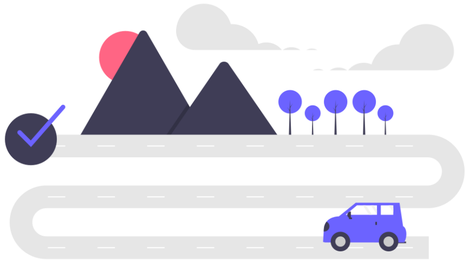
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

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

Stanford University

Recap



- Multicore programming: C++ threads, OpenMP
- CUDA programming; performance and optimization, profiling
- Final project

Distributed memory computing using MPI

Why distributed memory programming?

- Shared memory is a good model for a small number of processes.
  - Easy to program
  - High-performance
  - But puts a significant burden on the operating system and libraries to maintain memory consistency.
- Distributed memory: user to responsible for moving and copying data between separate memories.
  - Puts less burden on the underlying hardware and OS
  - Scales to a large number of processes

What this means…

- Processes/threads can no longer just read and write to the same memory locations
  - `P0: x = 1`
  - `P1: y = x`
  - `P2: z = f(y)`

Messages

- Instead, processes **exchange messages.**
- Programmed by the user explicitly.
- Send + Receive:
  - P0: sends to P1
  - P1: receives from P0
- Two-sided communication

MPI

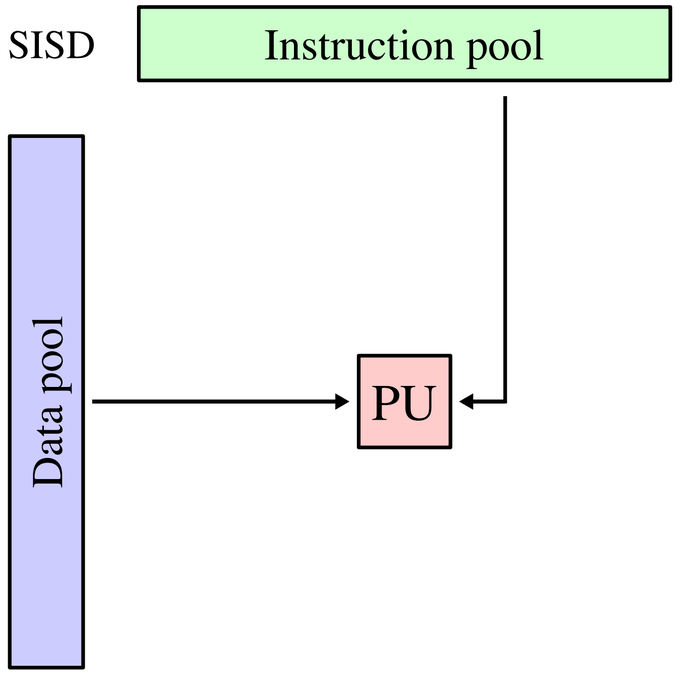
- This can be done using MPI.
- MPI is the standard for distributed memory computing.
- **Message Passing Interface**

Flynn’s taxonomy

- A parallel computer can be characterized as a collection of processing elements that can communicate and cooperate to solve large problems fast.
- Many details need to be specified: the processing elements’ number and complexity, the interconnection network’s structure between the processing elements, and the coordination of the work between the processing elements…
- Flynn’s taxonomy gives a simple model for such a classification.
- This taxonomy characterizes parallel computers according to the global control and the resulting data and control flows.

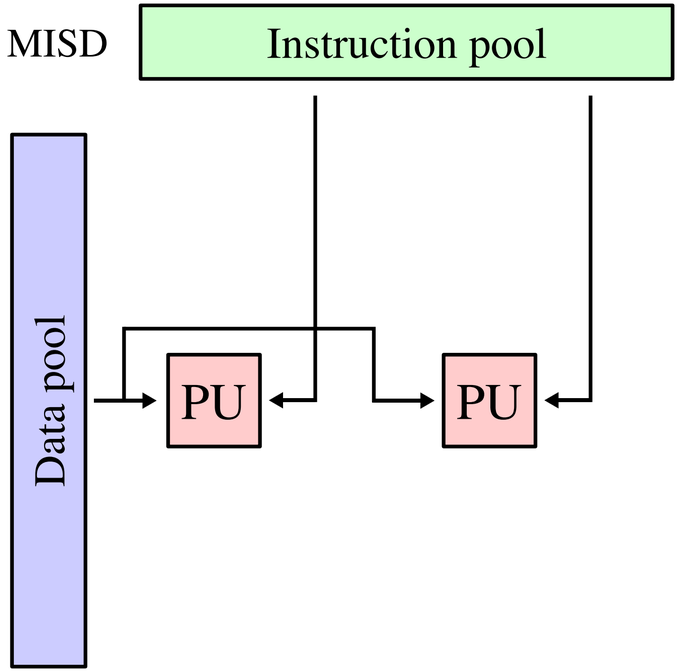
Single-Instruction, Single-Data (SISD)

- There is one processing element that has access to a single program and data storage.
  - In each step, the processing element loads an instruction and the corresponding data and executes the instruction.
  - The result is stored back in the data storage.
- SISD is the conventional sequential computer according to the von Neumann model.



Multiple-Instruction, Single-Data (MISD)

- Multiple processing elements have a private program memory, but there is only one common access to a single global data memory.
- In each step, each processing element obtains the same data element from the data memory and loads an instruction from its private program memory.
- These possibly different instructions are then executed in parallel by the processing elements using the previously obtained (identical) data element as an operand.
- **Very uncommon.**

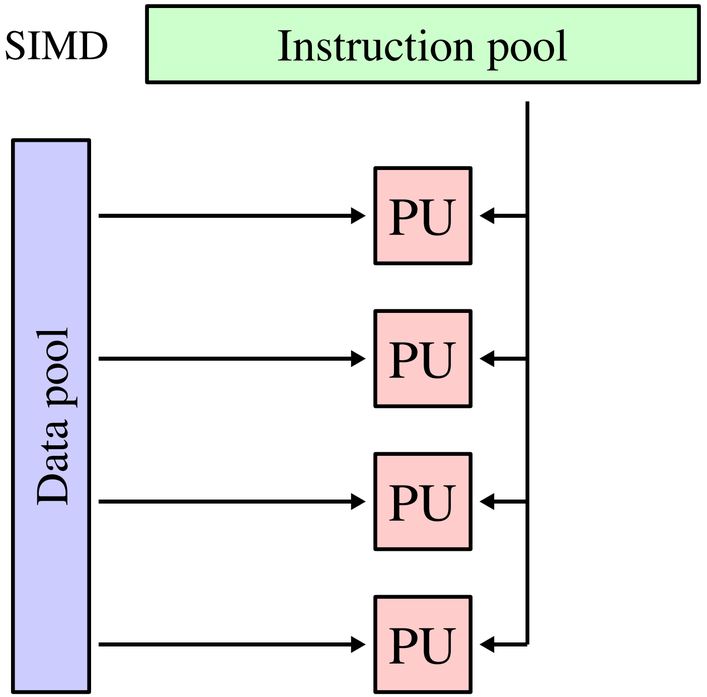


Single-Instruction, Multiple-Threads (SIMT)

- One instruction is dispatched to multiple threads.
- This is the model used for example on a GPU.
  - Warp in a CUDA thread-block.

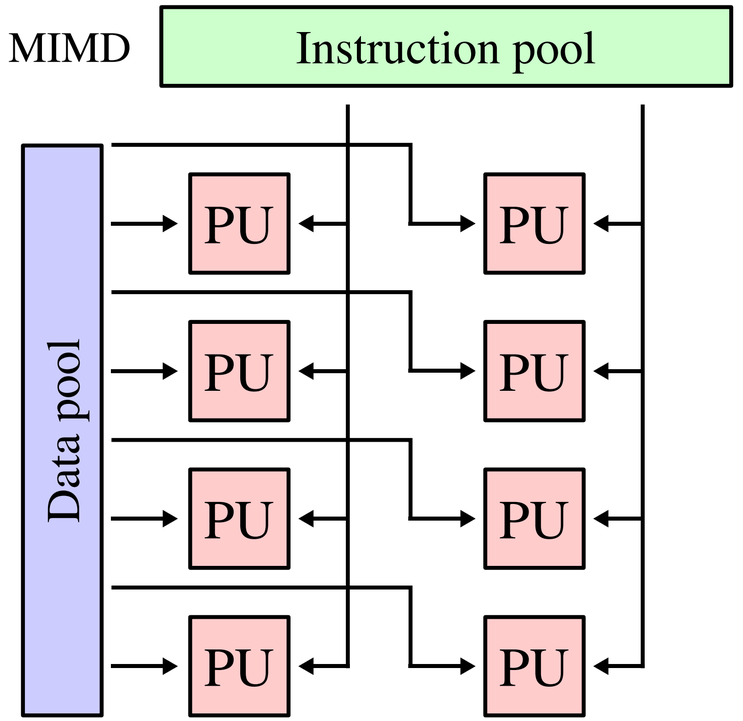
Single-Instruction, Multiple-Data (SIMD)

- There are multiple processing elements that have private access to (shared or distributed) data memory.
- But there is only one program memory from which a special control processor fetches and dispatches instructions.
- In each step, each processing element obtains from the control processor the **same instruction** and loads a separate data element through its private data access on which the instruction is performed.
- Thus, the instruction is synchronously applied in parallel by all processing elements to different data elements.
- Applications: multimedia applications or computer graphics algorithms to generate realistic three-dimensional views of computer-generated environments.



Multiple-Instruction, Multiple-Data (MIMD)

- There are multiple processing elements with separate instruction and data access to a (shared or distributed) program and data memory.
- In each step, each processing element loads a separate instruction and a separate data element, applies the instruction to the data element, and stores a possible result back into the data storage.
- The processing elements work asynchronously with each other.
- **A multicore processor** is an example of the MIMD model.



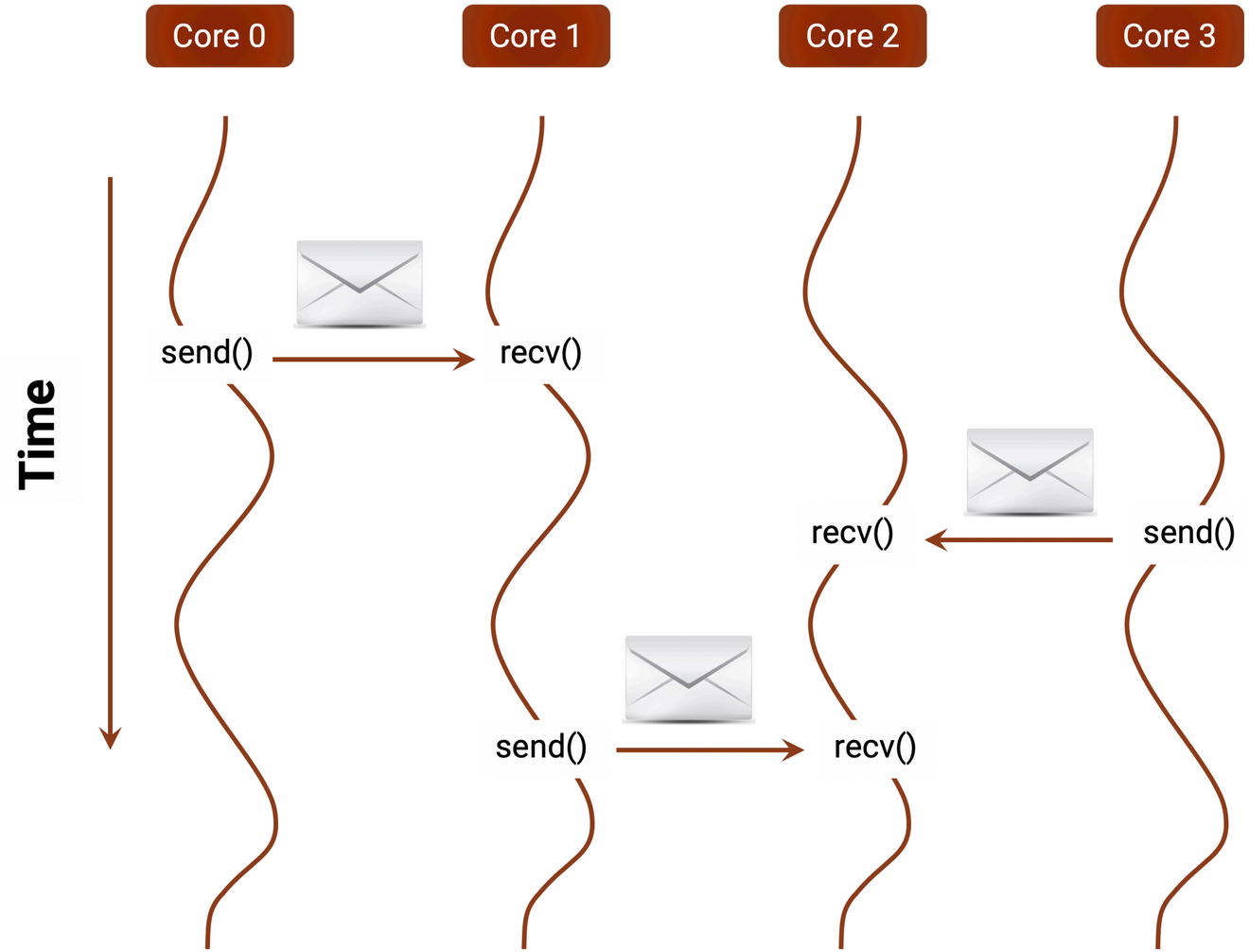
Single program, multiple data streams (SPMD)

- Sub-classification under MIMD.
- Multiple autonomous processors simultaneously execute the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data.
- Also termed single process, multiple data.
- SPMD is the **most common style of parallel programming.**

SPMD programming

- MPI
- The **same program** runs on different processors.
- Processors communicate through a network by **explicitly exchanging messages or data**.
- Each program has a **unique ID or rank,** which determines what computations the program should perform.

Example of program execution



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?

- Computer cluster
- Cloud computing
- You can also 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`

Hello World

- Logic is different from what we have seen previously.
- We run multiple copies of the same program.
- The difference is that each program gets assigned a different rank.
- Based on the rank, the program decides what to do.

```
// Some MPI magic to get started
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);

// Only one processor will do this
if (rank == MAIN) {
  printf("MAIN process: the number of MPI processes is: %2d\n", numprocs);
}

// Close down all MPI magic
MPI_Finalize();
```

Output

Example running with 2 nodes:

```
srun -n 2 --mpi=pmi2 -p CME ./mpi_hello
```

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 2 --mpi=pmi2 -p CME ./mpi_hello
Hello from rank  0 running on node: icmet01
MAIN process: the number of MPI processes is:  2
Hello from rank  1 running on node: icmet01
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

Communication

- In most cases, MPI programs will require processes to communicate.
- Let's estimate π by throwing darts at a unit square.
- By computing how often the dart falls into the unit circle we can approximate π.

- Each MPI_Send must be matched with an MPI_Recv.
- An if statement is used to distinguish processes that send data from the main process that receives data.
- We give an example of a wildcard: MPI_ANY_SOURCE that matches any source.

```
for (int i = 0; i < ROUNDS; i++)
{
  /* All tasks calculate pi using the dartboard algorithm */
  double my_pi = DartBoard(DARTS);

  int tag = i; // Message tag is set to the iteration count
  if (rank != MAIN)
  { // Workers send my_pi to MAIN
    // Message tag is set to the iteration count
    int rc = MPI_Send(&my_pi, 1, MPI_DOUBLE, MAIN, tag, MPI_COMM_WORLD);
    if (rc != MPI_SUCCESS)
      printf("%d: send error | %d\n", rank, i);
  }
  else
  {
    // MAIN receives messages from all workers
    double pisum = 0;
    for (int n = 1; n < numprocs; n++)
    {
      /* Message source is set to the wildcard MPI_ANY_SOURCE: */
      int rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, tag,
                        MPI_COMM_WORLD, &status);
      if (rc != MPI_SUCCESS)
        printf("%d: rcv error | %d; msg %d\n", rank, i, n);

      /* Running total of pi */
      pisum += pirecv;
    }
    /* MAIN calculates the average value of pi for this iteration */
    double pi = (pisum + my_pi) / numprocs;
    /* MAIN calculates the average value of pi over all iterations */
    avepi = ((avepi * i) + pi) / (i + 1);
    printf("   After %8d throws, average value of pi = %10.8f\n",
           (DARTS * (i + 1) * numprocs), avepi);
  }
}
```

Output

- 4 programs are running using 4 separate processes.
- Process 1, 2, and 3 send their data to process 0.
- Process 0 receives the data and calculate the final estimate of π.

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 4 --mpi=pmi2 -p CME ./mpi_pi_send
MPI process  2 has started on icmet01 [total number of processors 4]
MPI process  3 has started on icmet01 [total number of processors 4]
MPI process  0 has started on icmet01 [total number of processors 4]
   After  2000000 throws, average value of pi = 3.14268400
   After  4000000 throws, average value of pi = 3.14286000
   After  6000000 throws, average value of pi = 3.14260333
   After  8000000 throws, average value of pi = 3.14187700
   After 10000000 throws, average value of pi = 3.14184560
   After 12000000 throws, average value of pi = 3.14160500
   After 14000000 throws, average value of pi = 3.14171229
   After 16000000 throws, average value of pi = 3.14163700
   After 18000000 throws, average value of pi = 3.14200844
   After 20000000 throws, average value of pi = 3.14187180

Exact value of pi: 3.1415926535897
MPI process  1 has started on icmet01 [total number of processors 4]
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

MPI_Send

```
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 data types

This is required because data types may be stored (represented) differently on each computer node.

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		

MPI_Recv

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

`rmessage` buffer which receives the data

`source` rank of the process sending the message

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

Matching MPI messages

- Each `Send` must be matched with a corresponding `Recv`.
- Order guaranteed by MPI: 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.

Example where order is unspecified

```
// example to demonstrate the order of receive operations
if (rank == 0)
{
  MPI_Send(sendbuf1, count, MPI_INT, 2, tag, MPI_COMM_WORLD);
  MPI_Send(sendbuf2, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
}
else if (rank == 1)
{
  MPI_Recv(recvbuf1, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
  MPI_Send(recvbuf1, count, MPI_INT, 2, tag, MPI_COMM_WORLD);
}
else if (rank == 2)
{
  MPI_Recv(recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
           &status);
  MPI_Recv(recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
           &status);
}
```

In this example, the order in which the messages from 0 and 1 are received by process 2 is unspecified.

How to run MPI codes on icme-gpu

- As before, we can use sbatch.
- This launches a script on the remote nodes.

Example script

- `mpirun` must be used to start MPI codes.
- Other options: `srun, mpiexec`
- `–n 4` specifies the number of processes to use.

```
#!/bin/bash
#SBATCH --partition=CME
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4

mpirun -n 4 ./mpi_hello
```

Blocking commands

- Instead of running asynchronously, you can run using blocking commands.
- The output appears directly in your terminal window.
- The command completes only when the job completes.

salloc

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ salloc -n 4 -p CME mpirun -n 4 ./mpi_hello
salloc: Granted job allocation 18384
salloc: Waiting for resource configuration
salloc: Nodes icmet01 are ready for job
Hello from rank  3 running on node: icmet01
Hello from rank  0 running on node: icmet01
MAIN process: the number of MPI processes is:  4
Hello from rank  1 running on node: icmet01
Hello from rank  2 running on node: icmet01
salloc: Relinquishing job allocation 18384
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

- Allocates resources
- Runs the command immediately
- Blocks until the command completes.

srun

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 4 --mpi=pmi2 -p CME ./mpi_hello
Hello from rank  0 running on node: icmet01
MAIN process: the number of MPI processes is:  4
Hello from rank  1 running on node: icmet01
Hello from rank  2 running on node: icmet01
Hello from rank  3 running on node: icmet01
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

With the option `--mpi=pmi2`, it launches the executable in a way similar to `mpirun`.

Final project

`-n 4 -G 4`

```
darve@icme-gpu1:~/final_project$ srun -n 4 -G 4 --mpi=pmi2 -p CME ./main
Number of MPI processes         4
Number of CUDA devices          4
Grading mode off
Number of neurons              32
Number of epochs                1
Batch size                     32
Regularization      9.9999997e-05
Learning rate               0.001
The sequential code is not run
The debug option is off
Loading training data
Training data information:
Size of x_train, N =  60000
Size of label_train = 60000
Loading testing data

Start Parallel Training
Time for Parallel Training: 1.14894 seconds
Precision on validation set for parallel training =   0.7316666841506958
Precision on testing set for parallel training =   0.7175999879837036
darve@icme-gpu1:~/final_project$
```

Combining MPI and openMP

- Each MPI program can run a multithreaded process.
- MPI + openMP can provide improved performance compared to pure MPI programs.

```c
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int numprocs;
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int len;
  char hostname[MPI_MAX_PROCESSOR_NAME];
  MPI_Get_processor_name(hostname, &len);

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    printf("host %s, rank %1d/%1d, thread ID %d/%d\n", hostname, rank,
    numprocs, tid, nthreads);
  }

  MPI_Finalize();
  return 0;
}
```

Pure openMP

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 1 --ntasks-per-node=1 --cpus-per-task=4 -p CME ./mpi_openmp
host icmet01, rank 0/0, thread ID 1/3
host icmet01, rank 0/0, thread ID 3/3
host icmet01, rank 0/0, thread ID 0/3
host icmet01, rank 0/0, thread ID 2/3
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

Pure distributed MPI

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 4 --ntasks-per-node=1 --cpus-per-task=1 --threads-per-core=1 -p CME ./mpi_openmp
srun: job 18719 queued and waiting for resources
srun: job 18719 has been allocated resources
host icmet01, rank 0/3, thread ID 0/0
host icmet04, rank 3/3, thread ID 0/0
host icmet02, rank 1/3, thread ID 0/0
host icmet03, rank 2/3, thread ID 0/0
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

Hybrid: 4 nodes; 1 process per node; 4 threads per process

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 4 --ntasks-per-node=1 --cpus-per-task=4 -p CME ./mpi_openmp
host icmet02, rank 1/3, thread ID 2/3
host icmet02, rank 1/3, thread ID 1/3
host icmet02, rank 1/3, thread ID 0/3
host icmet02, rank 1/3, thread ID 3/3
host icmet03, rank 2/3, thread ID 1/3
host icmet03, rank 2/3, thread ID 0/3
host icmet03, rank 2/3, thread ID 2/3
host icmet03, rank 2/3, thread ID 3/3
host icmet04, rank 3/3, thread ID 1/3
host icmet04, rank 3/3, thread ID 0/3
host icmet04, rank 3/3, thread ID 2/3
host icmet04, rank 3/3, thread ID 3/3
host icmet01, rank 0/3, thread ID 0/3
host icmet01, rank 0/3, thread ID 1/3
host icmet01, rank 0/3, thread ID 2/3
host icmet01, rank 0/3, thread ID 3/3
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

Hybrid: 1 node; 4 processes; 4 threads per process

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 4 --ntasks-per-node=4 --cpus-per-task=4 -p CME ./mpi_openmp
host icmet02, rank 1/3, thread ID 2/3
host icmet02, rank 1/3, thread ID 1/3
host icmet02, rank 1/3, thread ID 0/3
host icmet02, rank 1/3, thread ID 3/3
host icmet02, rank 2/3, thread ID 2/3
host icmet02, rank 2/3, thread ID 0/3
host icmet02, rank 2/3, thread ID 3/3
host icmet02, rank 2/3, thread ID 1/3
host icmet02, rank 0/3, thread ID 2/3
host icmet02, rank 0/3, thread ID 3/3
host icmet02, rank 0/3, thread ID 0/3
host icmet02, rank 0/3, thread ID 1/3
host icmet02, rank 3/3, thread ID 0/3
host icmet02, rank 3/3, thread ID 2/3
host icmet02, rank 3/3, thread ID 1/3
host icmet02, rank 3/3, thread ID 3/3
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

Hybrid: 2 nodes; 2 processes per node; 4 threads per process

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -N 2 --ntasks-per-node=2 --cpus-per-task=4 -p CME ./mpi_openmp
host icmet02, rank 2/3, thread ID 0/3
host icmet02, rank 2/3, thread ID 1/3
host icmet02, rank 2/3, thread ID 3/3
host icmet02, rank 2/3, thread ID 2/3
host icmet02, rank 3/3, thread ID 1/3
host icmet02, rank 3/3, thread ID 2/3
host icmet02, rank 3/3, thread ID 3/3
host icmet02, rank 3/3, thread ID 0/3
host icmet01, rank 1/3, thread ID 0/3
host icmet01, rank 1/3, thread ID 3/3
host icmet01, rank 1/3, thread ID 2/3
host icmet01, rank 1/3, thread ID 1/3
host icmet01, rank 0/3, thread ID 2/3
host icmet01, rank 0/3, thread ID 3/3
host icmet01, rank 0/3, thread ID 1/3
host icmet01, rank 0/3, thread ID 0/3
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

- 6 nodes
- 4 processes per node
- 4 threads per process
- 96 threads

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -N 6 --ntasks-per-node=4 --cpus-per-task=4 -p CME ./mpi_openmp
host icmet03, rank 9/23, thread ID 0/3
host icmet03, rank 9/23, thread ID 2/3
host icmet03, rank 9/23, thread ID 3/3
host icmet03, rank 9/23, thread ID 1/3
host icmet05, rank 17/23, thread ID 3/3
host icmet05, rank 17/23, thread ID 0/3
host icmet05, rank 17/23, thread ID 2/3
host icmet05, rank 17/23, thread ID 1/3
host icmet06, rank 21/23, thread ID 0/3
host icmet06, rank 21/23, thread ID 2/3
host icmet06, rank 21/23, thread ID 3/3
host icmet06, rank 21/23, thread ID 1/3
host icmet01, rank 1/23, thread ID 0/3
host icmet01, rank 1/23, thread ID 1/3
host icmet01, rank 1/23, thread ID 3/3
host icmet01, rank 1/23, thread ID 2/3
host icmet03, rank 10/23, thread ID 1/3
host icmet03, rank 10/23, thread ID 0/3
host icmet03, rank 10/23, thread ID 3/3
host icmet03, rank 10/23, thread ID 2/3
host icmet04, rank 13/23, thread ID 2/3
host icmet04, rank 13/23, thread ID 1/3
host icmet04, rank 13/23, thread ID 0/3
host icmet04, rank 13/23, thread ID 3/3
host icmet05, rank 18/23, thread ID 0/3
host icmet05, rank 18/23, thread ID 2/3
host icmet05, rank 18/23, thread ID 3/3
host icmet05, rank 18/23, thread ID 1/3
host icmet06, rank 22/23, thread ID 1/3
host icmet06, rank 22/23, thread ID 0/3
host icmet06, rank 22/23, thread ID 2/3
host icmet06, rank 22/23, thread ID 3/3
host icmet02, rank 5/23, thread ID 3/3
host icmet02, rank 5/23, thread ID 1/3
host icmet02, rank 5/23, thread ID 0/3
host icmet02, rank 5/23, thread ID 2/3
host icmet01, rank 2/23, thread ID 1/3
host icmet01, rank 2/23, thread ID 3/3
host icmet01, rank 2/23, thread ID 0/3
host icmet01, rank 2/23, thread ID 2/3
host icmet02, rank 7/23, thread ID 0/3
host icmet02, rank 7/23, thread ID 2/3
host icmet02, rank 7/23, thread ID 1/3
host icmet02, rank 7/23, thread ID 3/3
host icmet04, rank 14/23, thread ID 0/3
host icmet04, rank 14/23, thread ID 3/3
host icmet04, rank 14/23, thread ID 2/3
host icmet04, rank 14/23, thread ID 1/3
host icmet05, rank 16/23, thread ID 0/3
host icmet05, rank 16/23, thread ID 2/3
host icmet05, rank 16/23, thread ID 3/3
host icmet05, rank 16/23, thread ID 1/3
host icmet06, rank 23/23, thread ID 2/3
host icmet06, rank 23/23, thread ID 3/3
host icmet06, rank 23/23, thread ID 1/3
host icmet06, rank 23/23, thread ID 0/3
host icmet02, rank 6/23, thread ID 0/3
host icmet02, rank 6/23, thread ID 3/3
host icmet02, rank 6/23, thread ID 1/3
host icmet02, rank 6/23, thread ID 2/3
host icmet02, rank 4/23, thread ID 1/3
host icmet02, rank 4/23, thread ID 3/3
host icmet02, rank 4/23, thread ID 2/3
host icmet02, rank 4/23, thread ID 0/3
host icmet03, rank 11/23, thread ID 2/3
host icmet03, rank 11/23, thread ID 1/3
host icmet03, rank 11/23, thread ID 3/3
host icmet03, rank 11/23, thread ID 0/3
host icmet04, rank 15/23, thread ID 0/3
host icmet04, rank 15/23, thread ID 2/3
host icmet04, rank 15/23, thread ID 3/3
host icmet04, rank 15/23, thread ID 1/3
host icmet01, rank 3/23, thread ID 0/3
host icmet01, rank 3/23, thread ID 2/3
host icmet01, rank 3/23, thread ID 1/3
host icmet01, rank 3/23, thread ID 3/3
host icmet01, rank 0/23, thread ID 0/3
host icmet01, rank 0/23, thread ID 2/3
host icmet01, rank 0/23, thread ID 3/3
host icmet01, rank 0/23, thread ID 1/3
host icmet03, rank 8/23, thread ID 0/3
host icmet03, rank 8/23, thread ID 3/3
host icmet03, rank 8/23, thread ID 2/3
host icmet03, rank 8/23, thread ID 1/3
host icmet04, rank 12/23, thread ID 2/3
host icmet04, rank 12/23, thread ID 1/3
host icmet04, rank 12/23, thread ID 3/3
host icmet04, rank 12/23, thread ID 0/3
host icmet06, rank 20/23, thread ID 1/3
host icmet06, rank 20/23, thread ID 3/3
host icmet06, rank 20/23, thread ID 0/3
host icmet06, rank 20/23, thread ID 2/3
host icmet05, rank 19/23, thread ID 1/3
host icmet05, rank 19/23, thread ID 3/3
host icmet05, rank 19/23, thread ID 2/3
host icmet05, rank 19/23, thread ID 0/3
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

Collective communications

Collective communication operations

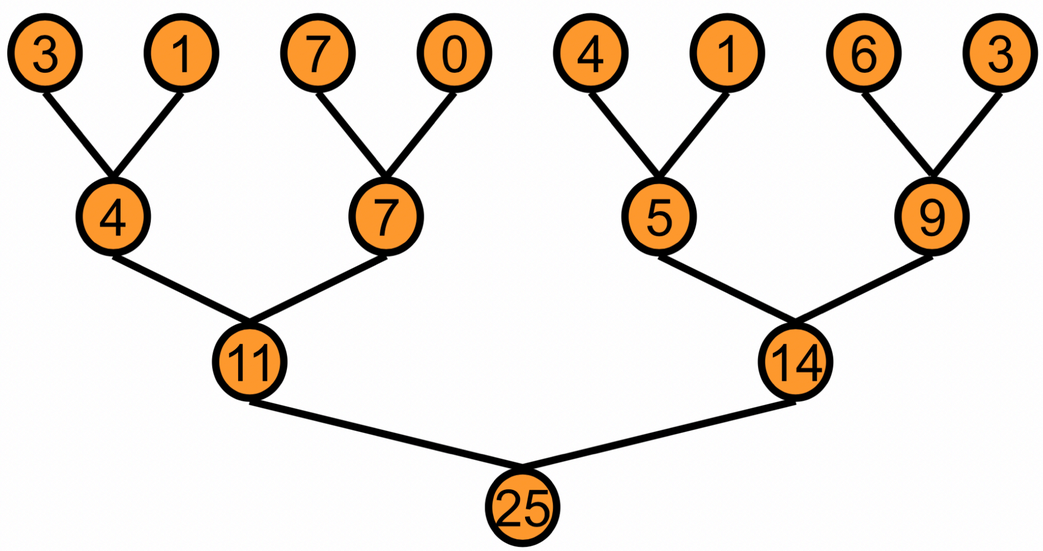
- So far, we have seen how two processes can communicate with each other and exchange data.
- However, there are cases where we want multiple processes to exchange data.
- A communication operation is called **collective or global** if all or a subset of the processes of a parallel program are involved.

Example: reduction

- Let’s say that we have a group of processes that 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 main node is inefficient.

Example of tree-based reduction

- A tree-based reduction allows using the interconnection network more effectively.
- If all processes send their data to process 0, there is potentially a bottleneck at process 0.
- By using a binary tree, we can make more efficient use of the topology of the network.



Computer network = network of highways

- Each highway has several lanes and a maximum traffic it can support.
- This is the bandwidth.
- We can improve performance by routing the messages optimally through the network and avoiding congestion and bottlenecks.

Examples of networks

a) complete graph

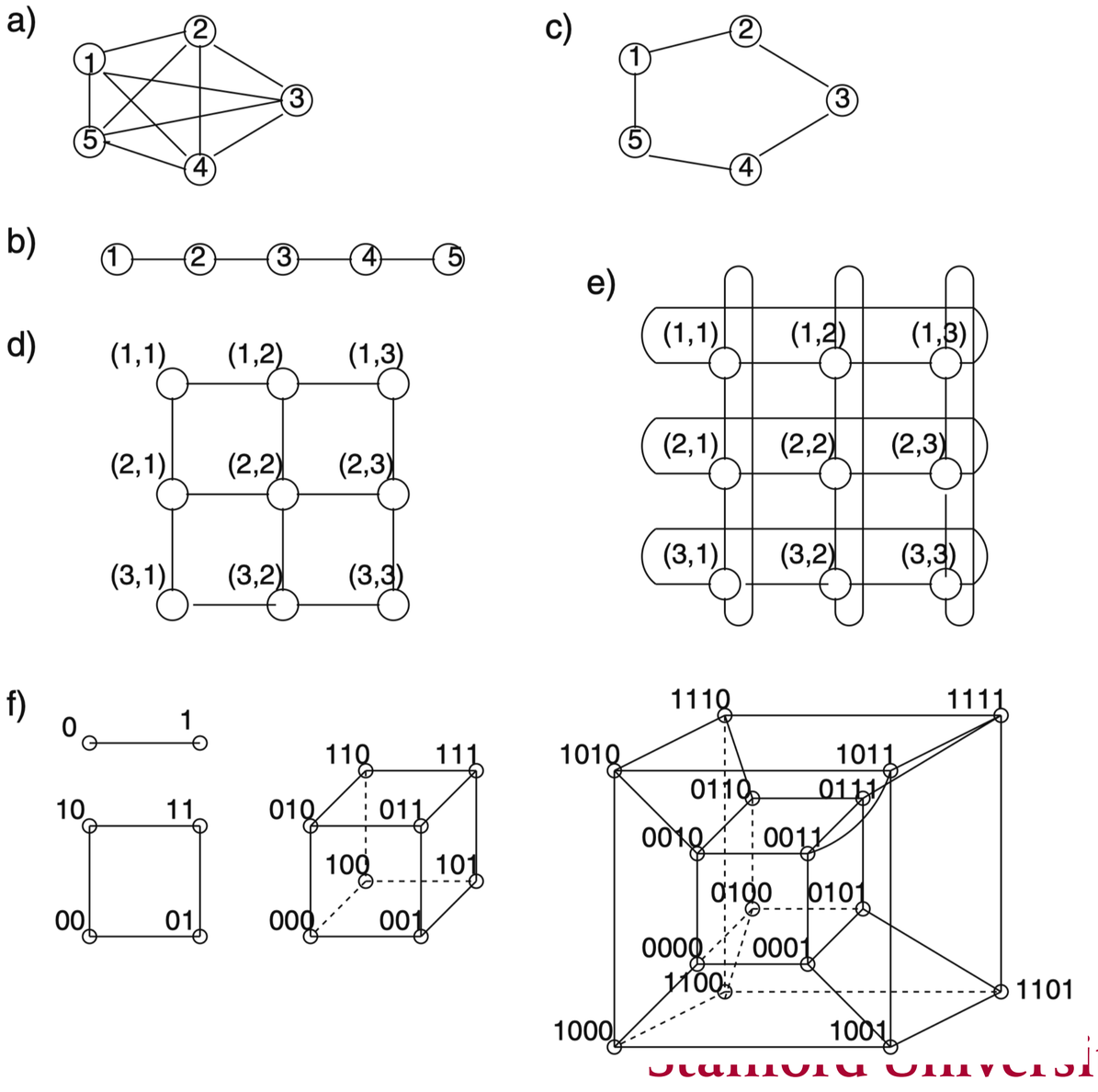
b) linear array

c) ring

d) 2D mesh

e) 2D torus

f) k-dimensional cube



MPI collective library

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

- Three key issues:

1. These communication algorithms can be complicated.

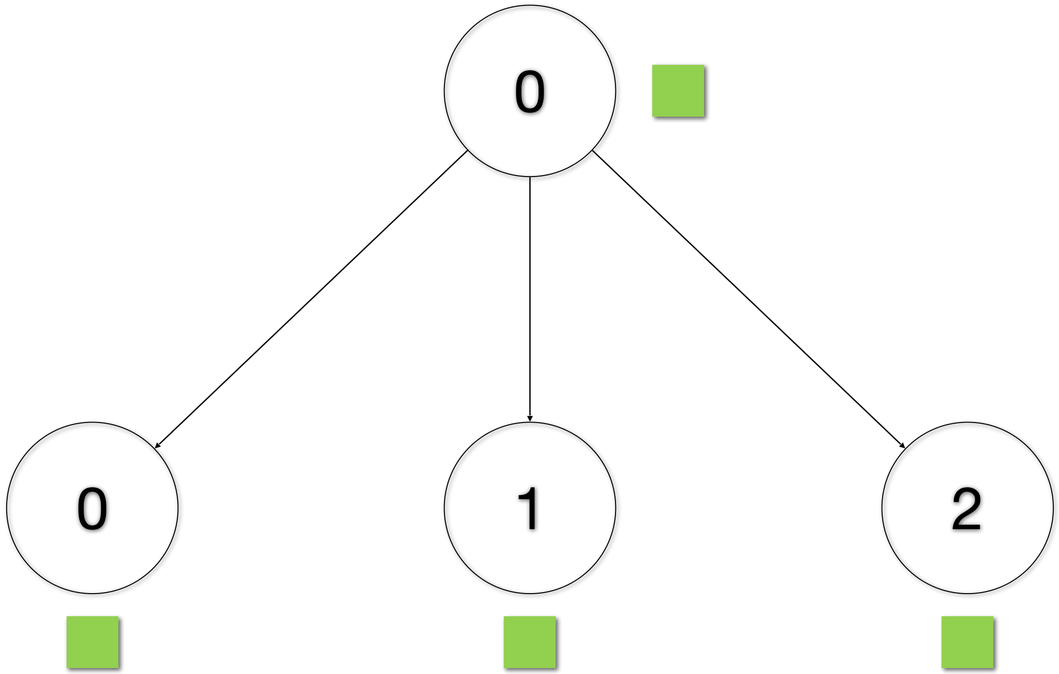
2. They depend on the network topology.

3. There are relatively few collective communication patterns that get reused repeatedly.

Let's review the main collective functions in MPI

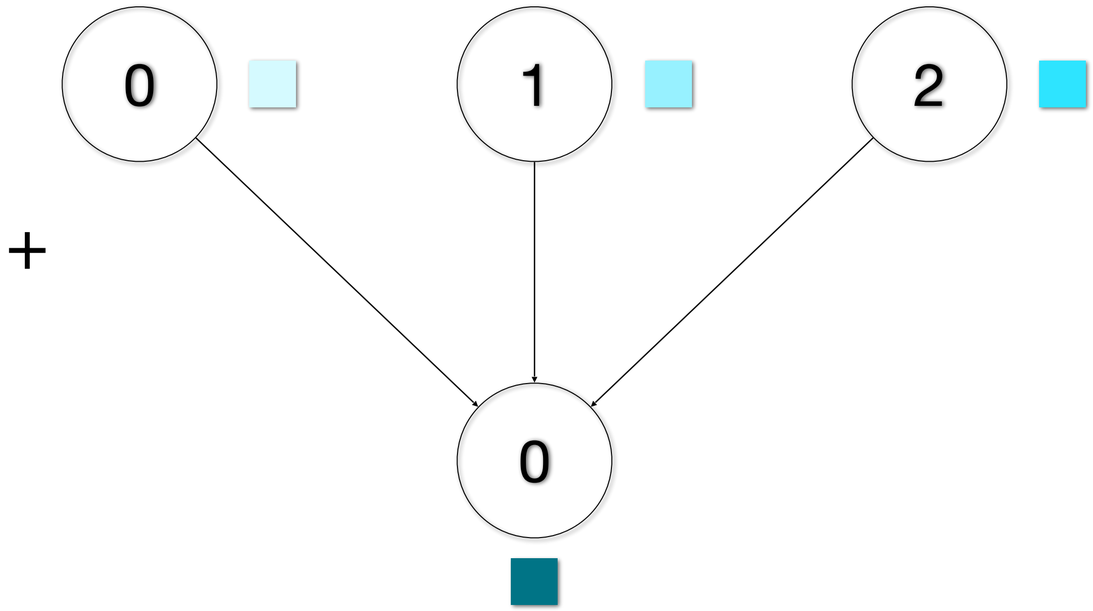
`MPI_Bcast(&buffer,count,datatype,root,comm)`

One process sends data to all the other processes



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

One process receives the result of reducing data from all the processes.



All MPI reduction operations

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

Example: `mpi_prime.cpp`

```
// Total number of primes found by all processes:     MPI_SUM
MPI_Reduce(&pc, &pcsum, 1, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);

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

We perform two reductions.

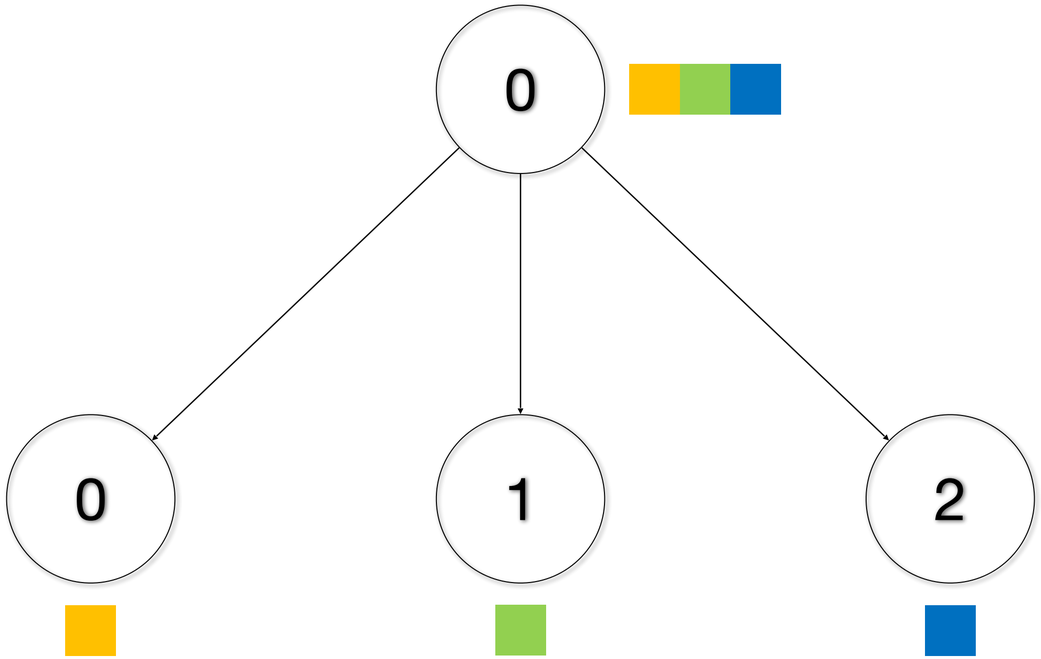
- `MPI_SUM` calculates the total number of primes that were found.
- `MPI_MAX` calculates the largest prime number that was found.

Output

```
darve@icme-gpu1:~/2023/MPI/Lecture_16$ srun -n 4 -p CME ./mpi_prime
MPI task 1 has started on icmet01 [total number of processors 4]
MPI task 2 has started on icmet01 [total number of processors 4]
MPI task 3 has started on icmet01 [total number of processors 4]
MPI task 0 has started on icmet01 [total number of processors 4]
Using 4 tasks to scan 40000000 numbers...
Done.
Largest prime is 39999983.
Total number of primes found: 2433654
Wall clock time elapsed: 12.75 seconds
darve@icme-gpu1:~/2023/MPI/Lecture_16$
```

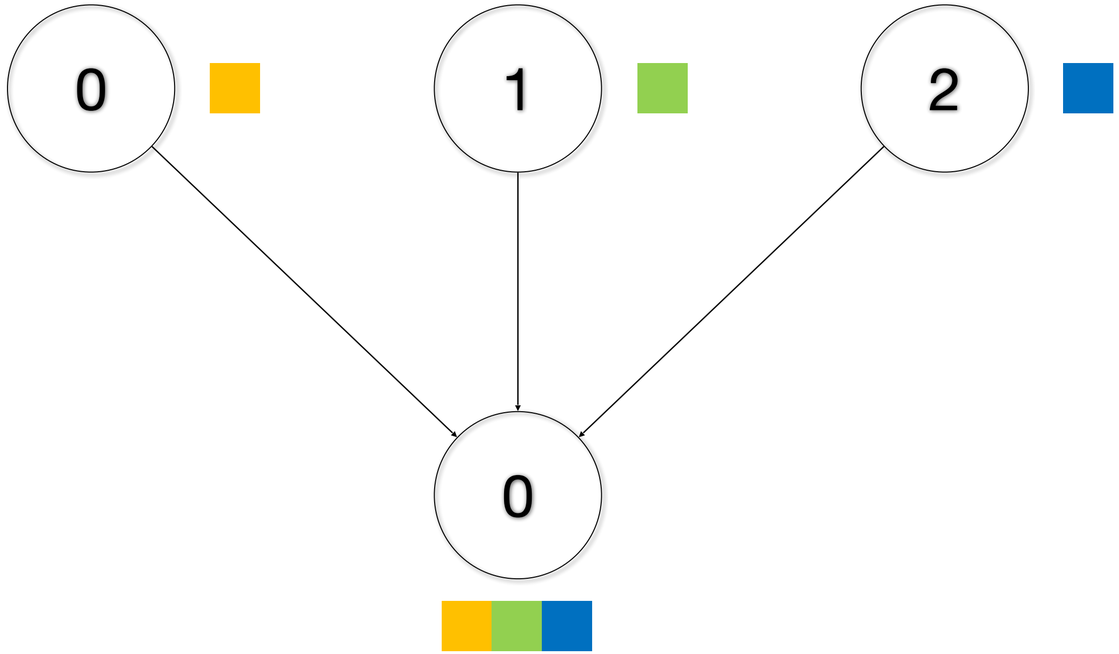
```
MPI_Scatter(&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)
```

One process sends different data to the other processes



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

One process gathers data from the other processes



Final project

- Rank 0 reads MNIST data from disk.
- `MPI_Scatter` can send the images to all the other processes.

Number of GPUs = 3

- Another MPI function, `MPI_Scatterv`, is required when the number of processes does not divide the number of images, e.g., the number of GPUs is 3.

- In that case, process 0 needs to send a different number of images to the other processes.

Variant: `MPI_Scatterv`

Scatters a buffer in parts to all tasks in a group.

```
int MPI_Scatterv(&sendbuf, &sendcounts, &displs,
sendtype, &recvbuf, recvcount, recvtype, root, comm)
```

`MPI_Scatterv` extends the functionality of `MPI_Scatter` by allowing a **varying count of data** to be sent to each process since `sendcounts` is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing the new argument, `displs`.

Example

This example uses a stride.

The root process scatters sets of 100 ints to the other processes, but the sets of 100 are **stride** ints apart in the sending buffer.

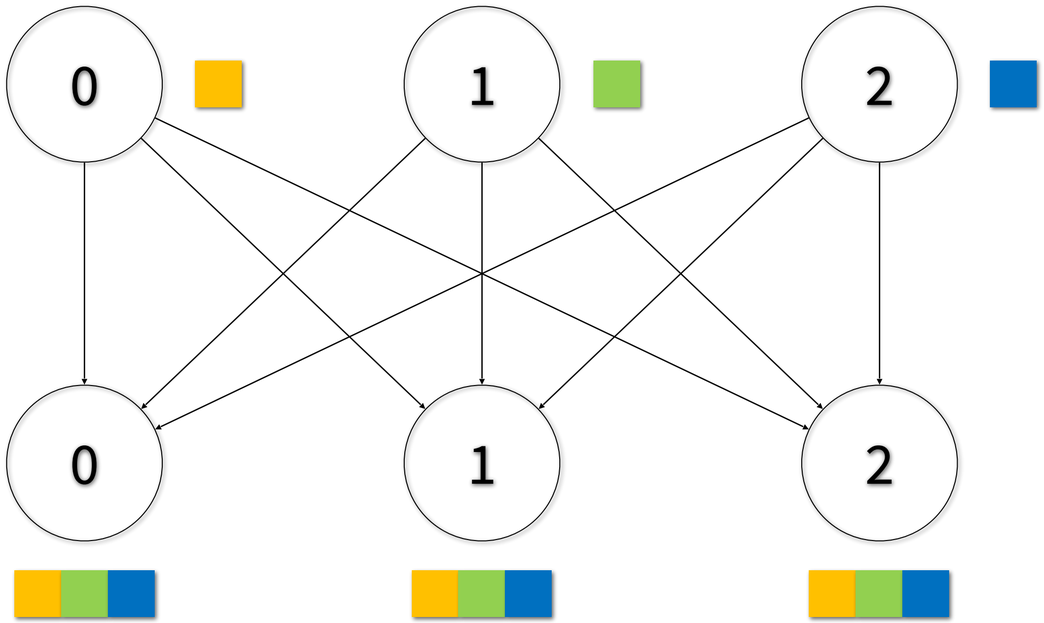
See more details and examples in the documentation.

https://www.open-mpi.org/doc/v4.1/man3/MPI_Scatterv.3.php

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
  displs[i] = i*stride;
  scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT,
             rbuf, 100, MPI_INT, root, comm);
```

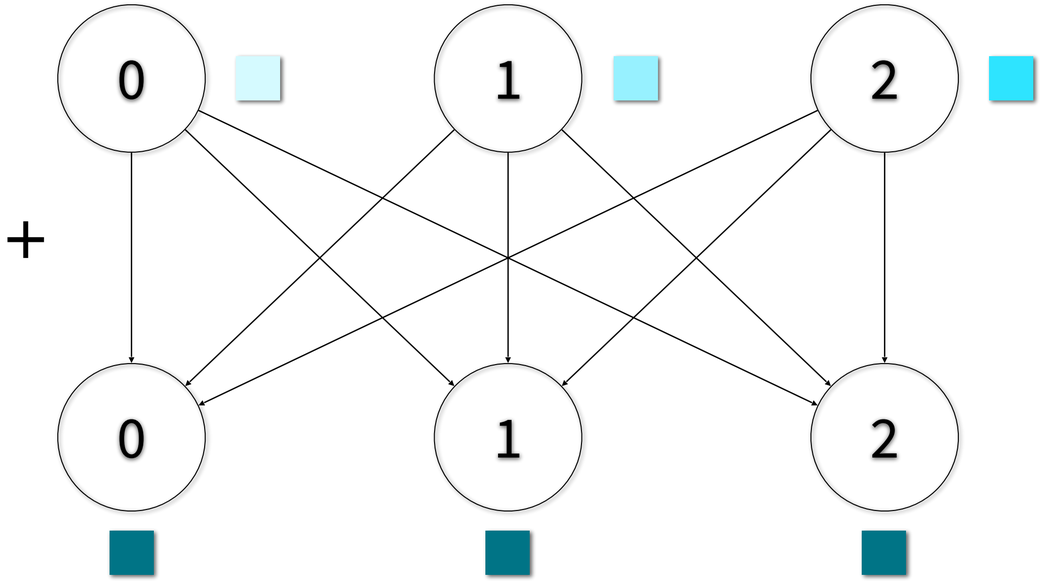
```
MPI_Allgather(&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm)
```

All processes gather data from all the other processes.



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

Same as reduction but result is stored at all processes.



Final project

There are a few reductions that you will have to perform using MPI.

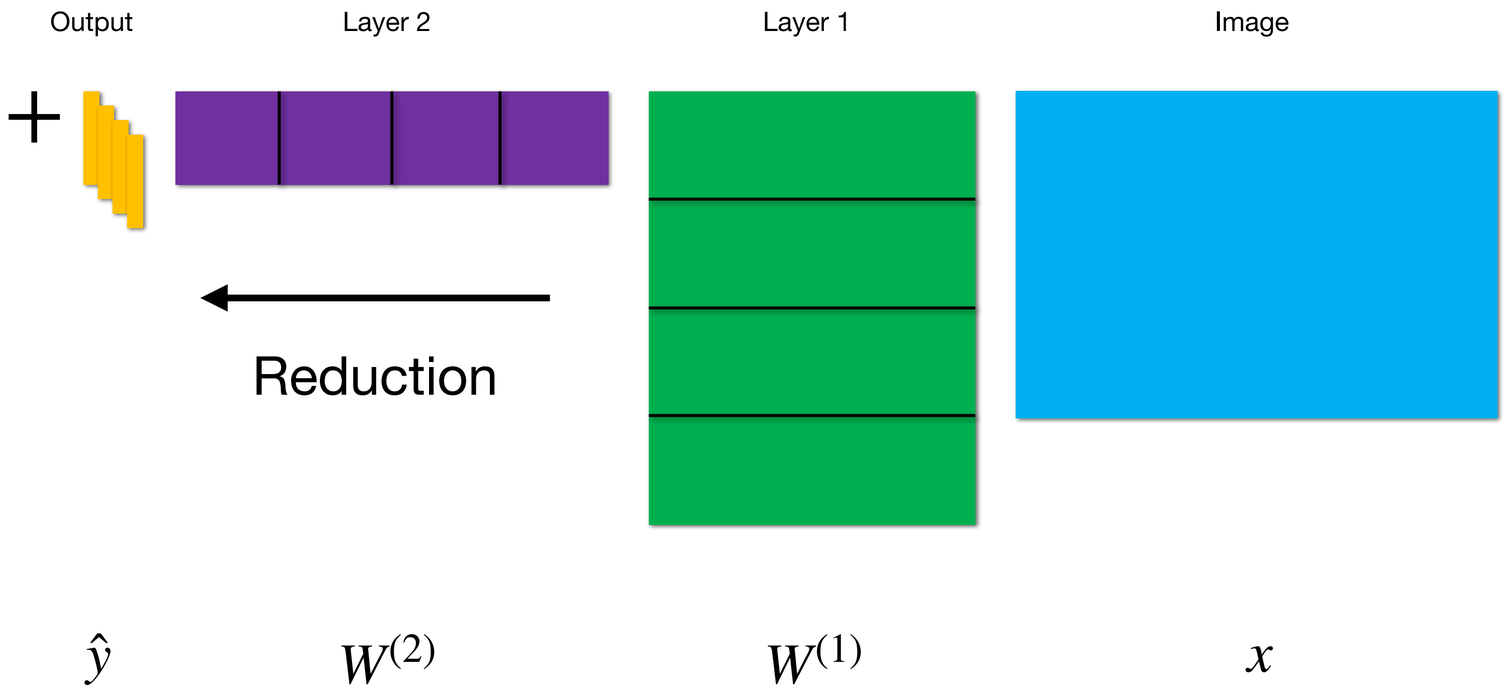
Implementation 1: reduction over the input images.

$$p \leftarrow p - \frac{\alpha}{N} \nabla_p \sum_{i=1}^{N} H(y_i, \hat{y}_i(p))$$

Reduction is required over the index i.

`MPI_Allreduce` to get the complete gradient on all processes.

Final project



Implementation 2: reduction for layer 2 in order to obtain the final output.