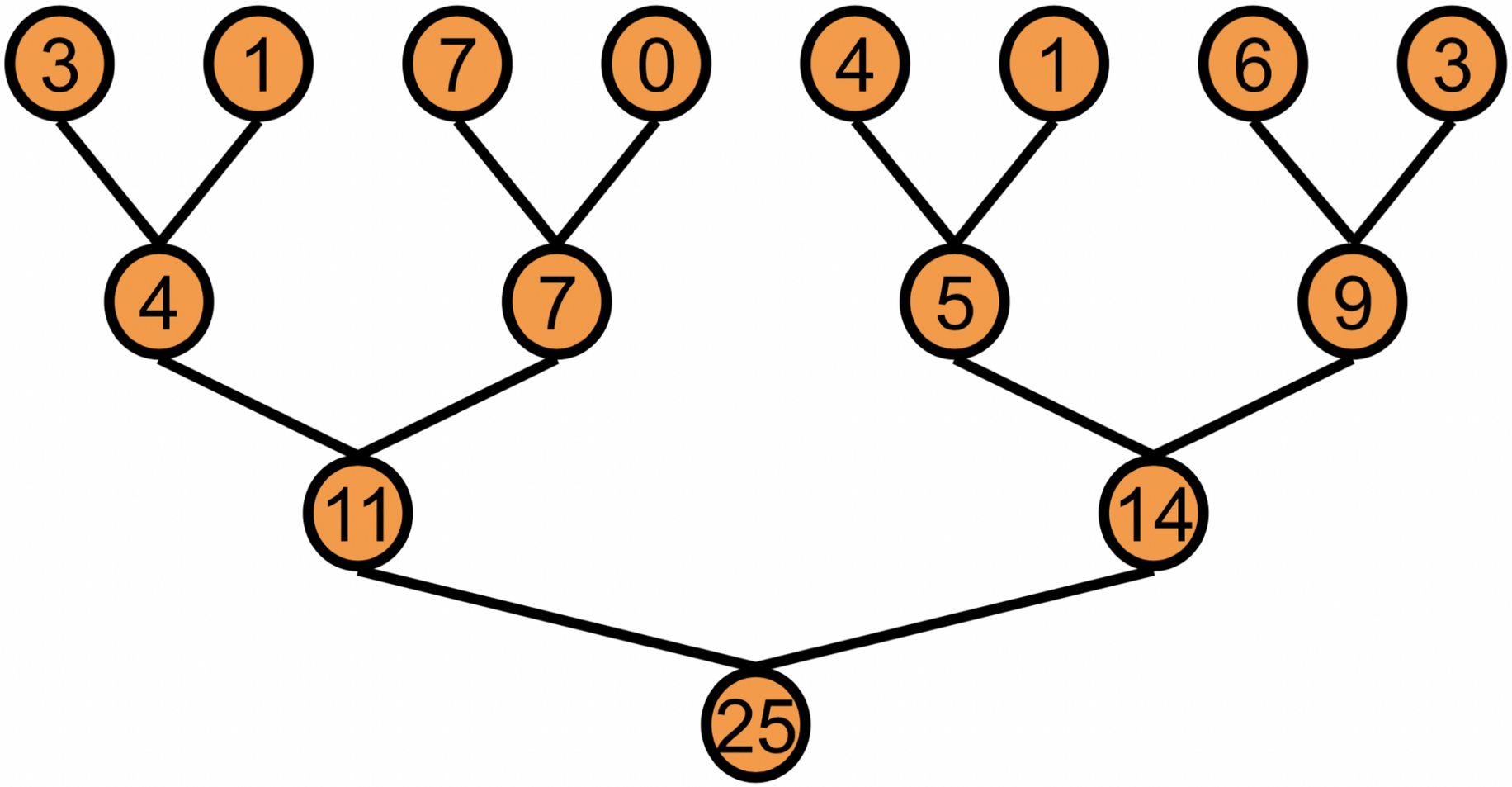
Let’s say that we have a group of processes that need to exchange data.

For example we want to do a **reduction.**

This is called a collective communication, i.e., multiple processes need to communicate.

For best performance, we need to orchestrate the communication.

Simply having each process send its data to the master node is inefficient.


3
1
7
0
4
1
6
3
4
7
5
9
11
14
25

Computer network = network of highways

Each highway has a number of lanes and a maximum traffic it can support. This is the bandwidth.

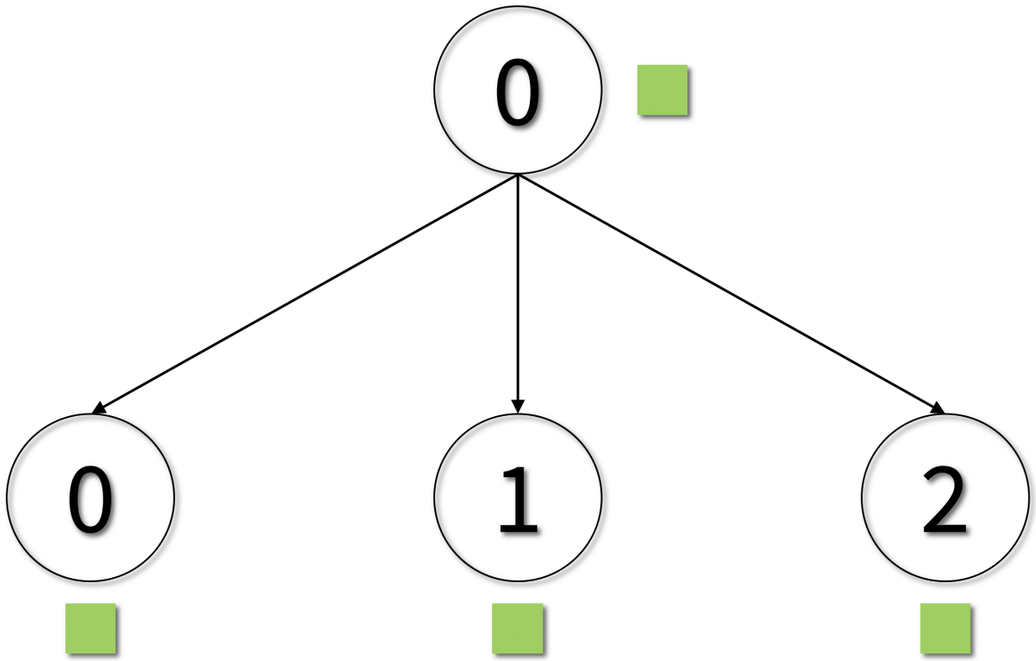
Depending on the network topology, there is an optimal algorithm to route the messages in order to minimize the total wall clock time of the collective communication.

Three key issues:

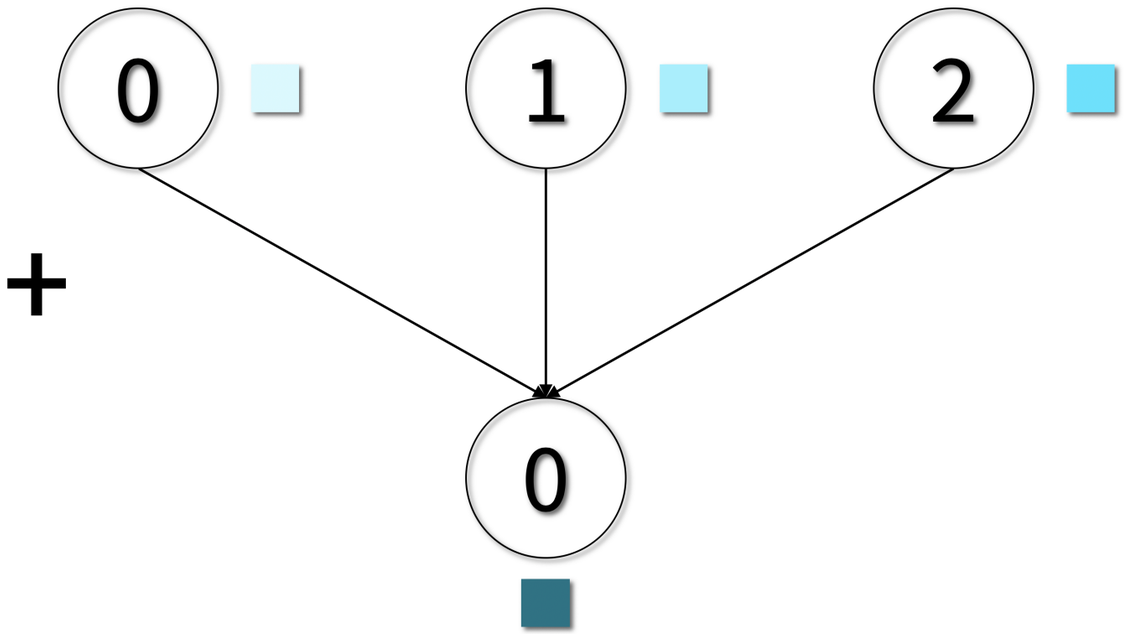
1. These communication algorithms can be complicated.
2. They depend on the network topology.
3. There are relatively few collective communication patterns that get reused over and over again.

Let's review the main functions

```
MPI_Bcast(&buffer,count,datatype,root,comm)
```



```
MPI_Reduce(&sendbuf,&recvbuf,count,datatype,op,root,comm)
```



MPI Reduction Operation		C Data Types	Fortran Data Type
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical AND	integer	logical
MPI_BAND	bit-wise AND	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical OR	integer	logical
MPI_BOR	bit-wise OR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer	logical
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double	real, complex,double precision
MPI_MINLOC	min value and location	float, double and long double	real, complex, double precision

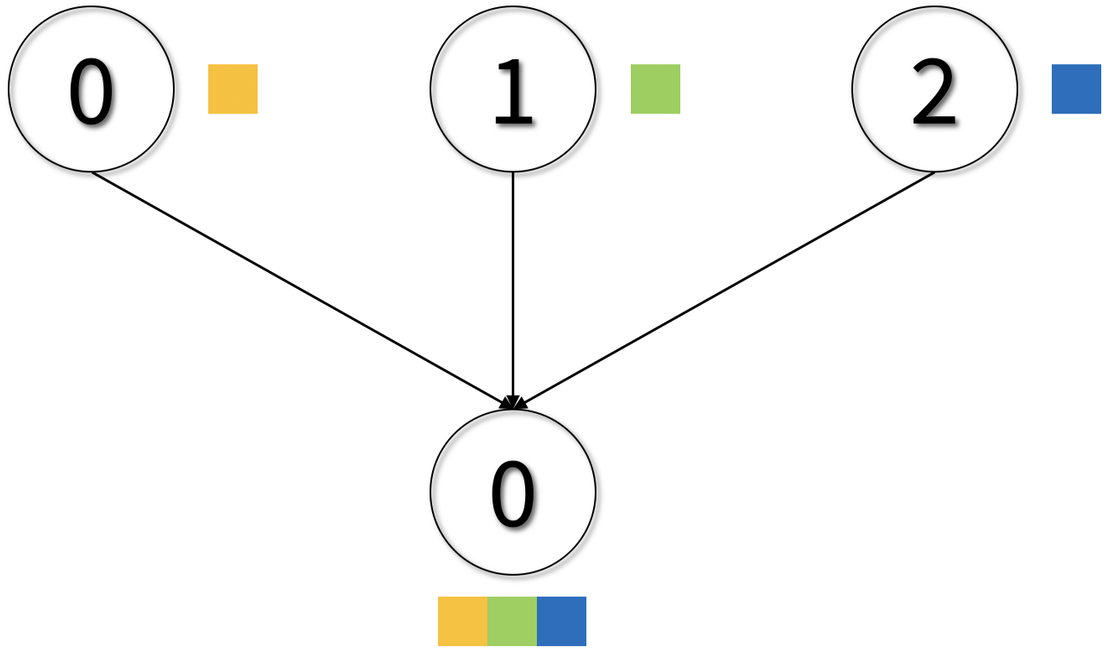
```
Using 16 tasks to scan 40000000 numbers...
Done.
Largest prime is 39999983.
Total number of primes found: 2433654
Wall clock time elapsed: 3.19 seconds
```

mpi_prime.cpp

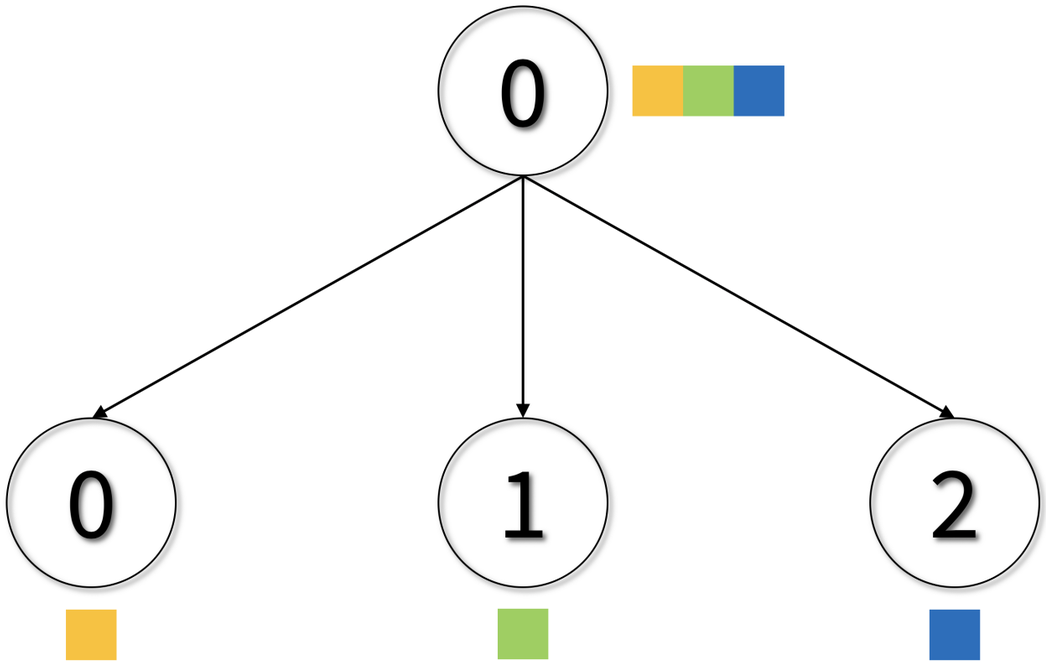
```
for (int n = mystart; n <= LIMIT; n += stride) {
    if (IsPrime(n)) {
        pc++;         // found a prime
        foundone = n; // last prime that we have found
    }
}
// Total number of primes found by all processes: MPI_SUM
MPI_Reduce(&pc, &pcsum, 1, MPI_INT, MPI_SUM, MASTER, MPI_COMM_WORLD);

// The largest prime that was found by all processes: MPI_MAX
MPI_Reduce(&foundone, &maxprime, 1, MPI_INT, MPI_MAX, MASTER, MPI_COMM_WORLD);
```

```
MPI_Gather(&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm)
```



```
MPI_Scatter(&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)
```

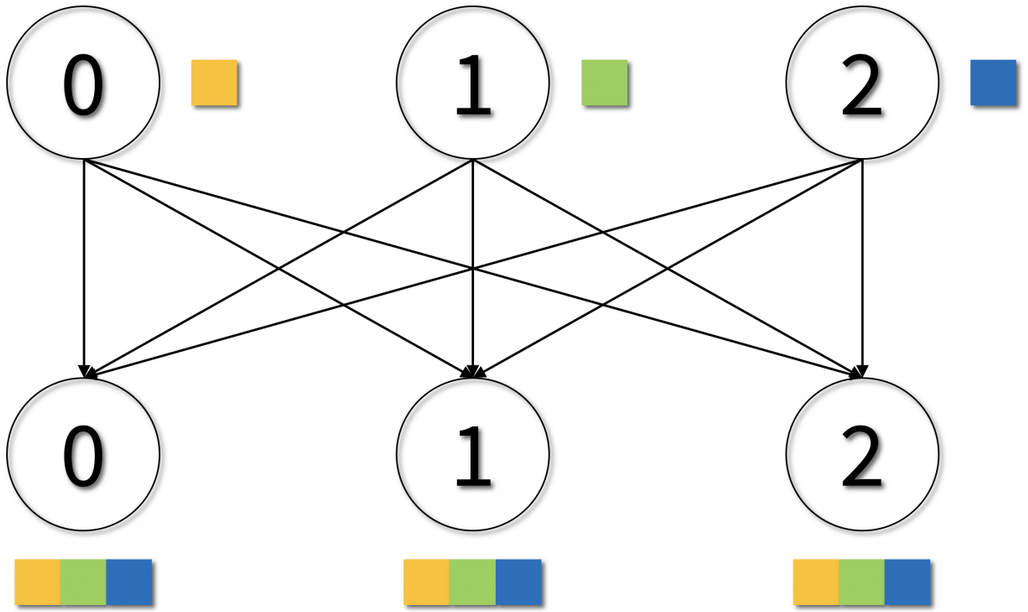


Final project

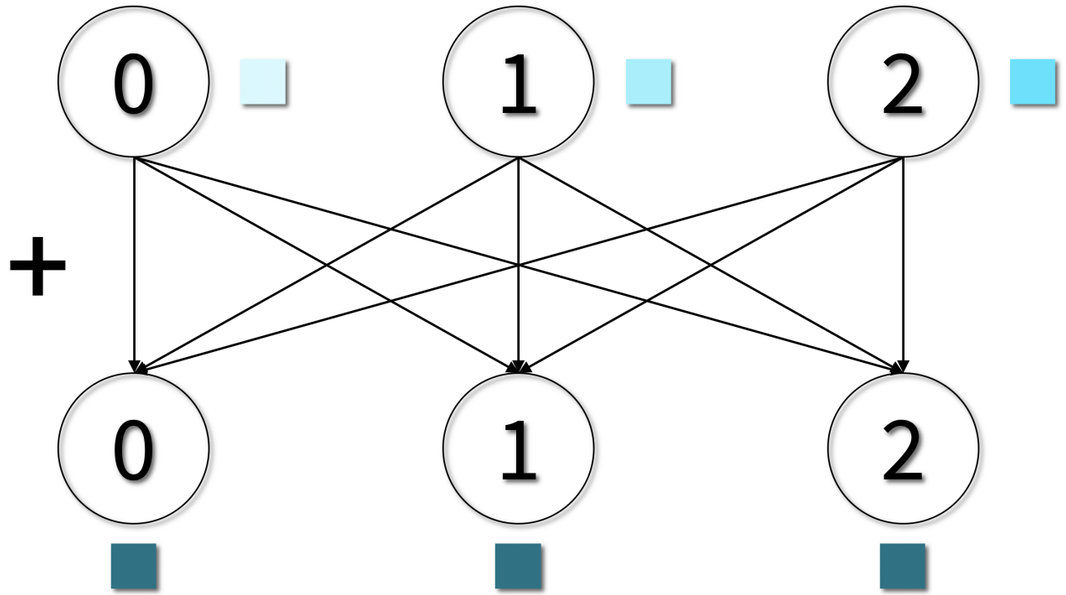
Rank 0 reads MNIST data from disk

`MPI_Scatter` the images to all processors

```
MPI_Allgather(&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm)
```



```
MPI_Allreduce(&sendbuf,&recvbuf,count,datatype,op,comm)
```



Final project

$$J(p) = \frac{1}{N} \sum_{i=1}^{N} \text{error}^{(i)}(y_i, \hat{y}_i)$$

$$p \leftarrow p - \alpha \nabla J_p$$

Each process has a partial ∇J_p

`MPI_Allreduce` to get the complete gradient on all processors

```
Rank   0 has values:   8071   1347    839   2390   5379
Rank   1 has values:   8542   1166   3510   7451   2227
Rank   2 has values:   4341   4158   6383   1559   1566
Rank   3 has values:   2437   1848   4815   1564   2616

Rank  0 has the lowest value of 839

Rank  0 has received the value:   839
Rank  1 has received the value:   839
Rank  2 has received the value:   839
Rank  3 has received the value:   839
```

proc_min_value.cpp

```c
int localres[2], globalres[2];
localres[0] = localarr[0]; // Minimum
for (int i = 1; i < locn; i++)
    if (localarr[i] < localres[0])
        localres[0] = localarr[i];

// The second entry is the rank of this process.
localres[1] = rank;

// MPI_MINLOC: like the operator min. The difference is that it
// takes as input two numbers; the first one is used to determine the
// minimum value. The second number just goes along for the ride.
// MPI_2INT: type for 2 integers.
MPI_Allreduce(localres, globalres, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
```

```
MPI_Alltoall(&sendbuf,sendcount,sendtype,&recvbuf,recvcnt,recvtype,comm)
```