# CME 213

## SPRING 2019

Eric Darve

# MPI

# Parallel programming using Message Passing

# Why MPI?

- Shared memory is a good model for a small number of processes.
- The main difficulty is ensuring a consistent view of memory as threads read and write to memory.
- This can scale to many cores but beyond ~ 100s, it does not work anymore.
- Instead, we need to switch to computer nodes connected through a network.
- In that case, the memory becomes distributed.
- Processes need to communicate explicitly. This can be done using MPI.
- MPI is the standard for distributed memory computing.

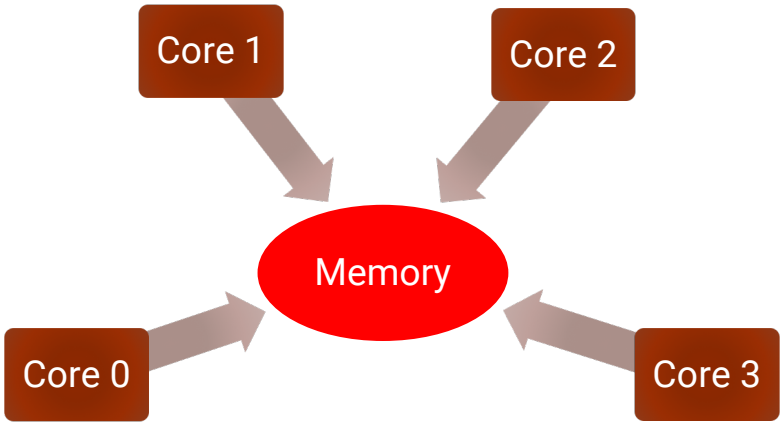
# Flynn's taxonomy

We have seen the following different types of parallelism:

- SIMD: single instruction multiple data
  - All processing units execute the same instruction at any given clock cycle.
  - Each processing unit can operate on a different data element.
  - This applies for example to a GPU streaming multiprocessor.
  - The thread index is used to determine which data the thread operates with.

- MIMD/SPMD: multiple instruction, multiple data
  - Every processor executes a different instruction stream.
  - Every processor works with a different data stream.
  - This applies to Pthreads for example. Threads can execute different routines.
- MPI follows this model.

# Parallel computer memory architecture

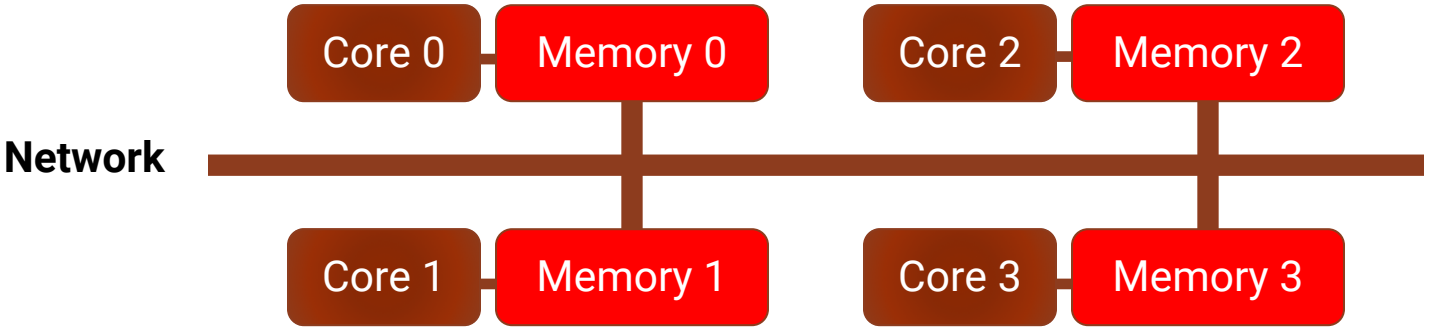



Shared memory: all cores can read and write to the same memory space
CUDA, Pthreads, OpenMP

Distributed memory:

- Cores read and write to different memory spaces
- If a core needs data in some other memory space, explicit communication is required

MPI



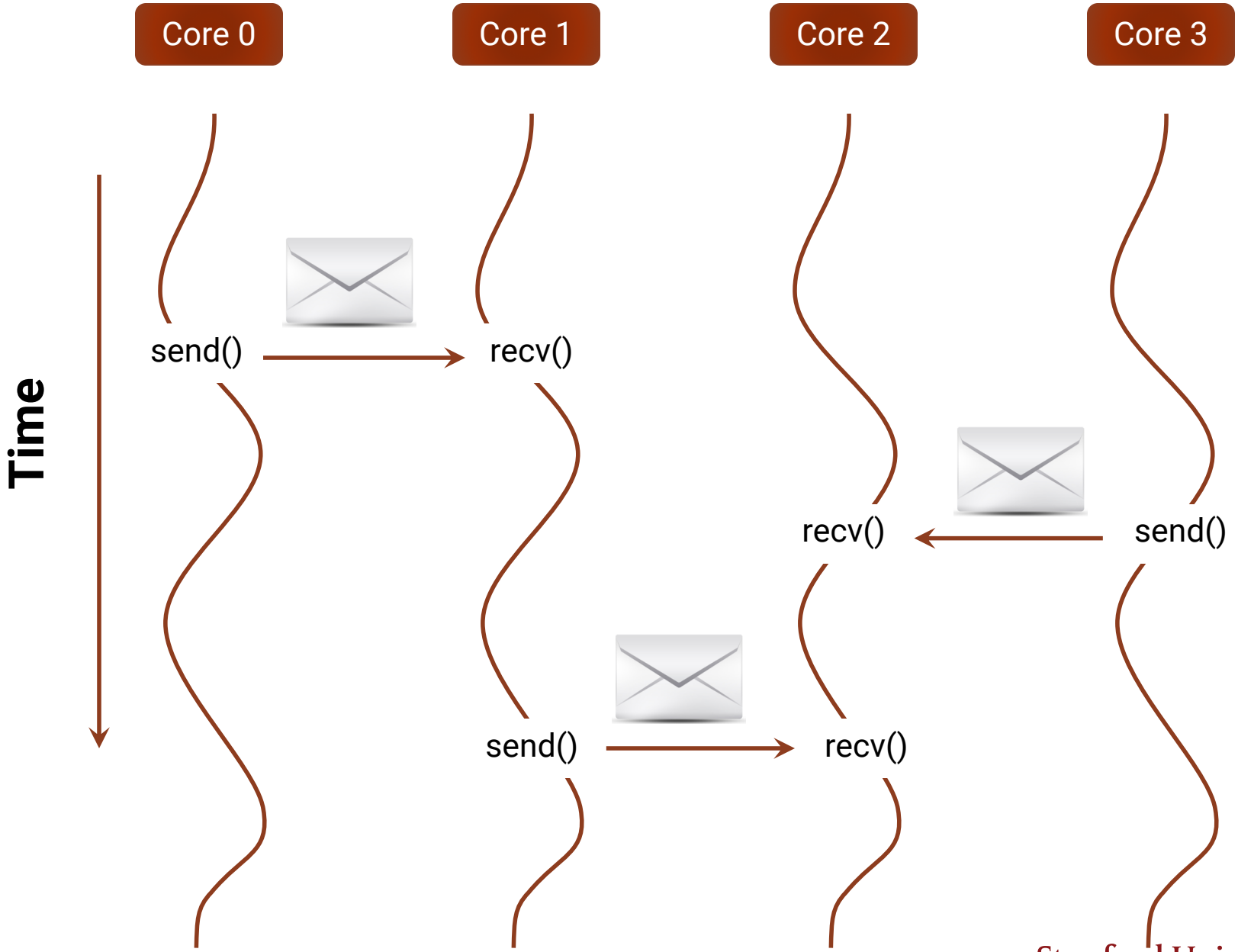
# MPI: message passing interface

- Simple scenario: all processes run the same program. This is an SPMD system.
- Processes are assigned a rank.
- Based on the rank, processes perform calculations on different data.

Processes communicate by sending and receiving messages.

Message passing:

- Data transfer requires cooperative operations to be performed by each process.
- For example, a send operation must have a
- matching receive operation.


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

# Our first MPI program

# MPI implementations

- **OpenMPI:** www.open-mpi.org (what we use for the GCP VM)
- MVAPICH: mvapich.cse.ohio-state.edu
- MPICH: www.mpich.org

- You can test MPI using a multicore computer.
- Each process runs on its own core.
- You can run this on GCP or even your laptop

# MPI on GCP

- A reminder that GCP gives you access to virtual machines.
- Each script we give you corresponds to a **machine type**, which specifies virtualized hardware resources, including the memory size, virtual CPU (vCPU) count, and maximum persistent disk capability.
- A vCPU is implemented as a single hardware hyper-thread on one of the available CPU Platforms.
- This means that when you request cores, you don't really know what cores you get and where they are.
- They could be physically on the same processor/board (the most likely) but they may be on different motherboards.
- However, when querying the computer, it will look like you have a single processor with many cores and shared memory.

# Compiling an MPI code

- Compile with:

```
mpic++
```

- Header:

```
mpi.h
```

# mpirun

- Running your code requires starting multiple processes at once, and terminating all of them when the calculation completes.
- This is facilitated by **mpirun**

```
darve@mpi8:~/MPI$ mpirun -n 8 ./mpi_hello
--------------------------------------------------------------------------
[[42778,1],1]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: mpi8

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------
Hello from task  0 running on node: mpi8
MASTER process: the number of MPI tasks is:  8
Hello from task  1 running on node: mpi8
Hello from task  2 running on node: mpi8
Hello from task  3 running on node: mpi8
Hello from task  4 running on node: mpi8
Hello from task  5 running on node: mpi8
Hello from task  6 running on node: mpi8
Hello from task  7 running on node: mpi8
[mpi8:01978] 7 more processes have sent help message help-mpi-btl-base.txt / btl:no-nics
[mpi8:01978] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
darve@mpi8:~/MPI$
```

# Transfer layer

- mpirun documentation
- Many transport layers (software/hardware used to exchange data) can be used
- Examples: Elan, InfiniBand, Myrinet, Portals, shared memory, TCP sockets, etc
- **openib** is the option to use the *OpenFabrics* network (not available on GCP).
- Use this command to remove this error message:

```
mpirun -mca btl ^openib -n 8 ./mpi_hello
```

```
darve@mpi8:~/MPI$ mpirun -mca btl ^openib -n 8 ./mpi_hello
Hello from task  0 running on node: mpi8
MASTER process: the number of MPI tasks is:  8
Hello from task  3 running on node: mpi8
Hello from task  5 running on node: mpi8
Hello from task  1 running on node: mpi8
Hello from task  2 running on node: mpi8
Hello from task  6 running on node: mpi8
Hello from task  7 running on node: mpi8
Hello from task  4 running on node: mpi8
darve@mpi8:~/MPI$
```

# BTL

- **`mca`** Modular Component Architecture (controls many options in openmpi)
- **`btl`** Byte transfer layer (point-to-point byte movement)
- **`^openib`** do not use openib
- **`$ ompi_info`** for more information

Examples of BTL options:

- openib Open Fabrics
- self send to self semantics
- sm shared memory
- smcuda shared memory with CUDA support
- tcp Transmission Control Protocol
- ugni Cray Gemini/Aries
- vader shared memory using XPMEM/Cross-mapping

```c
// Some MPI magic to get started
MPI_Init(&argc, &argv);

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

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

// Only one processor will do this
if (taskid == MASTER)
{
  printf("MASTER process: the number of MPI tasks is: %2d\n", numtasks);
}

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

ersity

# Example

- Example code:
  **`mpi_hello.c mpi_pi_send.c`**
- **`mpi_pi_send:`** calculates pi by throwing darts, i.e., one draws two uniform random numbers (x,y) in the (0,1) interval and calculates how many, on average, fall inside the 2D unit circle. From this average, pi can be estimated.
- We will see later on how to do this better using collective communications.

# Send

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

**smessage** buffer which contains the data elements to be sent

**count** number of elements to be sent

**datatype** data type of entries

**dest** rank of the target process

**tag** message tag which can be used by the receiver to distinguish between different messages from the same sender

**comm** communicator used for the communication (more on this later)

```
darve@mpi8:~/MPI$ mpirun -mca btl ^openib -n 8 ./mpi_pi_send
MPI task  0 has started on mpi8 [total number of processors 8]
MPI task  1 has started on mpi8 [total number of processors 8]
MPI task  3 has started on mpi8 [total number of processors 8]
MPI task  5 has started on mpi8 [total number of processors 8]
MPI task  2 has started on mpi8 [total number of processors 8]
MPI task  7 has started on mpi8 [total number of processors 8]
MPI task  4 has started on mpi8 [total number of processors 8]
MPI task  6 has started on mpi8 [total number of processors 8]
   After  4000000 throws, average value of pi = 3.14189700
   After  8000000 throws, average value of pi = 3.14156450
   After 12000000 throws, average value of pi = 3.14163733
   After 16000000 throws, average value of pi = 3.14153375
   After 20000000 throws, average value of pi = 3.14150340
   After 24000000 throws, average value of pi = 3.14158200
   After 28000000 throws, average value of pi = 3.14155329
   After 32000000 throws, average value of pi = 3.14160950
   After 36000000 throws, average value of pi = 3.14149211
   After 40000000 throws, average value of pi = 3.14149820

Exact value of pi: 3.1415926535897
darve@mpi8:~/MPI$
```

```
for(int i = 0; i < ROUNDS; i++) {
  double my_pi = DartBoard(DARTS);
  if(taskid != MASTER) {
    int tag = i;
    int rc = MPI_Send(&my_pi, 1, MPI_DOUBLE,
                      MASTER, tag, MPI_COMM_WORLD);
  } else {
    int tag = i;
    double pisum = 0;
    for(int n = 1; n < numtasks; n++) {
      double pirecv;
      MPI_Status status;
      int rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                        tag, MPI_COMM_WORLD, &status);
      pisum += pirecv;
    }
    double pi = (pisum + my_pi)/numtasks;
    printf("   pi for this round = %10.8f\n", pi);
    avepi = ((avepi * i) + pi)/(i + 1);
    printf("   After %8d throws, average value of pi = %10.8f\n",
           (DARTS * (i + 1) * numtasks), avepi);
  }
}
```

iversity

# mpi_pi_send

- 8 identical programs are running.
- Depending on **`taskid`**, cores do different things.
- A send/recv pair is used to exchange data.
- Messages can arrive in any order.
- The use of a tag ensures that master will wait for all messages corresponding to a given round before printing the approximation.

# Recv

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

- Same as before.
- New argument: **`status`** data structure that contains information about the message that was received

# MPI data types

| MPI data type | C data type |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_LONG_LONG_INT | long long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_WCHAR | wide char |
| MPI_PACKED | special data type for packing |
| MPI_BYTE | single byte value |

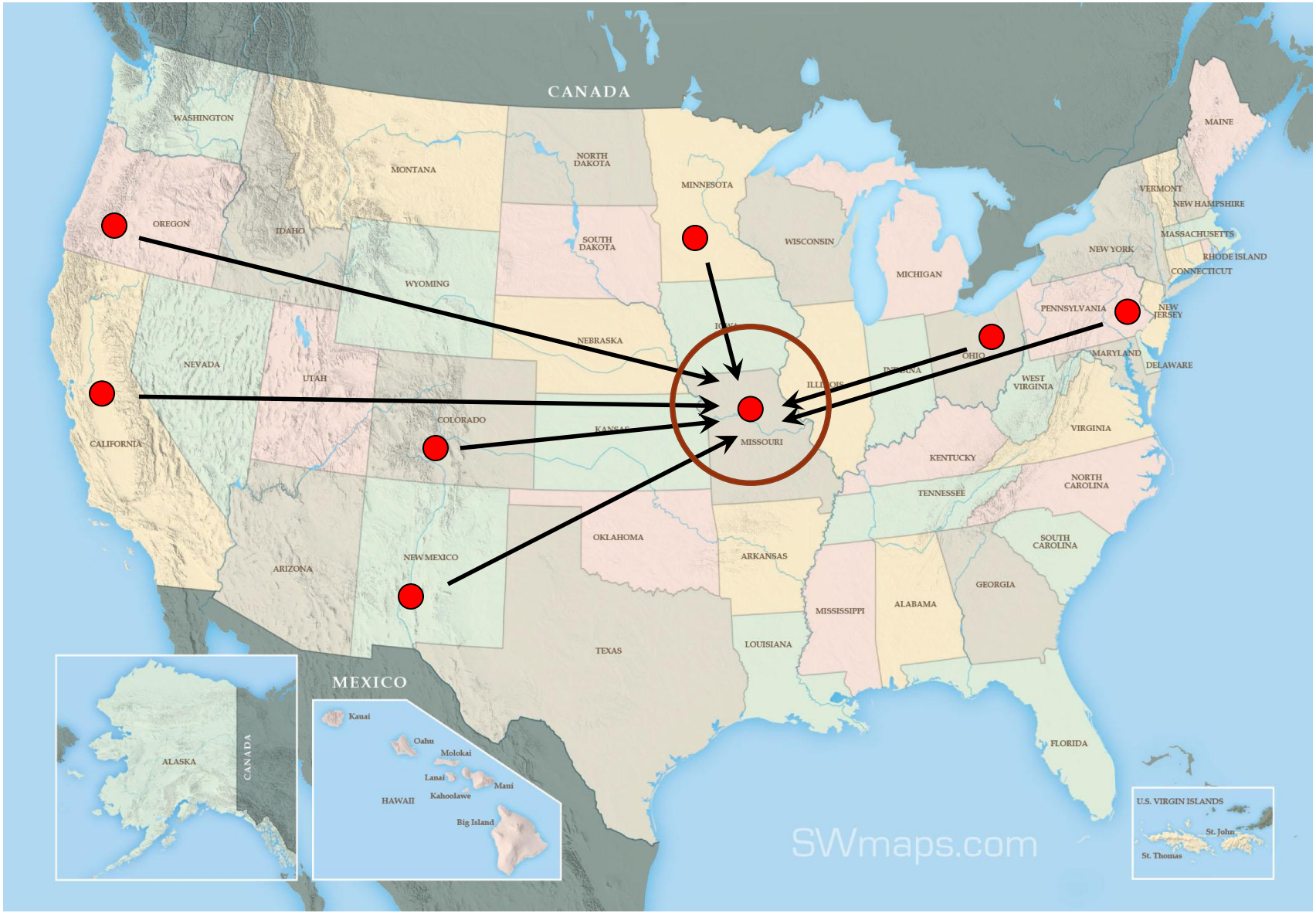
# How does it work?

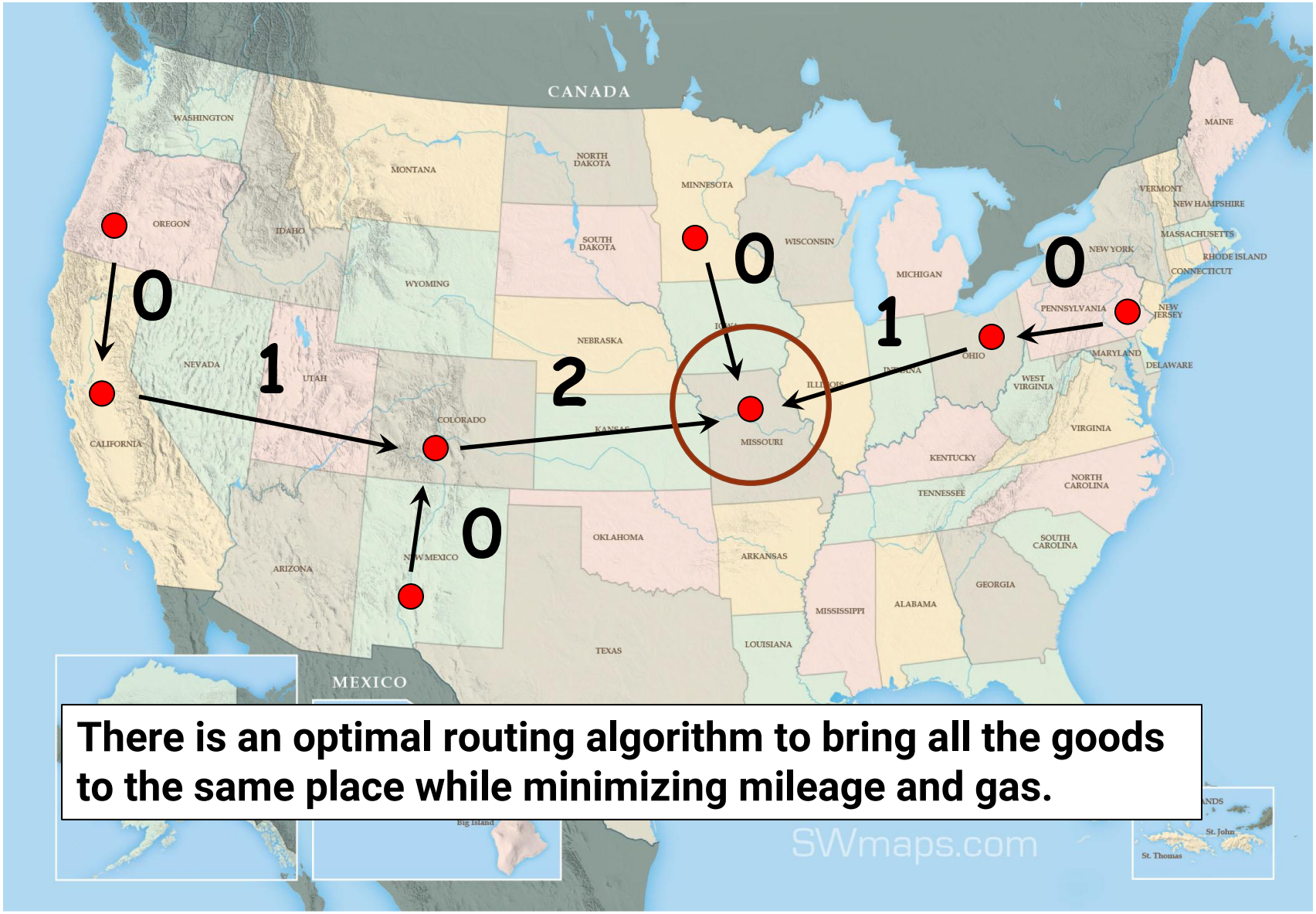
- Each `Send` must be matched with a corresponding `Recv`.
- Order: messages are delivered in the order in which they have been sent.
- If a sender sends two messages of the same type one after another to the same receiver, the MPI runtime system ensures that the first message sent is always received first.

# COLLECTIVE COMMUNICATIONS

# Point-to-point vs collective

- What we have discussed so far is point-to-point communication, that is one process communicates with another process.
- All communications can be ultimately broken down into this type of exchanges.
- Nevertheless, 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.
- In that case, for best performance, we need to orchestrate the communication. Simply having each process send its data to the master node is inefficient.


CANADA
WASHINGTON
OREGON
CALIFORNIA
NEVADA
IDAHO
MONTANA
WYOMING
UTAH
ARIZONA
COLORADO
NEW MEXICO
NORTH DAKOTA
SOUTH DAKOTA
NEBRASKA
KANSAS
OKLAHOMA
TEXAS
MINNESOTA
IOWA
MISSOURI
ARKANSAS
LOUISIANA
WISCONSIN
ILLINOIS
MICHIGAN
INDIANA
OHIO
KENTUCKY
TENNESSEE
MISSISSIPPI
ALABAMA
GEORGIA
FLORIDA
SOUTH CAROLINA
NORTH CAROLINA
VIRGINIA
WEST VIRGINIA
PENNSYLVANIA
NEW YORK
MARYLAND
DELAWARE
NEW JERSEY
CONNECTICUT
RHODE ISLAND
MASSACHUSETTS
VERMONT
NEW HAMPSHIRE
MAINE
MEXICO
ALASKA
CANADA
HAWAII
Kauai
Oahu
Molokai
Lanai
Maui
Kahoolawe
Big Island
U.S. VIRGIN ISLANDS
St. John
St. Thomas
SWmaps.com


CANADA
WASHINGTON
OREGON
CALIFORNIA
IDAHO
NEVADA
MONTANA
WYOMING
UTAH
ARIZONA
COLORADO
NEW MEXICO
NORTH DAKOTA
SOUTH DAKOTA
NEBRASKA
KANSAS
OKLAHOMA
TEXAS
MINNESOTA
IOWA
MISSOURI
ARKANSAS
LOUISIANA
WISCONSIN
ILLINOIS
MICHIGAN
INDIANA
OHIO
KENTUCKY
TENNESSEE
MISSISSIPPI
ALABAMA
GEORGIA
SOUTH CAROLINA
NORTH CAROLINA
VIRGINIA
WEST VIRGINIA
PENNSYLVANIA
NEW YORK
MARYLAND
DELAWARE
NEW JERSEY
CONNECTICUT
RHODE ISLAND
MASSACHUSETTS
VERMONT
NEW HAMPSHIRE
MAINE
MEXICO
0
1
2
0
0
1
0
There is an optimal routing algorithm to bring all the goods to the same place while minimizing mileage and gas.
SWmaps.com
Big Island
St. John
St. Thomas

# Network and highways

- Think of a computer network as a network of highways.
- Each highway has a number of lanes and a maximum traffic it can support. This is the bandwidth.
- If all the cars in the bay area decide to take US 101 at the same time, you get a huge traffic jam.
- Computer network is the same. You cannot have too many messages traveling across the same wire of the network. There is a maximum bandwidth that each wire in the network can support.
- 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.

There are three key issues:

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

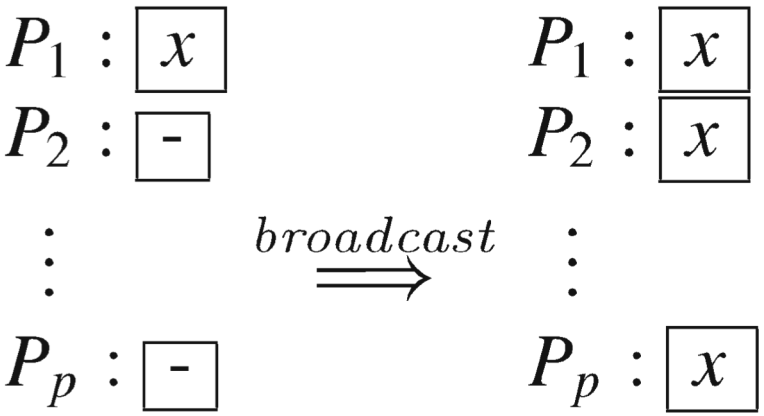
# Collective communications

- MPI provides a library of collective communications.
- It covers 99% of the use cases.
- For the rest, we still have **point-to-point.**
- All these operations are blocking.

# Single broadcast

- The simplest communication: one process sends a piece of data to all other processes.

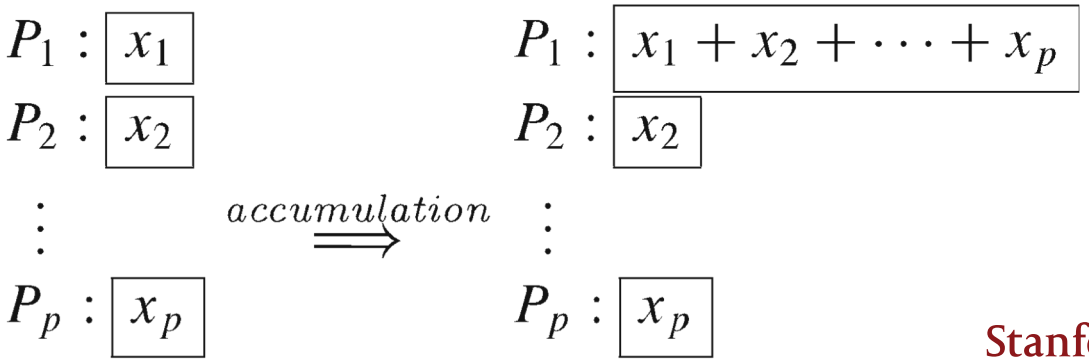
```
int MPI_Bcast(void *message, int count,
              MPI_Datatype type, int root,
              MPI_Comm comm)
```



# Single accumulation

- Each process provides a block of data with the same type and size.
- When performing the operation, a reduction operation is applied element by element to the data blocks provided by the processes
- The resulting accumulated data block is collected at a specific root process.

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

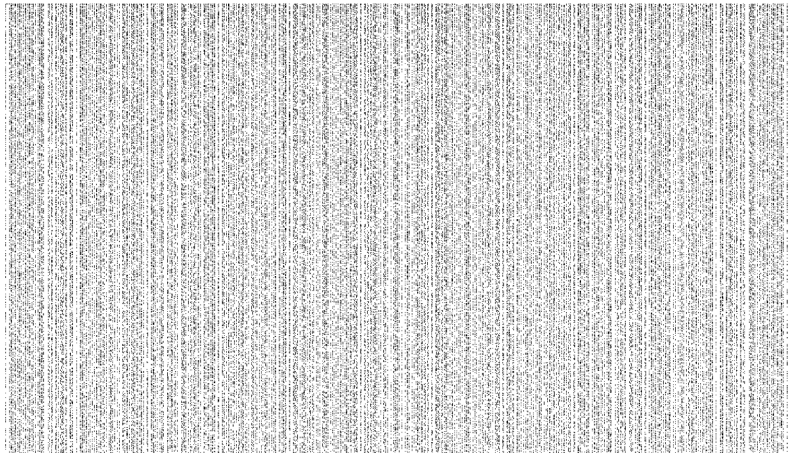


| Representation | Operation |
|---|---|
| MPI_MAX | Maximum |
| MPI_MIN | Minimum |
| MPI_SUM | Sum |
| MPI_PROD | Product |
| MPI_LAND | Logical and |
| MPI_BAND | Bit-wise and |
| MPI_LOR | Logical or |
| MPI_BOR | Bit-wise or |
| MPI_LXOR | Logical exclusive or |
| MPI_BXOR | Bit-wise exclusive or |
| MPI_MAXLOC | Maximum value and corresponding index |
| MPI_MINLOC | Minimum value and corresponding index |

# Code example

**`mpi_prime.cpp`**

- Computing prime numbers in parallel.
- We want to compute the total number of primes smaller than N and the largest prime smaller than N.



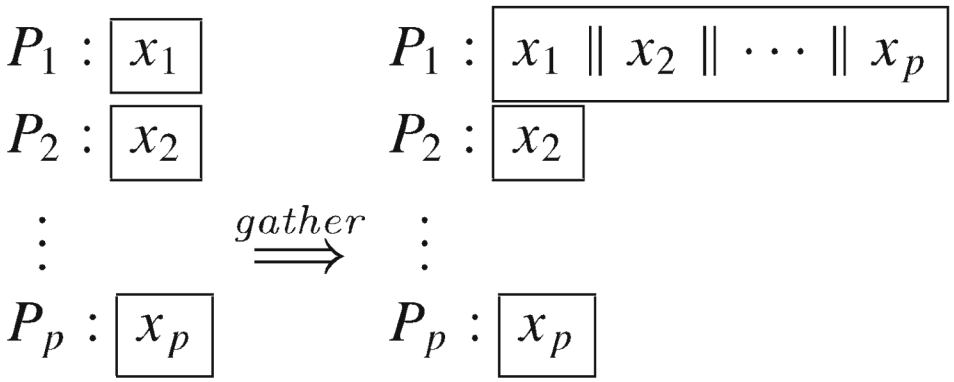
Distribution of prime numbers

```
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);
```

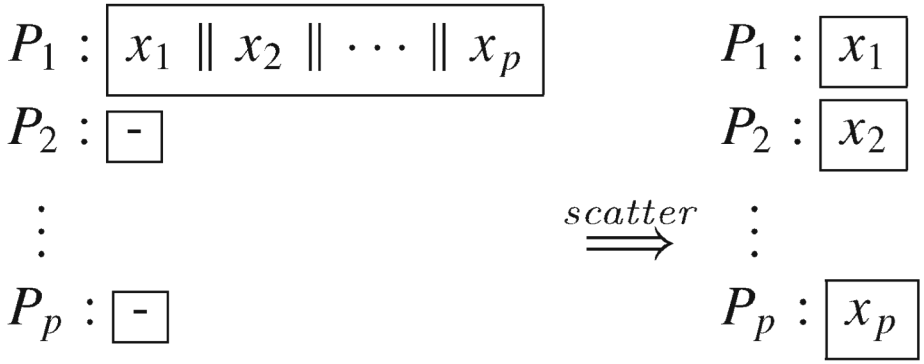
# Scalability

```
Using 1 tasks to scan 40000000 numbers...
Wall clock time elapsed: 19.29 seconds
Using 2 tasks to scan 40000000 numbers...
Wall clock time elapsed: 9.67 seconds
Using 3 tasks to scan 40000000 numbers...
Wall clock time elapsed: 9.63 seconds
Using 4 tasks to scan 40000000 numbers...
Wall clock time elapsed: 4.83 seconds
Using 5 tasks to scan 40000000 numbers...
Wall clock time elapsed: 5.46 seconds
Using 6 tasks to scan 40000000 numbers...
Wall clock time elapsed: 6.89 seconds
Using 7 tasks to scan 40000000 numbers...
Wall clock time elapsed: 4.73 seconds
Using 8 tasks to scan 40000000 numbers...
Wall clock time elapsed: 3.63 seconds
darve@mpi8:~/MPI/prime$
```

# Gather/Scatter

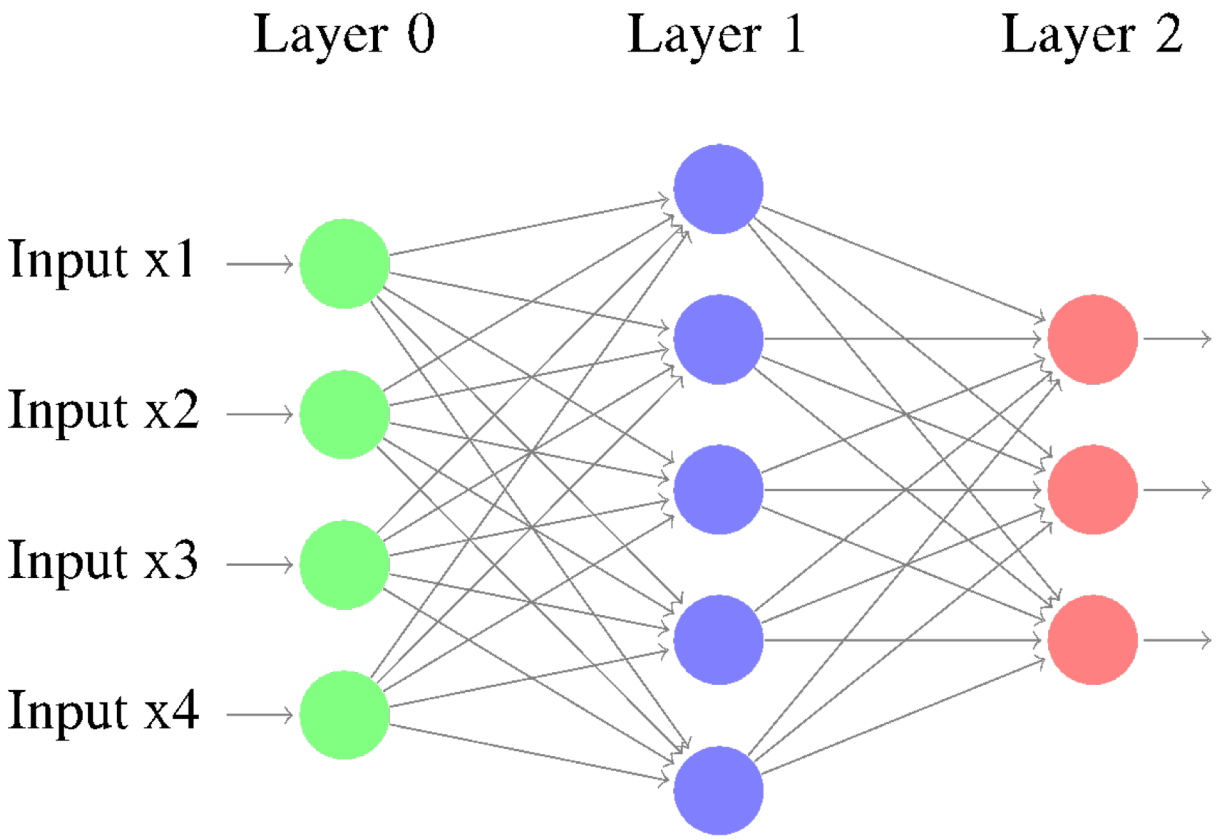



**MPI_Gather()**




**MPI_Scatter()**

# MPI and your final project



# Parallelizing over input images

- Split your set of input images.
- Assign each subset to an MPI process.

MPI process needs to:

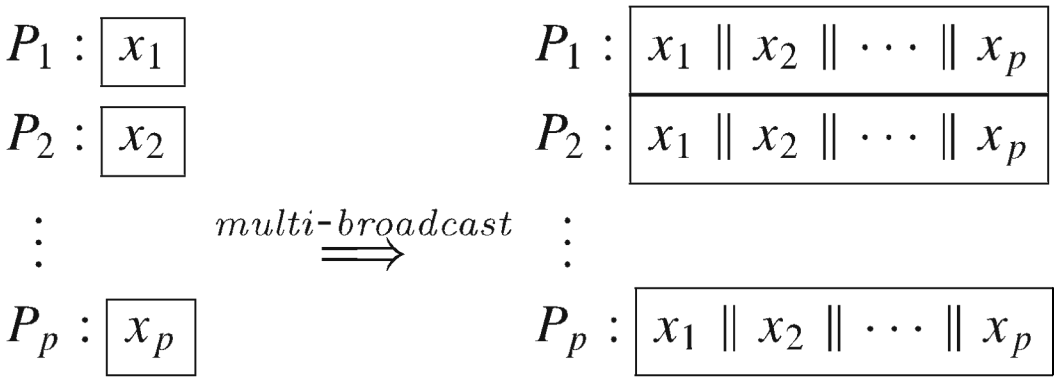
- Compute the feed-forward
- This means: a sequence of GEMM followed by sigmoid/softmax functions.
- Compute the back-propagation: derivative of error with respect to each NN coefficient. More GEMMs.

# MPI_Scatter

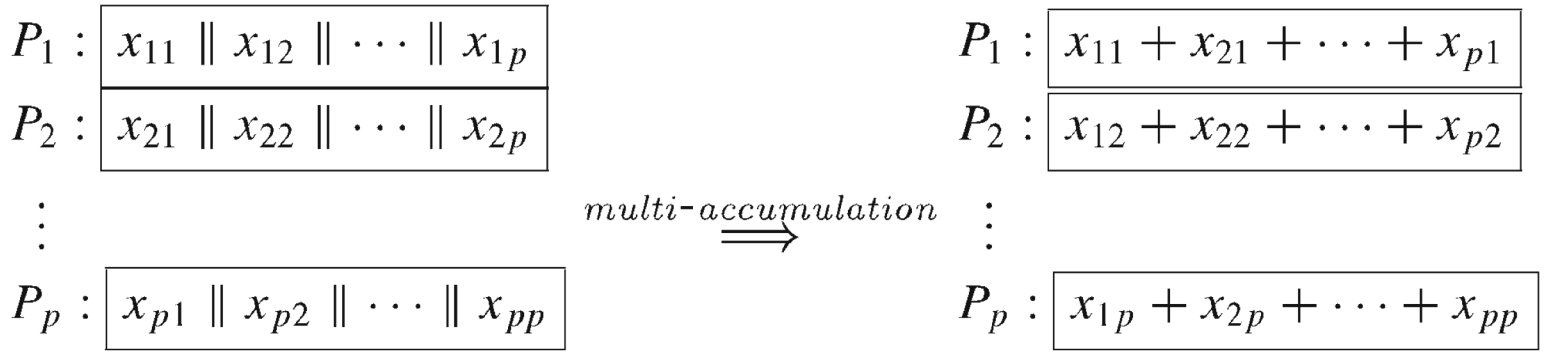
Two options:

- Each MPI process reads from file its own subset of images.
- Rank 0 process reads data from file.
  `MPI_Scatter()` used to send data from rank 0 to all other MPI ranks.

# Multi-broadcast/multi-accumulation




**`MPI_Allgather()`**




MPI provides a restricted version: **`MPI_Allreduce()`**, which is a reduction followed by a broadcast .

# Project: back-propagation and gradient of error

- The error is of the form:

$$J(W, b; x, y) = \frac{1}{N} \sum_{i=1}^{N} CE^{(i)}(y, \hat{y}) + 0.5 \, \lambda \, \|p\|^2$$

- A simple sum over all images!
- Hence, each $\Delta W$ is simply a sum over each image subset.
- MPI communication required: each process has a partial $\Delta W$.
- Add them up and broadcast the result.
- This is an `MPI_Allreduce()` operation.

# Code example

**`proc_min_value.cpp`**

**`mpiexec -n 8 ./proc_min_value`**

- Calculates the minimum value across all processes and the rank of the process that holds the minimum.
- This may be important if you need that process to send more data to the other processes (e.g., a broadcast).

Example that illustrates:

**`MPI_Allreduce`** reduction + broadcast.

**`MPI_MINLOC`** takes the minimum value + value attached to the minimum (in this example the rank of the process).

**`MPI_Barrier`** processes wait until all processes have reached that point. (This is only moderately useful in practice.)

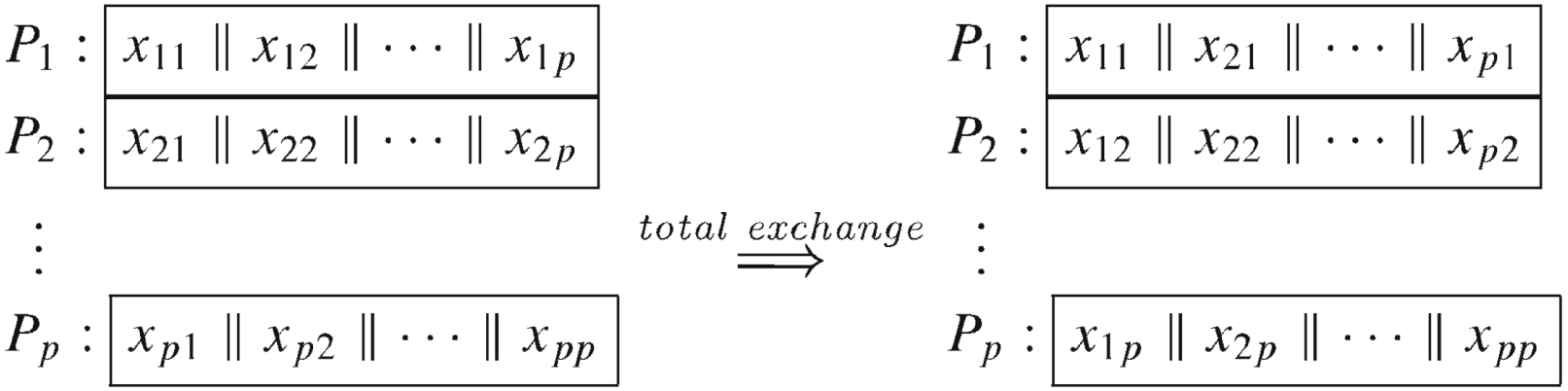
```
darve@mpi8:~/MPI/mpi_all_reduce$ mpirun -mca btl ^openib -n 8 ./proc_min_value
Rank   0 has values:   8071   1347    839   2390   5379
Rank   1 has values:   8542   1166   3510   7451   2227
Rank   2 has values:   4341   4158   6383   1559   1566
Rank   3 has values:   2437   1848   4815   1564   2616
Rank   4 has values:   2513   5514   5345   1181    306
Rank   5 has values:   2563   5993    260   9418   9435
Rank   6 has values:   8505   2571   7500    773   5311
Rank   7 has values:    346   8964   5624   6643   5136

Rank  5 has the lowest value of 260

Rank  7 has received the value:   260
Rank  1 has received the value:   260
Rank  3 has received the value:   260
Rank  5 has received the value:   260
Rank  0 has received the value:   260
Rank  2 has received the value:   260
Rank  4 has received the value:   260
Rank  6 has received the value:   260
```

```
int localres[2];
int globalres[2];
// Compute the minimum of localarr and store the result in localres[0]
localres[0] = localarr[0];
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_Allreduce(localres, globalres, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
```

# Total exchange



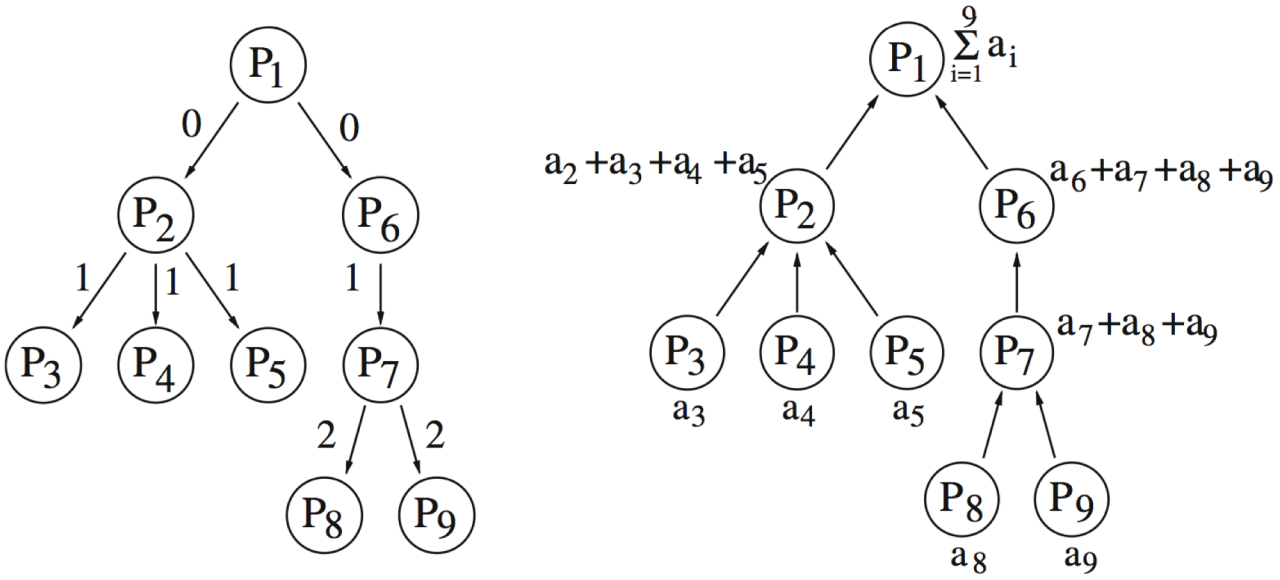

**`MPI_Alltoall()`**

Similar to a matrix transpose!

# Conceptual relations between collective communications

# Duality

- Some communication operations are dual of each other.
- Communication operations are dual if one can be obtained by reversing the direction and the sequence of communication of the other.




Left: single-broadcast operation using a spanning tree.
Right: single-accumulation that uses the same communication tree.

# Relation between collective communication operations

Specialization, e.g., multi-broadcast is the same as total exchange if the p data blocks of a process are the same.

- Dual = “reverse” flow and direction of communications
- Important to understand performance