Software

MPI: MESSAGE PASSING INTERFACE

Hajime Fujita and Wesley Bland

04/25/2019

Senior Software Development Engineer, Intel Corporation

Who Am I?

Hajime Fujita, Senior Software Development Engineer, Intel

- Working on open source enabling for MPICH

Previously worked at The University of Chicago

- Developed a resilience framework on top of MPI

PhD from The University of Tokyo, Japan

- Developed a dependable single system image OS (Linux kernel, TCP/IP)

What Have You Covered So Far?

Make Algorithms Faster

Locality, performance counters/tuning

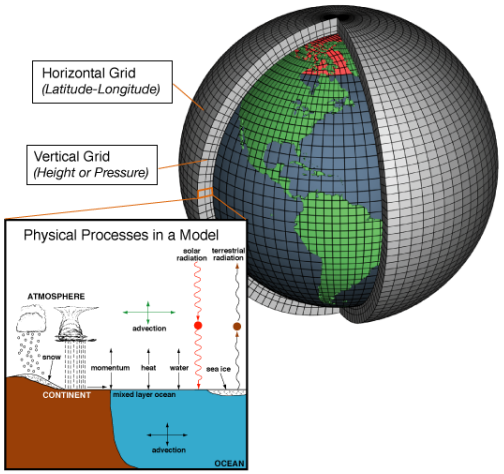
Parallelize Algorithms

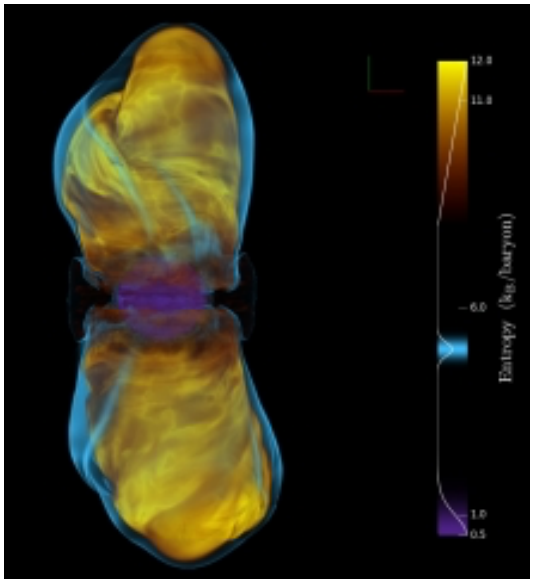
Thread-based speedups within a single node (cache coherency, synchronization)

What's Next?

Distributed Memory Programming

When the problem is too big for one node in **computational** or **memory** capacity





Images from ALCF Incite Program: https://www.alcf.anl.gov/

What is MPI?

A standard **communication interface** for **distributed memory programming**

The de-facto standard interface for modern supercomputers

- Supercomputers: massively distributed (~O(100,000) nodes)
- Programs have to communicate!

Image by Argonne National Laboratory

What Exactly is MPI?

MPI (the **Message Passing Interface**) is a standard, like C, C++, or Fortran.

- You don't download MPI just like you don't download C. It's just a document.

MPI **implementations** come as libraries

- Standard defines C, C++, and Fortran bindings
  - Many other unofficial languages bindings (e.g. Python)
- MPICH, MVAPICH2, and Open MPI are popular open source implementations that you can install on your laptops.
- Vendor implementations from Intel, Cray*, IBM*, Microsoft*, etc.

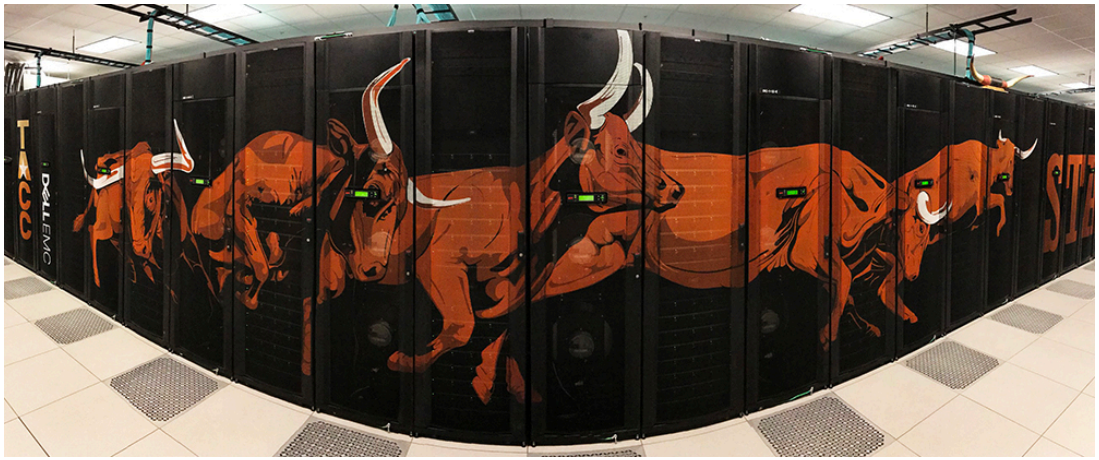
Where Can I Use MPI?

Supercomputers

Public Clouds

University/lab clusters

Laptop/desktops



Stampede2 @TACC
https://www.tacc.utexas.edu/systems/stampede2

Who Uses MPI?

- Scientific computing
  - Universities and national laboratories
- Commercial Users
  - Oil & Gas companies are a big user
- Other programming models (as the runtime for communication)
  - Parallel Global Address Space, Machine/Deep Learning Frameworks

How Do I Solve These Big Problems?

1. Stage input data on high capacity storage
2. **Distribute data across multiple machines interconnected by a high speed network**
3. **Perform computation on the input data while communicating with other processes**
4. Write the output to stable storage for analysis (sometimes by another distributed program!)

MPI helps 2. and 3.

Why Can't I Solve This in the Same Way as Before?

Previously, everything was using a shared memory model.

- All memory is addressable.
- There are some programming models that simulate this even with distributed memory (Parallel Global Address Space – PGAS).

Now we're thinking of things as distinct nodes with their own local memory.

- Only local memory is accessible.
- Other memory must be retrieved via passing messages.

How Would I Have Done This Before MPI?

- Somehow launch a bunch of processes on a bunch of nodes.
- Use BSD sockets (or something similar) to communicate data.
- Perform computation.
- Repeat. Repeat. Repeat.

- MPI makes all of the above easier.

HELLO, WORLD

MPI at a Glance

Job/process launching

- mpiexec, MPI_SPAWN, …

Point-to-point Communications for Data Transfer

- MPI_SEND, MPI_RECV, …

Collective Communications

- MPI_BARRIER, MPI_BCAST, MPI_REDUCE, …

More Advanced Communications

- One-sided (RMA), MPI-IO

How Do I Get MPI?

Remember that you're getting an **implementation** of MPI, so it won't be called MPI.

```
brew|yum|apt-get install mpich|open-mpi
```

This will install the libraries (`libmpi.so`) and the launcher (`mpiexec`).

As a student, you can get Intel® MPI Library for free with support for Linux, MacOS, and Windows:

- https://software.intel.com/en-us/qualify-for-free-software/student
- Search for "Intel MPI for Students" in your engine of choice

How Do I Use MPI

2 Steps:

1. Compile your code with the compiler wrapper for your language.
   - C – `mpicc`, C++ - `mpic++`, Fortran – `mpifort`
   - E.g. `mpicc –o my_prog my_prog.c`
   - This will automatically link with all of the right libraries. Acts just like your normal compiler (e.g. compiler flags).
2. Run your code with `mpiexec` (or something else – `mpirun`, `srun`, `aprun`).
   - `mpiexec –n 4 ./my_prog`
   - This will take care of launching your program multiple times and connecting all of them up.
   - I'll refer to this as a **job** for the rest of these slides.

My First MPI Program

```
#include <stdio.h>
#include <mpi.h>
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);
     printf("Hello, world! I'm no. %d"
            "in %d ranks.\n", rank, size);
     MPI_Finalize();
}
```

```
$ mpicc –o hello hello.c

$ mpiexec –n 4 ./hello
Hello, world! I'm no. 0 in 4 ranks
Hello, world! I'm no. 2 in 4 ranks
Hello, world! I'm no. 3 in 4 ranks
Hello, world! I'm no. 1 in 4 ranks
```

Every process executes same code "SPMD"; single program multiple data

Start up MPI

Get my rank (proc. ID) and communicator size (total no. of procs)

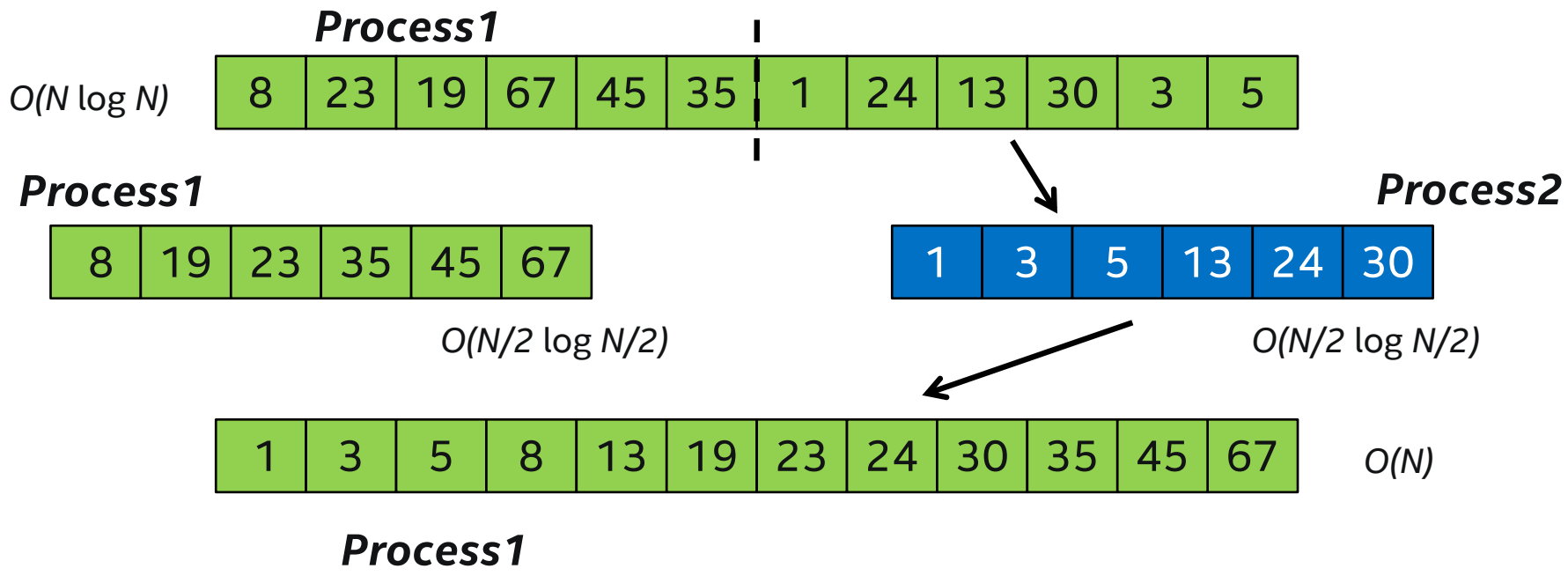
Shut down MPI

Outputs from 4 processes

POINT-TO-POINT COMMUNICATIONS (1)
BLOCKING SEND-RECEIVE

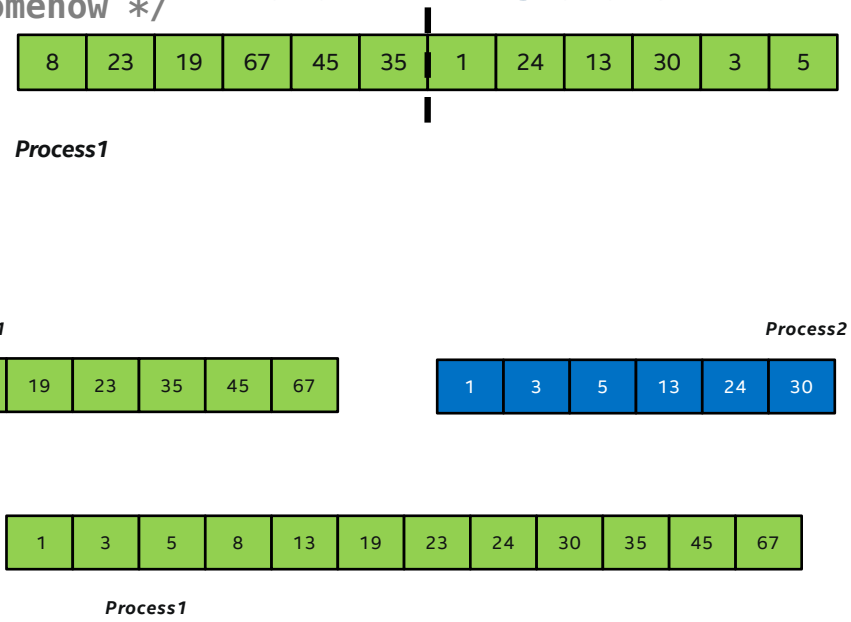
Let's Look at a Simple Example: Sorting Integers

Each process has to send/receive data to/from other processes



How Would This Look in Code?

```
#include <mpi.h>
int main(int argc, char *argv[]) {
     int rank, numbers[100]; /* Initialize numbers somehow */
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     if (rank == 0) {
          MPI_Send(&numbers[50], 50, ..., 1, ...);
     } else {
          MPI_Recv(&numbers[ 0], 50, ..., 0, ...);
     }
     sort_numbers(numbers, 50);
     if (rank == 0) {
          MPI_Recv(&numbers[50], 50, ..., 1, ...);
     } else {
          MPI_Send(&numbers[ 0], 50, ..., 0, ...);
     }
     combine_arrays(&numbers[0], &numbers[50], 50);
     MPI_Finalize();
}
```

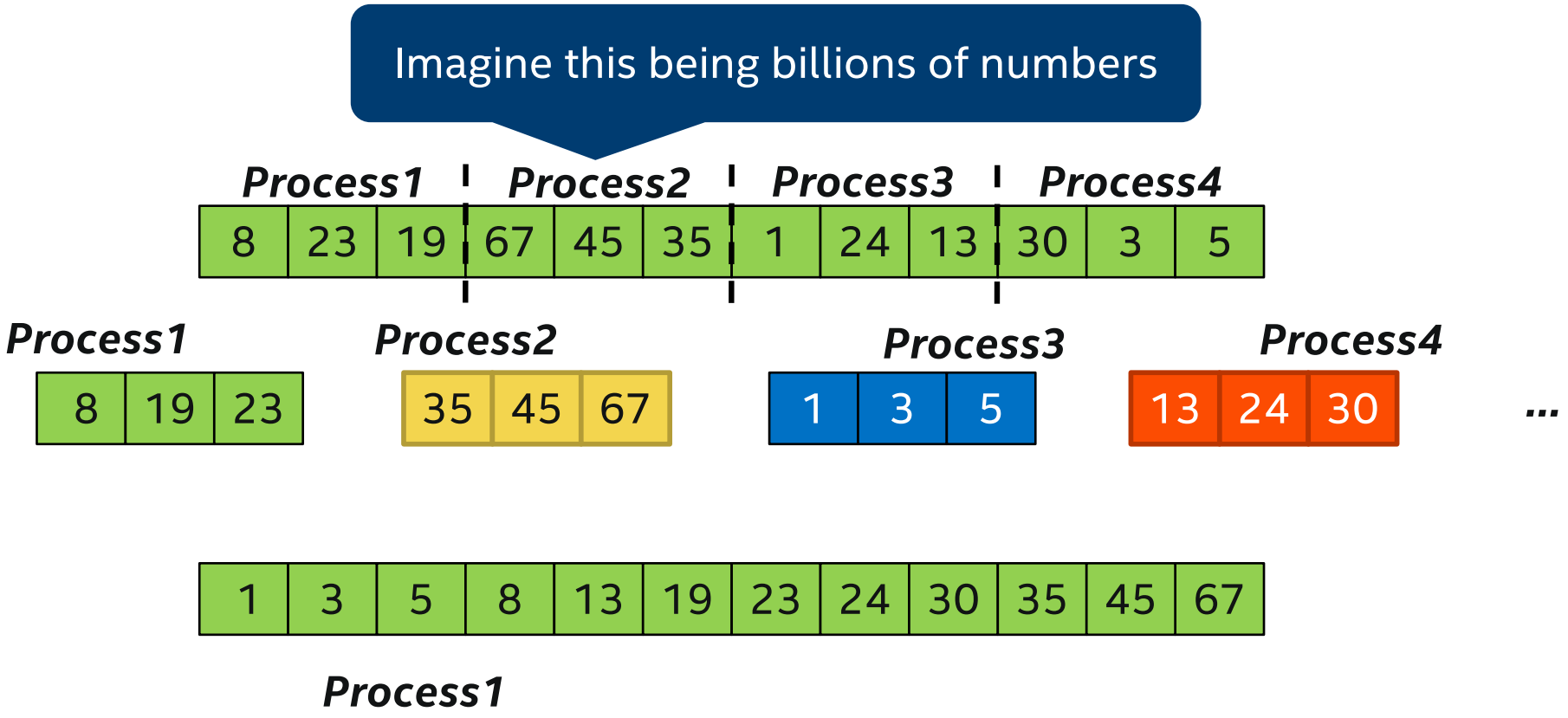


How Would This Look in Code?

```
#include <mpi.h>
int main(int argc, char *argv[]) {
     int rank, numbers[100]; /* Initialize numbers */
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     if (rank == 0) {
          MPI_Send(&numbers[50], 50, ..., 1, ...)
     } else {
          MPI_Recv(&numbers[ 0], 50, ..., 0, ...);
     }
     sort_numbers(numbers, 50);
     if (rank == 0) {
          MPI_Recv(&numbers[50], 50, ..., 1, ...);
     } else {
          MPI_Send(&numbers[ 0], 50, ..., 0, ...);
     }
     combine_arrays(&numbers[0], &numbers[50], 50);
     MPI_Finalize();
}
```

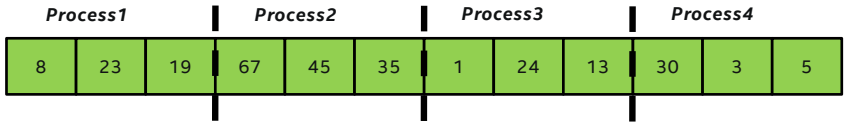
- Start up MPI
- Get my “rank” (ID)
- Distribute the initial values from rank 0 to rank 1.
- Sort
- Send the results of the sort back from rank 1 to rank 0.
- Combine the results
- Shut down MPI

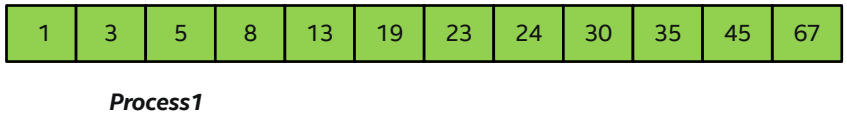
That Seems Kinda Trivial. Can We Make It Bigger?



How Would This Look in Code?

```
[...snip...]
int rank, size, numbers[100];

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

if (rank == 0) {
     for (i = 1; i < size; i++)
          MPI_Send(&numbers[(100/size)*i], (100/size), …, i, …);
else
     MPI_Recv(&numbers[0], (100/size), …);

sort_numbers(numbers, (100/size));

if (rank == 0) {
     for (i = 1; i < size; i++)
          MPI_Recv(&numbers[(100/size)*i], (100/size), …, i, …);
else
     MPI_Send(&numbers[0], (100/size), …);

combine_arrays(&numbers[0], &numbers[(100/size)], (100/size));
[...snip...]
```





How Would This Look in Code?

```
[...snip...]
int rank, size, numbers[100];

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

if (rank == 0) {
     for (i = 1; i < size; i++)
          MPI_Send(&numbers[(100/size)*i], (100/size), …, i, …);
else
     MPI_Recv(&numbers[0], (100/size), …);

sort_numbers(numbers, (100/size));

if (rank == 0) {
     for (i = 1; i < size; i++)
          MPI_Recv(&numbers[(100/size)*i], (100/size), …, i, …);
else
     MPI_Send(&numbers[0], (100/size), …);

combine_arrays(&numbers[0], &numbers[(100/size)], (100/size));
[...snip...]
```

Get my rank and size

Distribute the initial values from rank 0 to all ranks.

Send the results of the sort back from all ranks to rank 0.

Start thinking now about why this might be slow

So What Have We Seen Here?

MPI_INIT / MPI_FINALIZE

- Set up and tear down the innards of MPI

MPI_COMM_RANK / MPI_COMM_SIZE

- Get the size of my job and my rank (ID) within it

MPI_SEND / MPI_RECV

- Do some basic communication between MPI ranks

POINT-TO-POINT COMMUNICATION (2)
NON-BLOCKING SEND-RECEIVE

Blocking vs. Non-blocking Communication

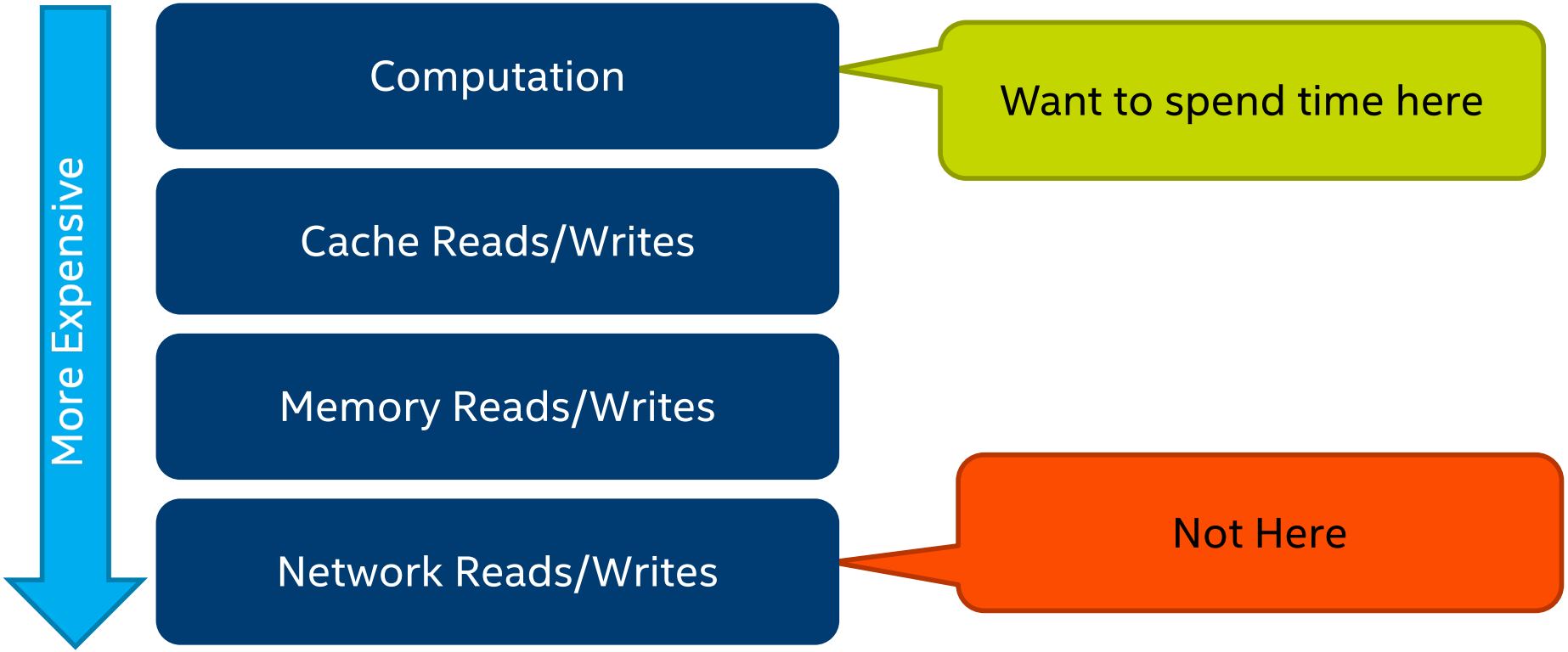
`MPI_SEND/MPI_RECV` are blocking communication calls

- Return of the routine implies completion
- When these calls return the memory locations used in the message transfer can be safely accessed for reuse
- For “send” completion implies variable sent can be reused/modified
  - Modifications will not affect data intended for the receiver
- For “receive” variable received can be read

`MPI_ISEND/MPI_IRECV` are nonblocking variants

- Routine returns immediately – completion has to be separately tested for
- These are primarily used to overlap computation and communication to improve performance

What's the Cost of Doing Computation?



Blocking Communication

In blocking communication:

- **MPI_SEND** does not return until buffer is empty (available for reuse)
- **MPI_RECV** does not return until buffer is full (available for use)

Exact completion semantics of communication generally depends on the message size and the system buffer size

Blocking communication is simple to use but can be prone to deadlocks

```
if (rank == 0) {
    MPI_SEND(..to rank 1..)
    MPI_RECV(..from rank 1..)
} else if (rank == 1) {
    MPI_SEND(..to rank 0..)
    MPI_RECV(..from rank 0..)
}
```

This will usually deadlock!

Non-Blocking Communication

Non-blocking (asynchronous) operations return requests that can be queried

- **`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`**
- **`MPI_Irecv(buf, count, datatype, src, tag, comm, request)`**
- **`MPI_Wait(request, status)`**
- **`MPI_Test(request, flag, status)`**

Non-blocking operations allow overlapping computation and communication

Anywhere you use **`MPI_SEND`** or **`MPI_RECV`**, you can use the pair of **`MPI_ISEND`** and **`MPI_WAIT`** or **`MPI_IRECV`** and **`MPI_WAIT`**

Multiple Completions

It is sometimes desirable to wait on multiple requests:

- **`MPI_Waitall(count, array_of_requests, array_of_statuses)`**
- **`MPI_Waitany(count, array_of_requests, &index, &status)`**
- **`MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`**

There are corresponding versions of **`MPI_TEST`** for each of these

Message Completion and Buffering

A send has completed when the user supplied buffer can be reused

```
*buf = 3;
MPI_Send(buf, 1, MPI_INT …)
*buf = 4; /* OK, receiver will always
             receive 3 */
```

```
*buf = 3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /* Receiver may get 3, 4, or
             anything else */
MPI_Wait(…);
```

Just because the send completes does not mean that the receive has completed

- Message may be buffered by the system
- Message may still be in transit

A Non-Blocking communication example

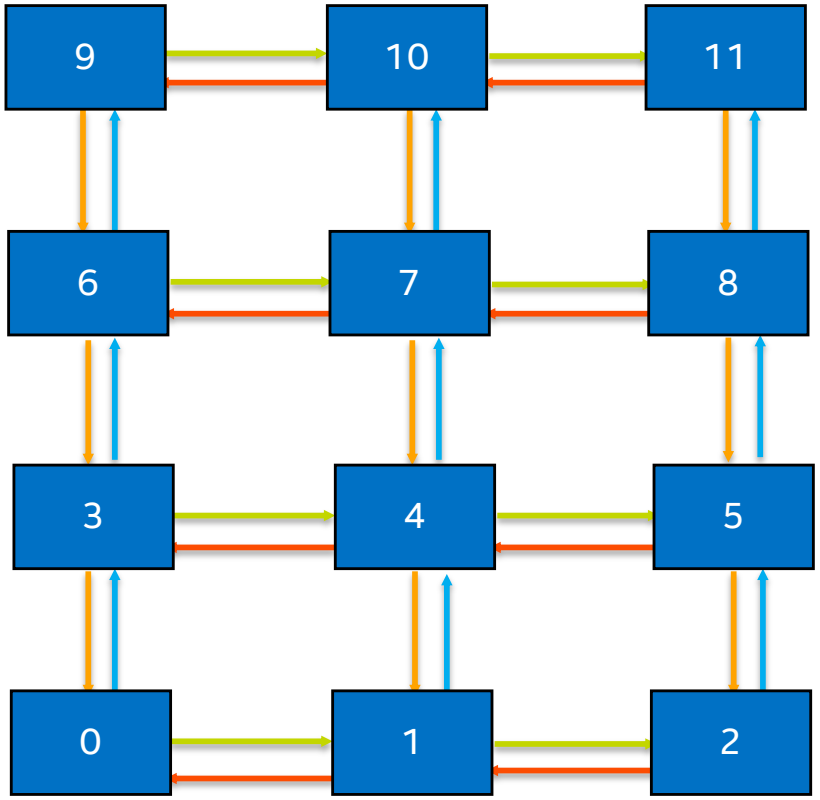
```
[...snip...]

/* Compute each data element and send it out */
if (rank == 0) {

     for (i=0; i< 100; i++) {
          data[i] = compute(i);
          MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request[i]);
     }
     MPI_Waitall(100, request, MPI_STATUSES_IGNORE)

} else if (rank == 1) {

     for (i = 0; i < 100; i++)
          MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

}

[...snip...]
```

Mesh Exchange

Let's imagine communicating with a mesh of processes.

Very common type of application communication pattern.

- Also known as a stencil (can be multiple dimensions)



Sample Code

What is wrong with this code?

```
for (i = 0; i < n_neighbors; i++) {
    MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
for (i = 0; i < n_neighbors; i++) {
    MPI_Recv(edge, len, MPI_DOUBLE, nbr[i], tag, comm, status);
}
```

Sample Code

What is wrong with this code?

```
for (i = 0; i < n_neighbors; i++) {
    MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
for (i = 0; i < n_neighbors; i++) {
    MPI_Recv(edge, len, MPI_DOUBLE, nbr[i], tag, comm, status);
}
```

Deadlocks!
All of the sends may block, waiting for a matching receive (will if large enough messages)

Fix 1: Swap Send and Recv

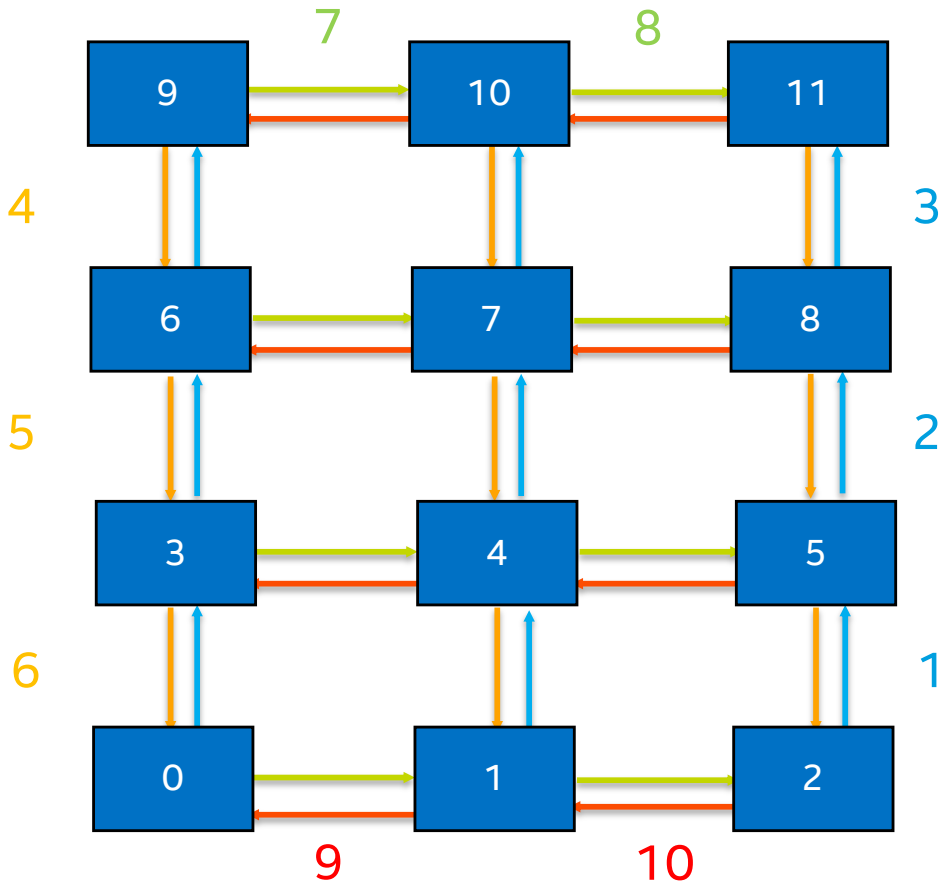
This variation solves the problem.

```
if (has up neighbor)
    MPI_Recv(…up…)
else
    MPI_Send(…down…)
```

But it introduces a performance problem.

Can anyone say what it is?

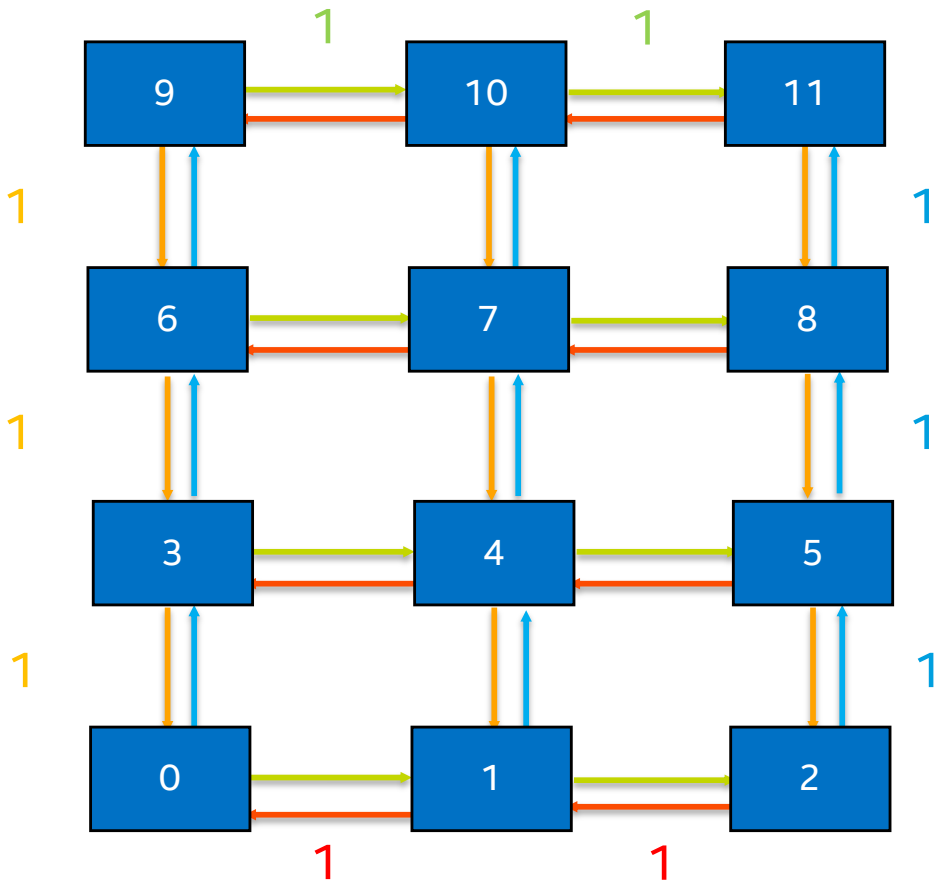
Sequentializes Communication



Fix 2: Use Isend and Irecv

```
for (i = 0; i < n_neighbors; i++) {
    MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
              comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
    MPI_Isend(edge, len, MPI_DOUBLE, nbr[i], tag, comm,
              requests[n_neighbors + i]);
}
MPI_Waitall(2 * n_neighbors, requests, statuses);
```

Parallelizes Communication



Lesson: Defer Synchronization

Send-receive accomplishes two things:

- Data transfer
- Synchronization

In many cases, there is more synchronization than required

Use non-blocking operations and **`MPI_Waitall`** to defer synchronization

Tools can help out with identifying performance issues

- Intel® Trace Analyzer and Collector (ITAC), Tau, HPCToolkit, and Scalasca are popular profiling tools

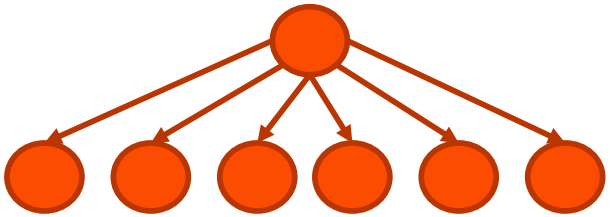
COLLECTIVE COMMUNICATIONS

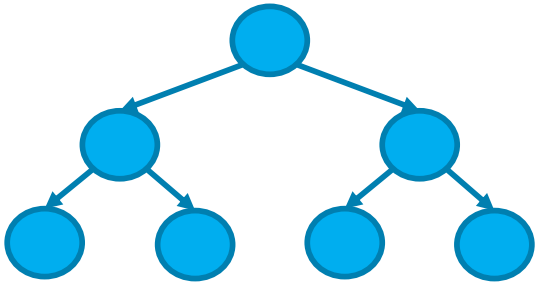
What Are Collectives?

A group of processes works together to accomplish something.

- E.g. Calculate a value, distribute some data, gather a result, synchronize operations, etc.

Instead of sending a value to each process in a `for` loop, use one **collective** call and let MPI optimize doing that for you.





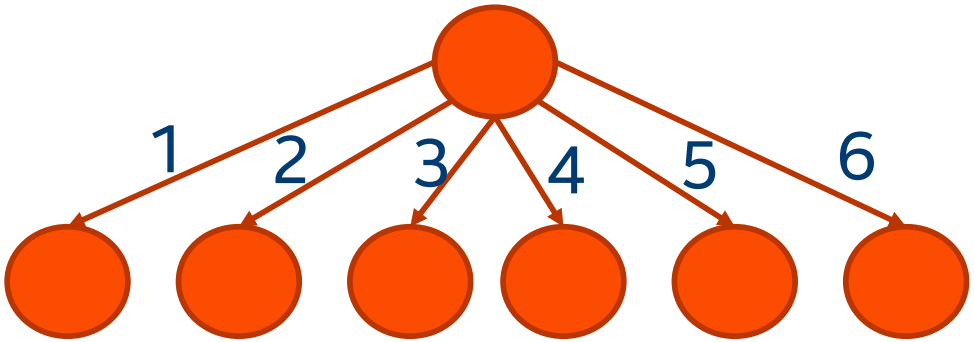
Sorting Example (Again)

```
[...snip...]
int rank, size, numbers[100];

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

if (rank == 0) {
     for (i = 1; i < size; i++)
          MPI_Send(&numbers[(100/size)*i], (100/size), …, i, …);
else
     MPI_Recv(&numbers[0], (100/size), …);

sort_numbers(numbers, (100/size));

if (rank == 0) {
     for (i = 1; i < size; i++)
          MPI_Recv(&numbers[(100/size)*i], (100/size), …, i, …);
else
     MPI_Send(&numbers[0], (100/size), …);

combine_arrays(&numbers[0], &numbers[(100/size)], (100/size));
[...snip...]
```

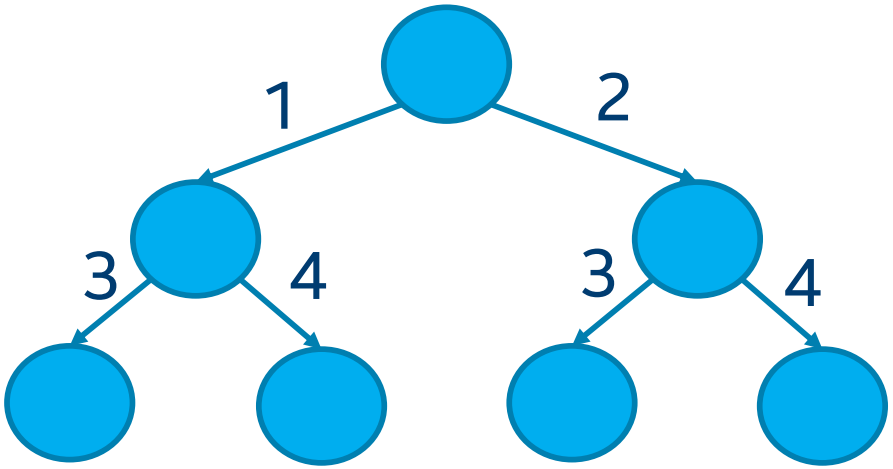
Distribute the initial values from rank 0 to all ranks.

Send the results of the sort back from all ranks to rank 0.

Why is the right side better?

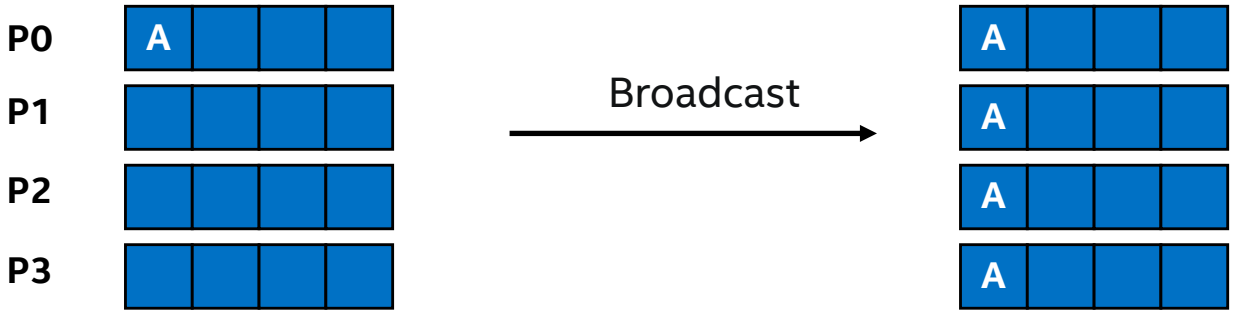


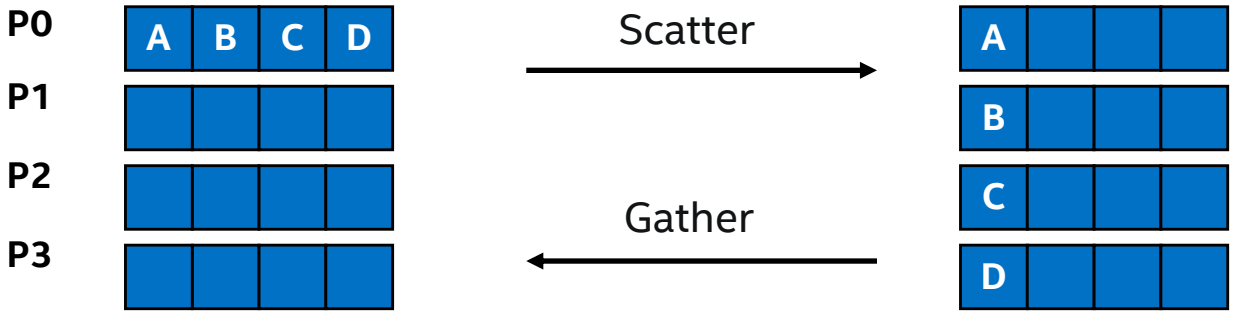
O(n) serial communications



2 serial communications (best case)
4 serial communications (worst case)
O(log(n)) (average case)

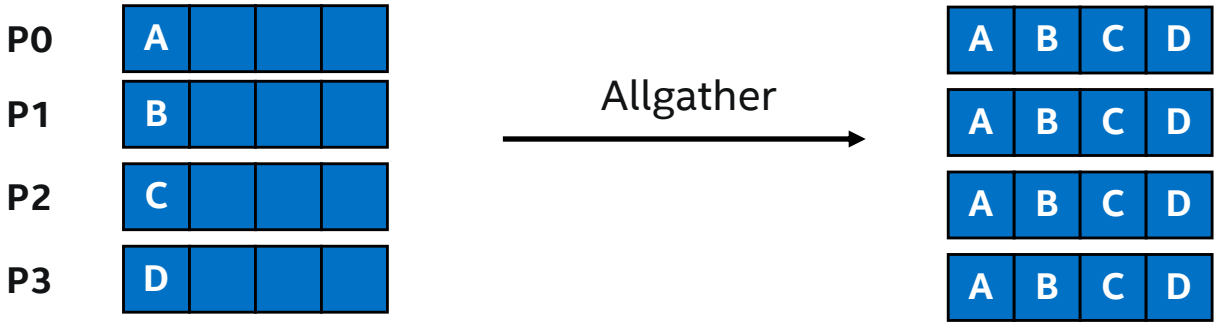
Collective Data Movement

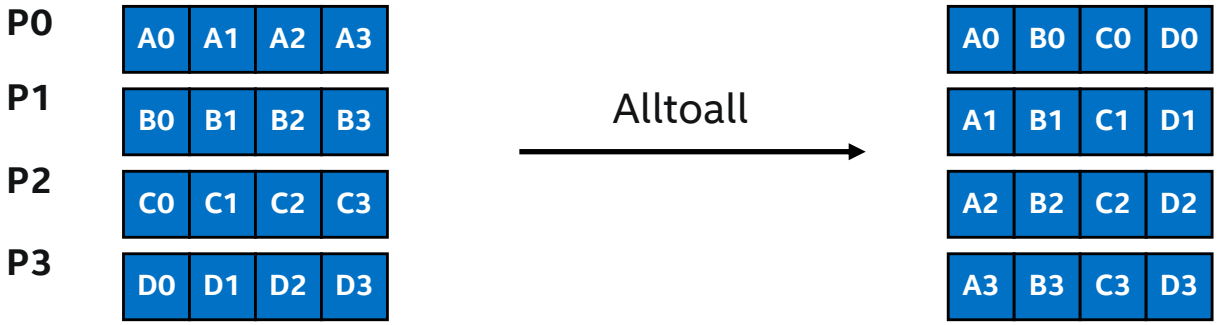




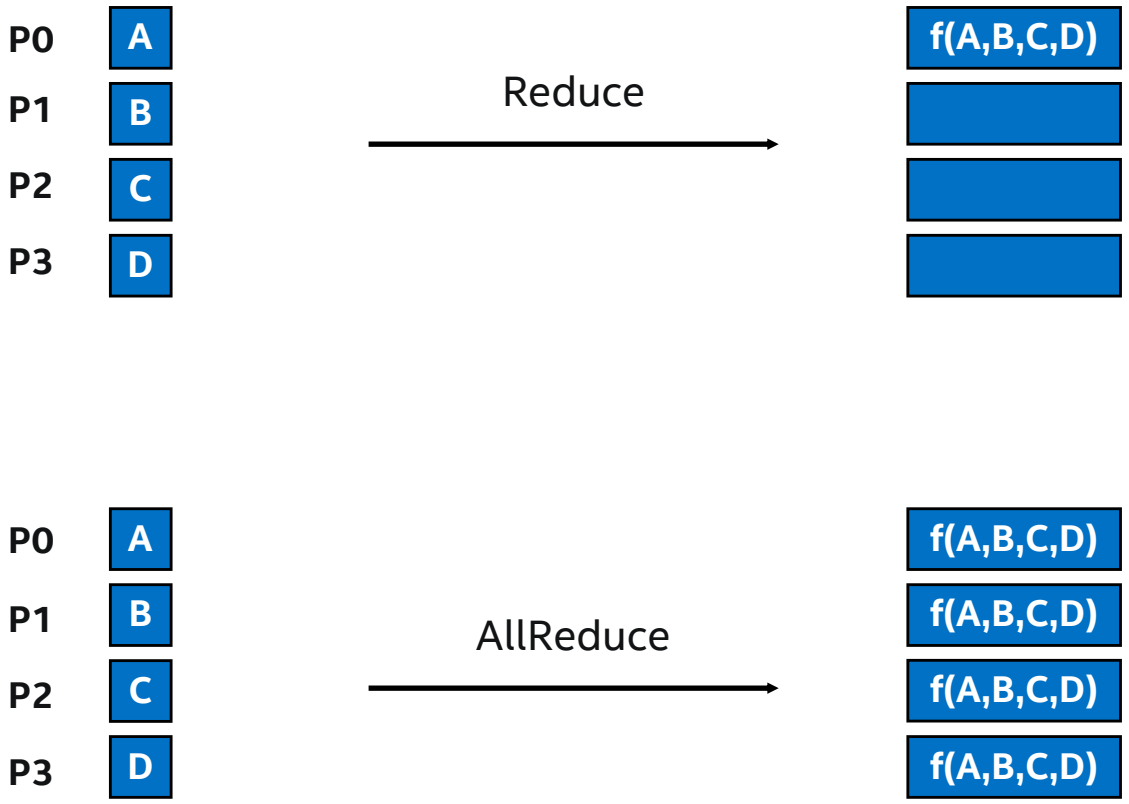
This is why sending and receiving integers individually for sorting is slow

More Collective Data Movement ("All"-Variants)





Collective Computation



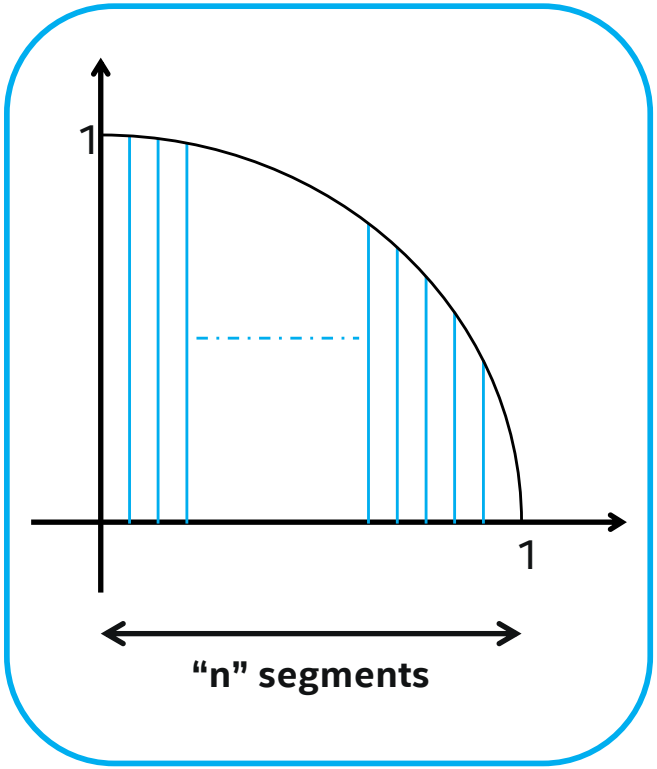
What Is That Reduction Thing Again?

Perform operations on data while communicating to one (or all) processes(es)

MPI_MAX	Maximum	**MPI_BAND**	Bitwise AND
MPI_MIN	Minimum	**MPI_BOR**	Bitwise OR
MPI_PROD	Product	**MPI_BXOR**	Bitwise Exclusive OR
MPI_SUM	Sum	**MPI_MAXLOC**	Maximum and location
MPI_LAND	Logical AND	**MPI_MINLOC**	Minimum and location
MPI_LOR	Logical OR	**USER**	User defined operation
MPI_LXOR	Logical Exclusive OR		

Example: Calculating π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



Calculating π via numerical integration

- Divide interval up into subintervals
- Assign subintervals to processes
- Each process calculates partial sum
- Add all the partial sums together to get π

1. Width of each segment (`w`) will be `1/n`
2. Distance (`d(i)`) of segment "`i`" from the origin will be "`i * w`"
3. Height of segment "`i`" will be `sqrt(1 – [d(i)]^2)`

Example: Pi in C

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[]) {
     double mypi = 0.0;
     [...snip...]

     MPI_Bcast(&num_segs, 1, MPI_INT, 0, MPI_COMM_WORLD);

     double width = 1.0 / (double) num_segs;
     for (int i = rank + 1; i <= n; i += size)
          mypi += width * sqrt(1 – (((double) i / num_segs) * ((double) i / num_segs));

     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

     if (rank == 0)
          printf("pi is approximately %.16f, Error is %.16f\n",
                 4 * pi, fabs((4 * pi) – PI25DT));
     [...snip...]
}
```

Example: Pi in C

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[]) {
     double mypi = 0.0;
     [...snip...]

     MPI_Bcast(&num_segs, 1, MPI_INT, 0, MPI_COMM_WORLD);

     double width = 1.0 / (double) num_segs;
     for (int i = rank + 1; i <= n; i += size)
          mypi += width * sqrt(1 – (((double) i / num_segs) * ((double) i / num_segs));

     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

     if (rank == 0)
          printf("pi is approximately %.16f, Error is %.16f\n",
                 4 * pi, fabs((4 * pi) – PI25DT));
     [...snip...]
}
```

Tell all processes how many rectangles there are

Calculate my share of pi

Send the result to rank 0 **and** calculate the total at the same time

SUMMARY

That Was A Lot!

We covered:

- What is MPI (and the implementations of it)?
- Startup & Finalize
- Blocking Send & Receive
- Non-blocking Send & Receive
- Collectives

Next Time

- Communicators
- Datatypes
- Brief look at advanced topics (RMA, threads, topology)
- Analyzing the performance of MPI


intel
Software