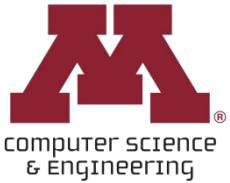
CSCI 5451: Introduction to Parallel Computing

Lecture 11: MPI in Practice



UNIVERSITY OF MINNESOTA
Driven to Discover®

Announcements (10/08)

Check the slack channel + course website for announcements later today on …

- ❑ HW1
  - Local Autograder - We will release a local autograder so you can more directly test the correctness of your implementation in line with the canvas autograder
  - Canvas Autograder - Will be run Friday/Saturday/Sunday between 6am-noon → submit early to assess your work on hidden tests & see full grade
- ❑ Project
  - PDF release later today
  - Timeline
    - ✔ Team Formation [3-5 people] (Oct 19)
    - ✔ Initial Project Idea (Nov 9)
    - ✔ Final Project Writeup + Code Due (Dec 14)

Lecture Overview

- Running an MPI Program in Practice
- Odd-Even Sort Example
- Topologies
- Non-Blocking MPI Communications

Lecture Overview

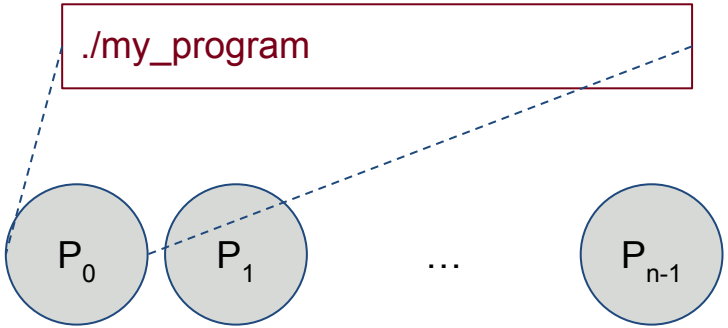
- **Running an MPI Program in Practice**
- Odd-Even Sort Example
- Topologies
- Non-Blocking MPI Communications

Command Line From C to MPI

- When we run a program on a shared address space machine in C, we can just call the executable directly with the necessary command line arguments
- This is not possible in MPI as ***this will only launch a single process*** on one machine
- We need to introduce some new logic

Command Line From C to MPI

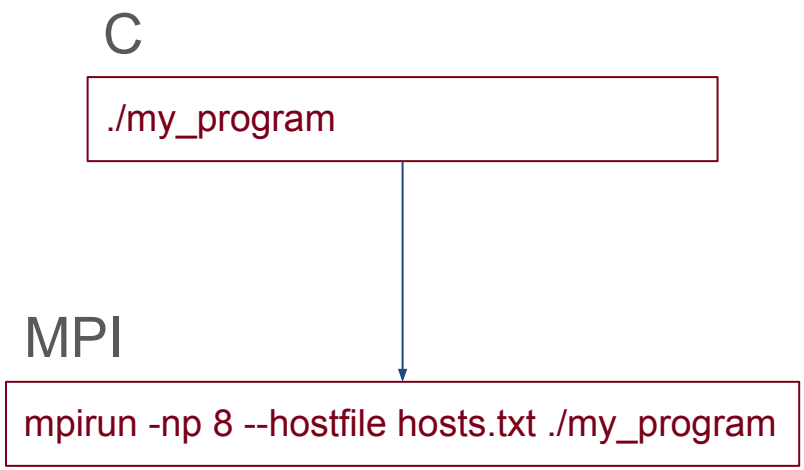
- When we run a program on a shared address space machine in C, we can just call the executable directly with the necessary command line arguments
- This is not possible in MPI as ***this will only launch a single process*** on one machine
- We need to introduce some new logic



- If we only run this command from processor P_0, the other processors will not be executed.
- MPI ***does not*** ssh onto other machines to launch the program using just ***MPI_Init***
- We need to introduce some special command line tools & configuration files

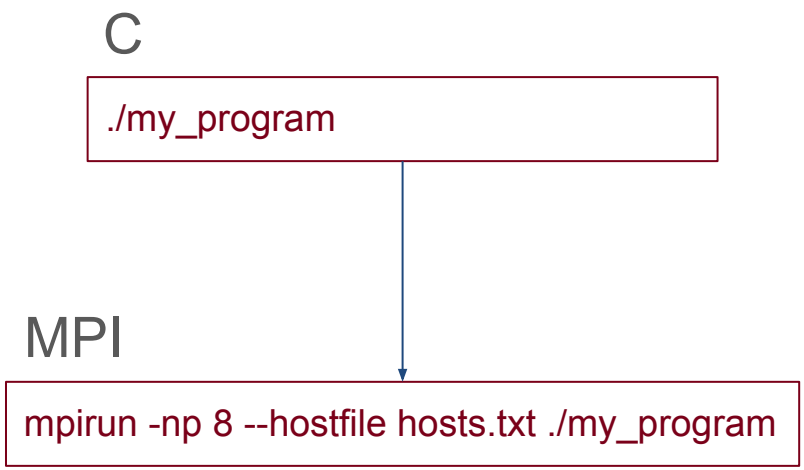
Command Line From C to MPI

- When we run a program on a shared address space machine in C, we can just call the executable directly with the necessary command line arguments
- This is not possible in MPI as ***this will only launch a single process*** on one machine
- We need to introduce some new logic



Command Line From C to MPI

- When we run a program on a shared address space machine in C, we can just call the executable directly with the necessary command line arguments
- This is not possible in MPI as ***this will only launch a single process*** on one machine
- We need to introduce some new logic



We'll walk through each part of this command line to better understand how to launch a program

Running MPI in Practice

```
mpirun -np 8 \
        --hostfile hosts.txt  \
        ./my_program
```

- The *mpirun* command
- The program to execute to parallel
- Specifying the processors to run the program on
- Specifying the number of processors to use

Running MPI in Practice

```
mpirun -np 8 \
       --hostfile hosts.txt  \
       ./my_program
```

- The *mpirun* command
- The program to execute to parallel
- Specifying the processors to run the program on
- Specifying the number of processors to use

Programs must be launched with the ***mpirun*** command

Running MPI in Practice

```
mpirun -np 8 \
        --hostfile hosts.txt  \
        ./my_program
```

- ❑ The *mpirun* command
- ❑ The program to execute to parallel
- ❑ Specifying the processors to run the program on
- ❑ Specifying the number of processors to use

The program we wish to execute in parallel. This executable must be present ***on each processor in the same location*** in order for the program to execute properly.

Running MPI in Practice

```
mpirun -np 8 \
        --hostfile hosts.txt  \
        ./my_program
```

- ❑ The *mpirun* command
- ❑ The program to execute to parallel
- ❑ Specifying the processors to run the program on
- ❑ Specifying the number of processors to use

How can we set this up so that compiled programs are on each processor …

Compiling an MPI Program

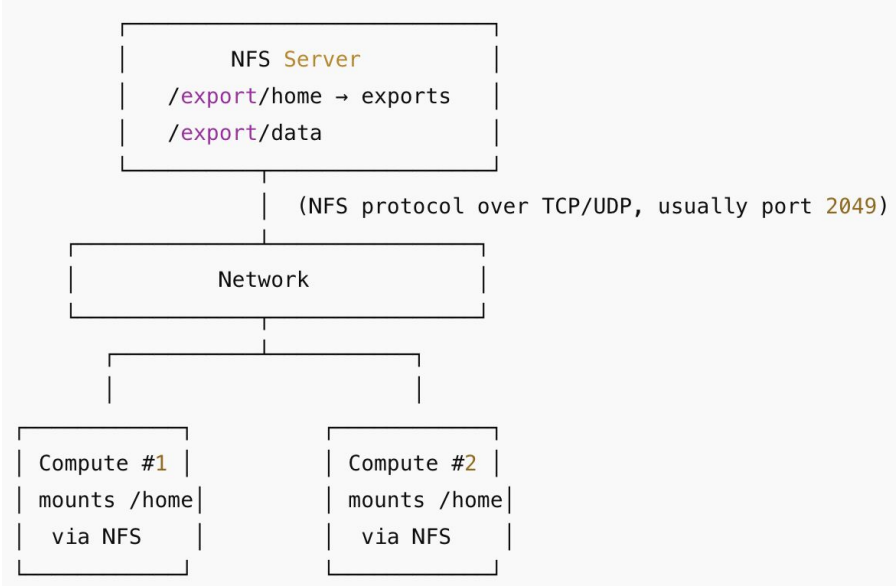
mpicc is used to compile programs using MPI

```
mpicc -O3 -o myprog myprog.c
```

How do we ensure our compiled program is present on each node?

Ensuring Binary is Available

- Most HPC clusters use what is called a Networked File System (NFS)
- A single server (oftentimes external to the cluster) stores the filesystem
- Each node remotely mounts this filesystem
- The ***plate*** servers use NFS, so if you compile on one machine, it will be accessible on all other machines
- In cases where the system does not use NFS, you will have to manually recompile or copy the executable on different machines



Running MPI in Practice

```
mpirun -np 8 \
        --hostfile hosts.txt \
        ./my_program
```

- ❑ The *mpirun* command
- ❑ The program to execute to parallel
- ❑ Specifying the processors to run the program on
- ❑ Specifying the number of processors to use

We specify the set of processors (also called ‘nodes’) to use in a file called *hosts.txt*

--hostfile

- MPI needs to know which processors (nodes) to run your program on
- Each line of this file has two things
  - The hostname you want MPI to make use of (e.g. csel-plate01.cselabs.umn.edu)
  - How many processes you want to run on each processor (the *slots* arg)

```
# hosts.txt
csel-plate01.cselabs.umn.edu slots=4
csel-plate02.cselabs.umn.edu slots=4
csel-plate03.cselabs.umn.edu slots=4
```

--hostfile

- MPI needs to know which processors (nodes) to run your program on
- Each line of this file has two things
  - The hostname you want MPI to make use of (e.g. csel-plate01.cselabs.umn.edu)
  - How many processes you want to run on each processor (the *slots* arg)

```
# hosts.txt
csel-plate01.cselabs.umn.edu slots=4
csel-plate02.cselabs.umn.edu slots=4
csel-plate03.cselabs.umn.edu slots=4
```

Note that you can launch more than one (1) process on each processor. You should keep the number of physical cores on each processor in mind when setting this value so as to not exceed the capacity of machine.

Running MPI in Practice

```
mpirun -np 8 \
        --hostfile hosts.txt  \
        ./my_program
```

- ❑ The *mpirun* command
- ❑ The program to execute to parallel
- ❑ Specifying the processors to run the program on
- ❑ Specifying the number of processors to use

The number of processes to launch in total. MPI will walk sequentially through the *hosts.txt* file, using up all *slots* on a processor before moving to the next processor.

Lecture Overview

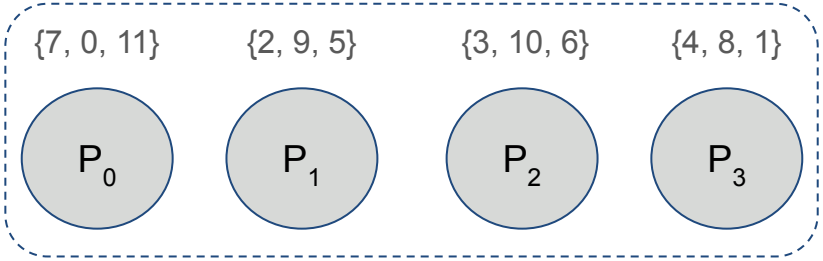
- Running an MPI Program in Practice
- **Odd-Even Sort Example**
- Topologies
- Non-Blocking MPI Communications

Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

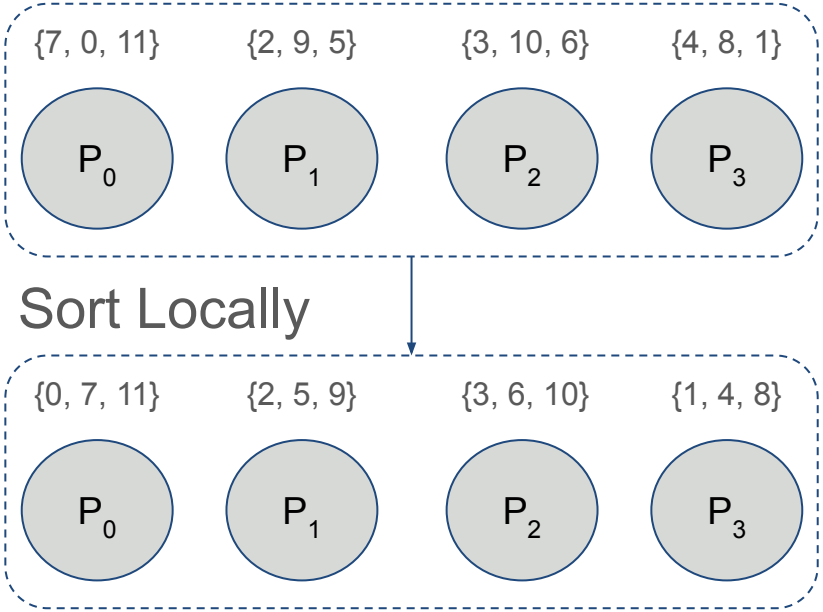
Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

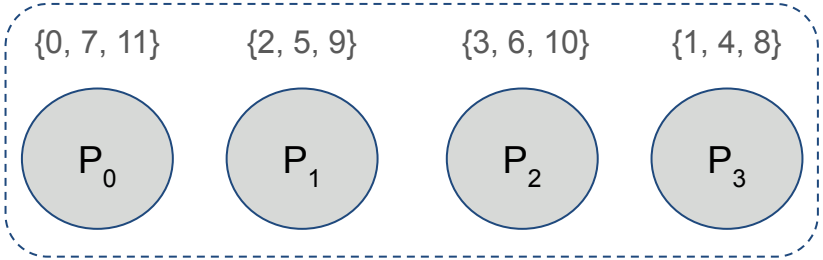


Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on



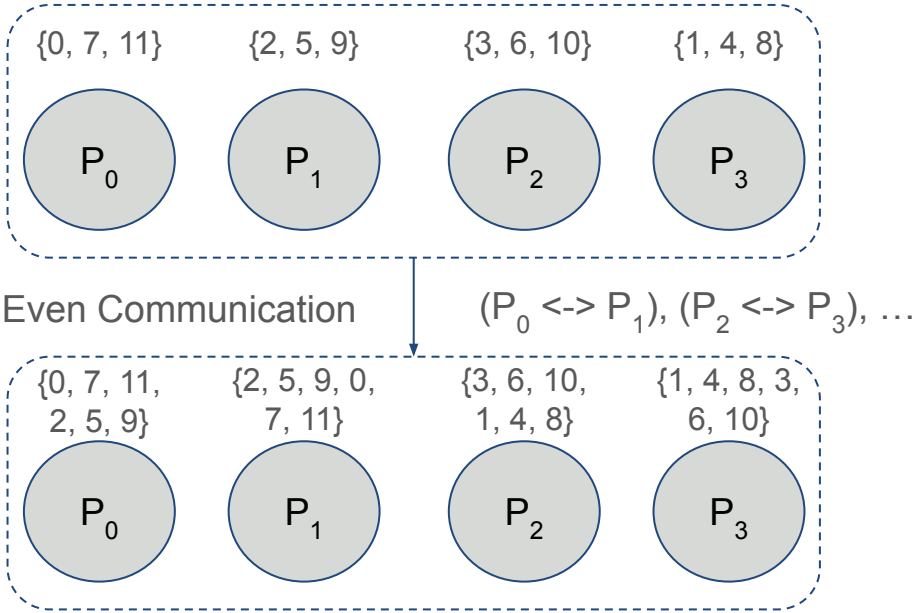
Odd-Even Sort



- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

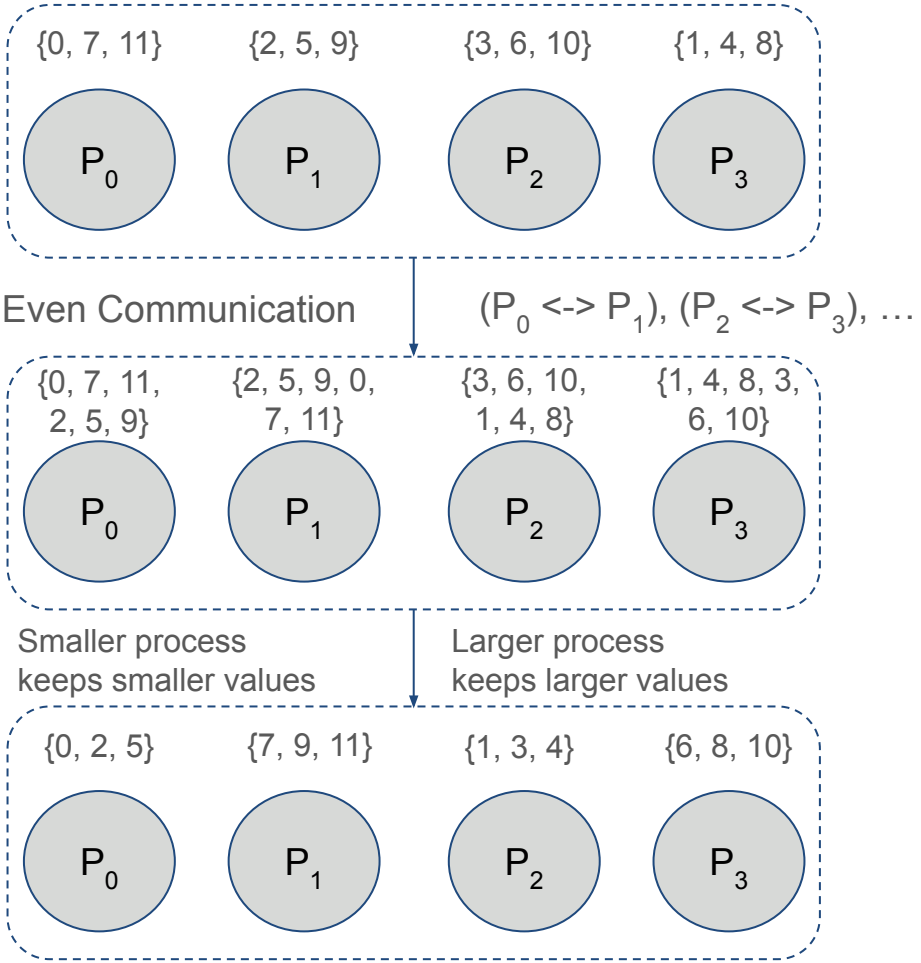
Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

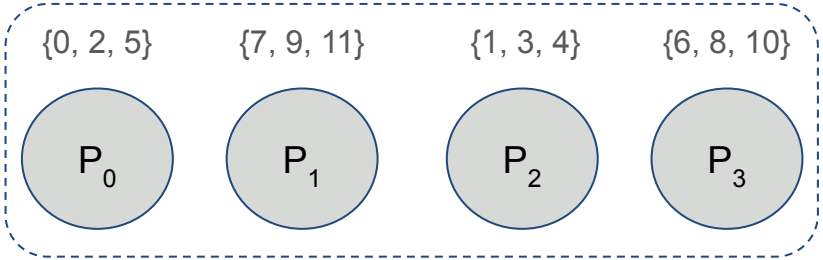


Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on



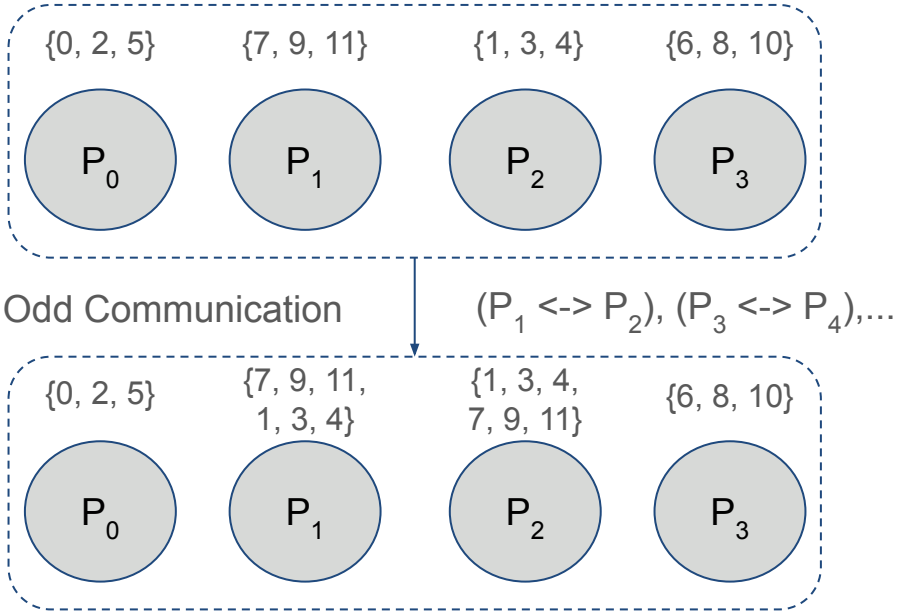
Odd-Even Sort



- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

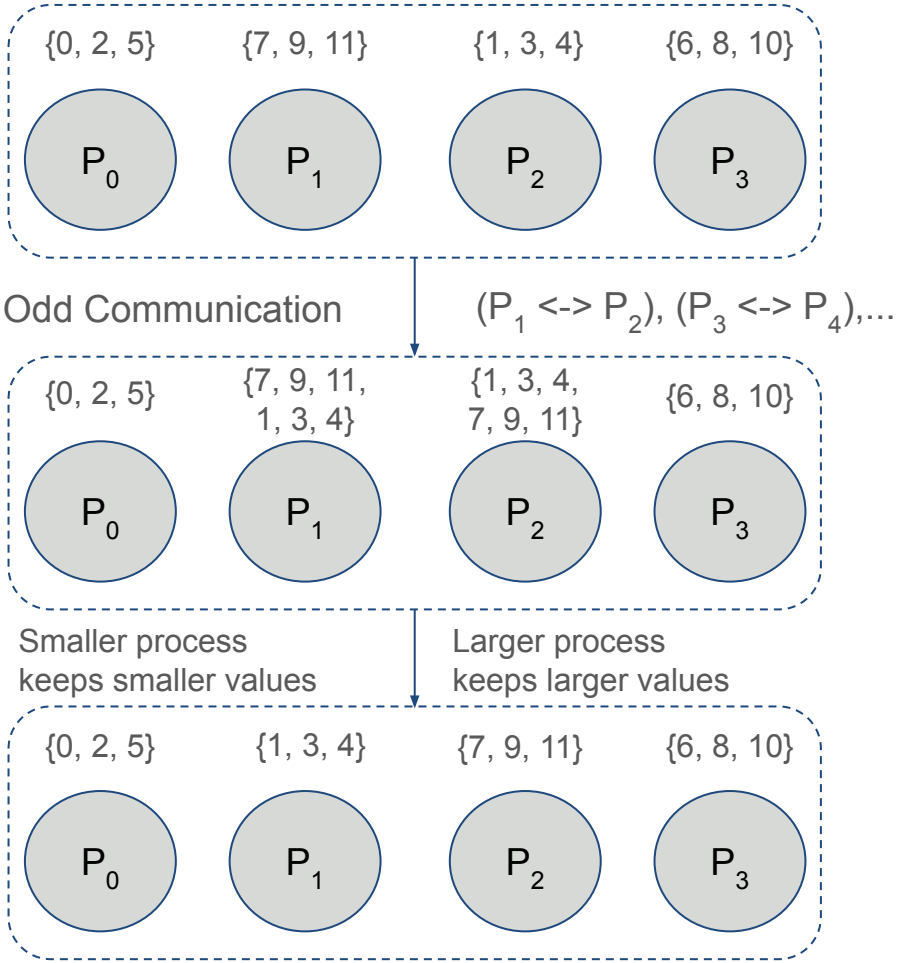
Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

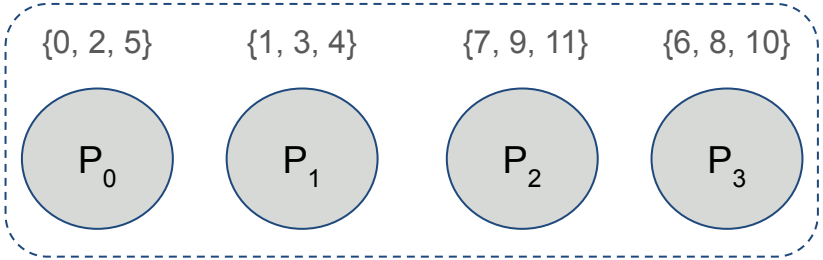


Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on



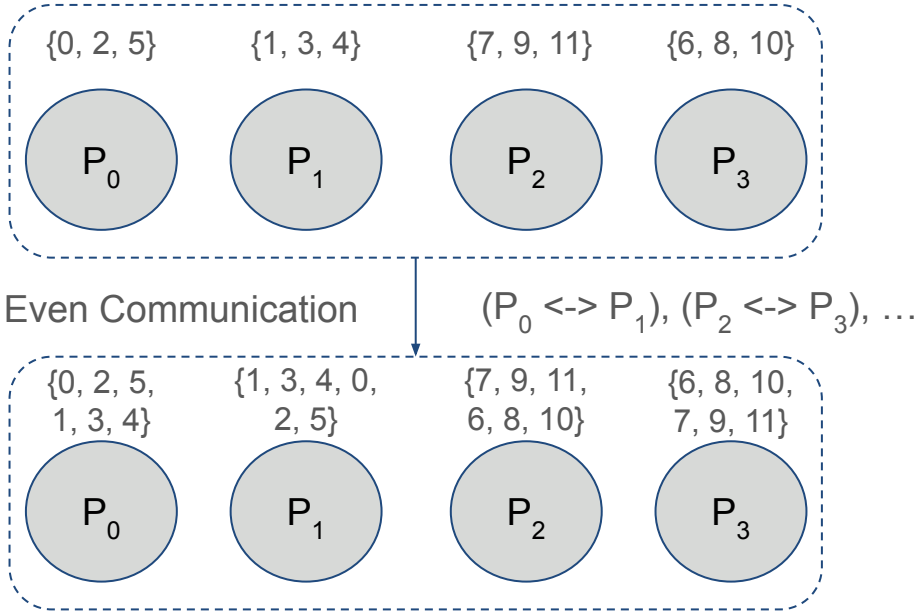
Odd-Even Sort



- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

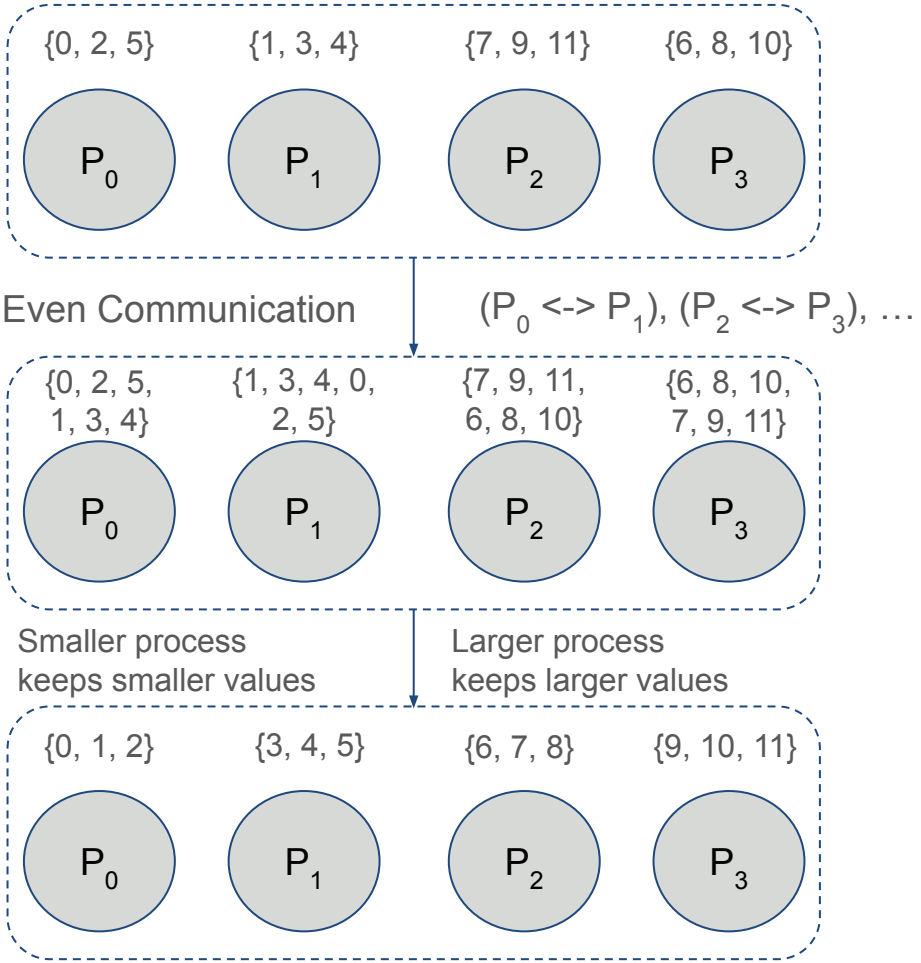
Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on



Odd-Even Sort

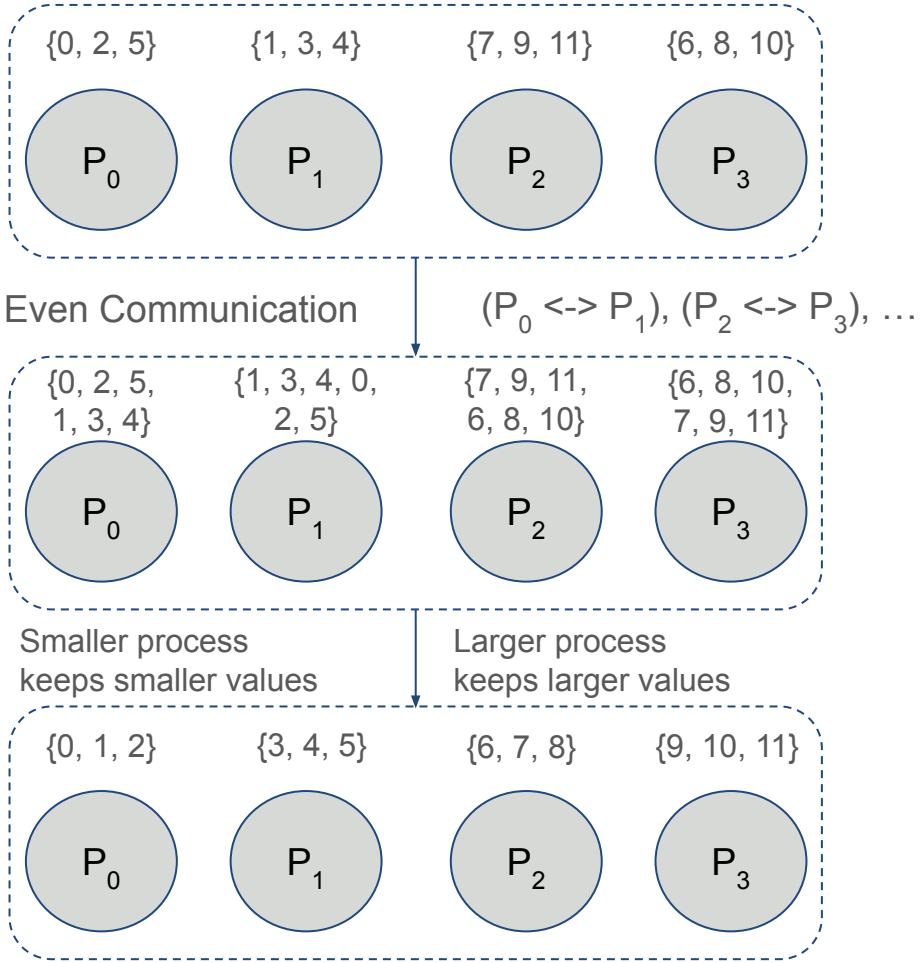
- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on



Odd-Even Sort

- Suppose we have an array of 12 elements, split across 4 processes
- We walk through how to sort this array such that P_0 has the first 3 elements in the array, P_1 the next 3, and so on

Sorted after *p-1* steps



Odd-Even Sort (MPI)

- ***CompareSplit*** keeps either the largest or smallest elements after communication
- ***IncOrder*** is a utility function for use with ***qsort***

```
/* This is the CompareSplit function */
CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
             int keepsmall)
{
  int i, j, k;

  for (i=0; i<nlocal; i++)
    wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */

  if (keepsmall) { /* Keep the nlocal smaller elements */
    for (i=j=k=0; k<nlocal; k++) {
      if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
        elmnts[k] = wspace[i++];
      else
        elmnts[k] = relmnts[j++];
    }
  }
  else { /* Keep the nlocal larger elements */
    for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
      if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
        elmnts[k] = wspace[i--];
      else
        elmnts[k] = relmnts[j--];
    }
  }
}

/* The IncOrder function that is called by qsort is defined as follows */
int IncOrder(const void *e1, const void *e2)
{
  return (*((int *)e1) - *((int *)e2));
}
```

Odd-Even Sort (MPI)

```
main(int argc, char *argv[])
{
  int n;          /* The total number of elements to be sorted */
  int npes;       /* The total number of processes */
  int myrank;     /* The rank of the calling process */
  int nlocal;     /* The local number of elements, and the array that stores th
  int *elmnts;    /* The array that stores the local elements */
  int *relmnts;   /* The array that stores the received elements */
  int oddrank;    /* The rank of the process during odd-phase communication */
  int evenrank;   /* The rank of the process during even-phase communication */
  int *wspace;    /* Working space during the compare-split operation */
  int i;
  MPI_Status status;

  /* Initialize MPI and get system information */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  n = atoi(argv[1]);
  nlocal = n/npes; /* Compute the number of elements to be stored locally. */
```

Odd-Even Sort (MPI)

```
/* Allocate memory for the various arrays */
elmnts  = (int *)malloc(nlocal*sizeof(int));
relmnts = (int *)malloc(nlocal*sizeof(int));
wspace  = (int *)malloc(nlocal*sizeof(int));

/* Fill-in the elmnts array with random elements */
srandom(myrank);
for (i=0; i<nlocal; i++)
  elmnts[i] = random();

/* Sort the local elements using the built-in quicksort routine */
qsort(elmnts, nlocal, sizeof(int), IncOrder);

/* Determine the rank of the processors that myrank needs to communicate du

/* odd and even phases of the algorithm */
if (myrank%2 == 0) {
  oddrank  = myrank-1;
  evenrank = myrank+1;
}
else {
  oddrank  = myrank+1;
  evenrank = myrank-1;
}
```

Odd-Even Sort (MPI)

```
  /* Set the ranks of the processors at the end of the linear */
  if (oddrank == -1 || oddrank == npes)
    oddrank = MPI_PROC_NULL;
  if (evenrank == -1 || evenrank == npes)
    evenrank = MPI_PROC_NULL;

  /* Get into the main loop of the odd-even sorting algorithm */
  for (i=0; i<npes-1; i++) {
    if (i%2 == 1) /* Odd phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
          nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
    else /* Even phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
          nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

     CompareSplit(nlocal, elmnts, relmnts, wspace,
                  myrank < status.MPI_SOURCE);
  }

  free(elmnts); free(relmnts); free(wspace);
  MPI_Finalize();
}
```

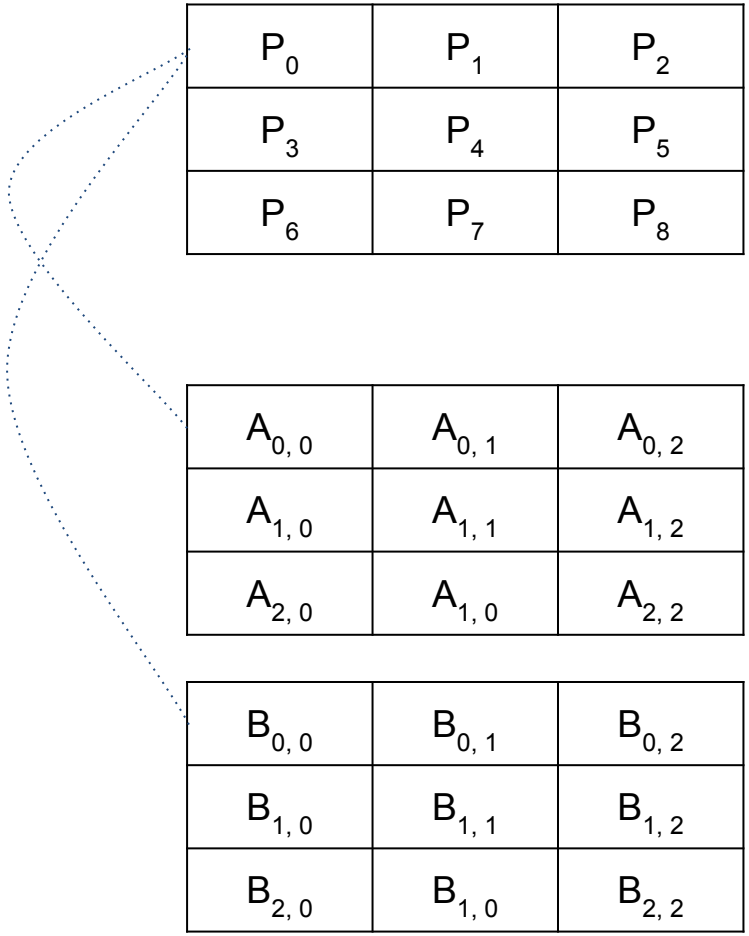
Lecture Overview

- Running an MPI Program in Practice
- Odd-Even Sort Example
- **Topologies**
- Non-Blocking MPI Communications

To better understand what *Topologies* in MPI are, let's start by examining an algorithm for parallelizing Matrix-Matrix multiplication

Cannon's Algorithm

- Suppose I want to multiply two *n x n* matrices **A** and **B**
- Further suppose these matrices are too big to fit on any one processor
- We can use a block 2-d distribution to split up **A** and **B** among p processors
- Each processor can then compute one block of output **C**…



Cannon's Algorithm

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$
$A_{2,0}$	$A_{1,0}$	$A_{2,2}$

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$
$B_{2,0}$	$B_{1,0}$	$B_{2,2}$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$
$C_{2,0}$	$C_{1,0}$	$C_{2,2}$

P_0 $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0}$

P_1 $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1}$

…

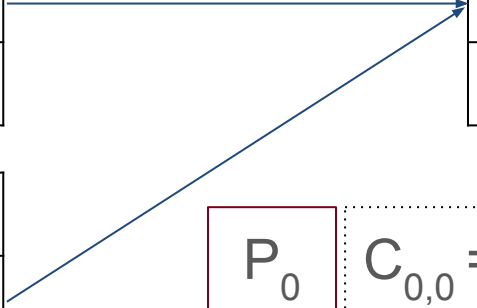
Cannon's Algorithm

All **bolded** values are present on other processes - we have to communicate them

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$
$A_{2,0}$	$A_{1,0}$	$A_{2,2}$

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$
$B_{2,0}$	$B_{1,0}$	$B_{2,2}$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$
$C_{2,0}$	$C_{1,0}$	$C_{2,2}$



P_0 $C_{0,0} = A_{0,0}B_{0,0} + \mathbf{A_{0,1}B_{1,0}} + \mathbf{A_{0,2}B_{2,0}}$

P_1 $C_{0,1} = \mathbf{A_{0,0}}B_{0,1} + A_{0,1}\mathbf{B_{1,1}} + \mathbf{A_{0,2}B_{2,1}}$

...

Cannon's Algorithm

- We can decompose this problem into sqrt(p) separate local matrix-multiplies
- Further we will have to perform sqrt(p) separate communications

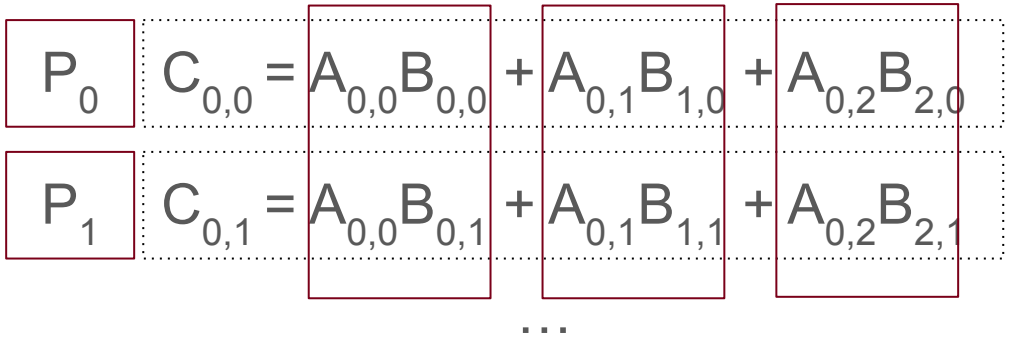
P_0 $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0}$

P_1 $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1}$

…

Cannon's Algorithm

- We can decompose this problem into sqrt(p) separate local matrix-multiplies
- Further we will have to perform sqrt(p) separate communications



Cannon's Algorithm

Can we design our communications/computations such that all processes communicate at the same time, and only with adjacent processes?

- We can decompose this problem into sqrt(p) separate local matrix-multiplies
- Further we will have to perform sqrt(p) separate communications

P_0 $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0}$

P_1 $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1}$

…

Cannon's Algorithm

Can we design our communications/computations such that all processes communicate at the same time, and only with adjacent processes?

Yes

- We can decompose this problem into sqrt(p) separate local matrix-multiplies
- Further we will have to perform sqrt(p) separate communications

P_0 $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0}$

P_1 $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1}$

…

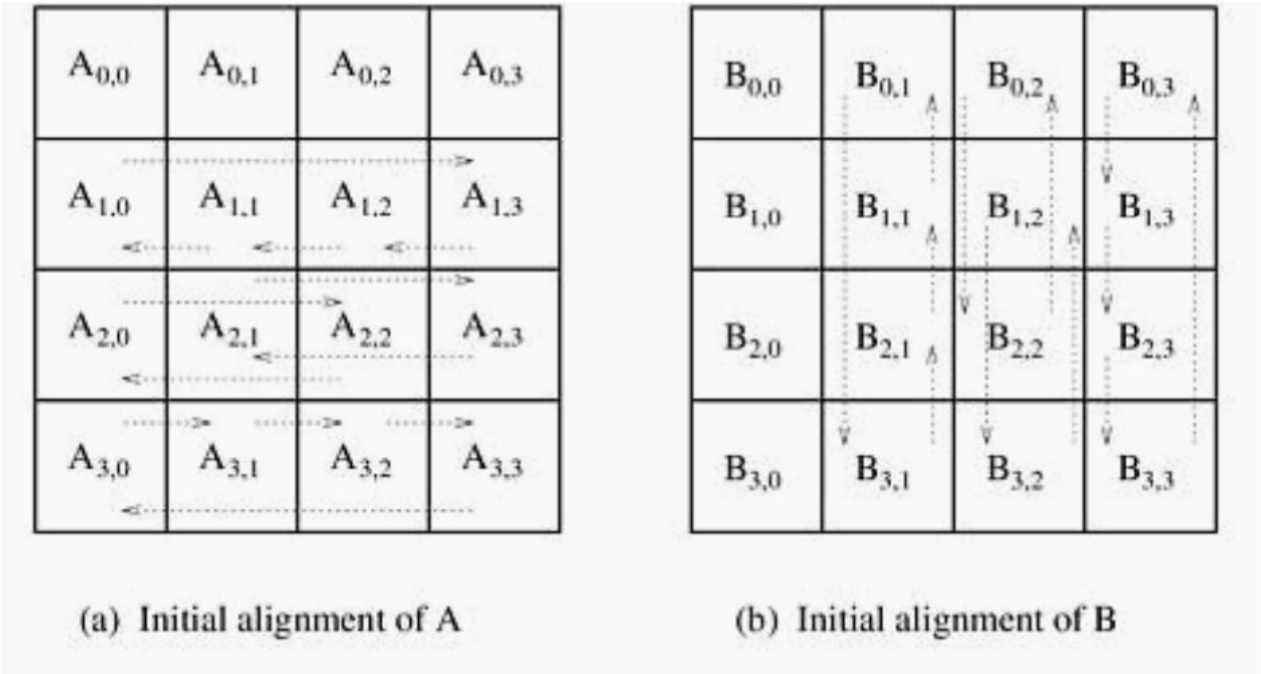
Cannon's Algorithm

- Let's switch to 16 processors & walk through the implementation in detail
- In this case, we have an initial problem → only processes along the diagonal can compute their values
- E.g. P_1 has $A_{0,1}$ and $B_{0,1}$ at the start of the program, but there is no term in $C_{0,1}$ which has $A_{0,1}B_{0,1}$
- As such we have to perform initial alignment

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{0,1}$	$A_{0,2}$ $B_{0,2}$	$A_{0,3}$ $B_{0,3}$
$A_{1,0}$ $B_{1,0}$	$A_{1,1}$ $B_{1,1}$	$A_{2,1}$ $B_{2,1}$	$A_{1,3}$ $B_{1,3}$
$A_{2,0}$ $B_{2,0}$	$A_{2,1}$ $B_{2,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{2,3}$
$A_{3,0}$ $B_{3,0}$	$A_{3,1}$ $B_{3,1}$	$A_{3,2}$ $B_{3,2}$	$A_{3,3}$ $B_{3,3}$

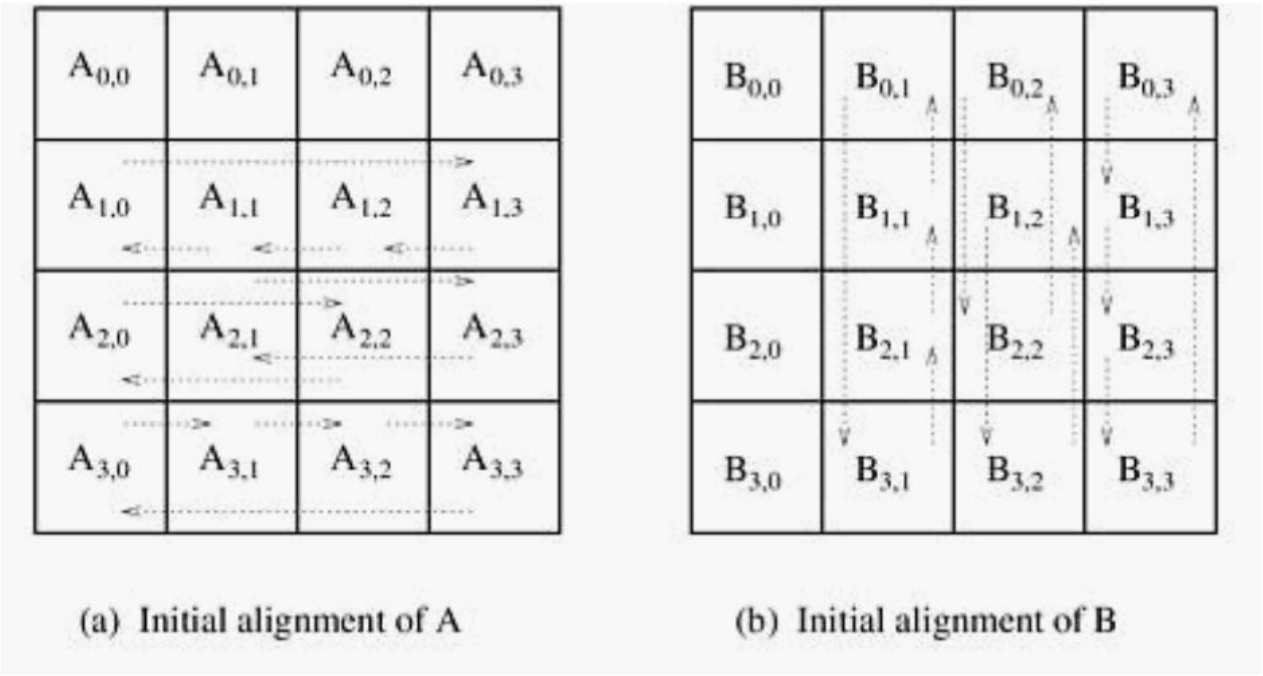
Cannon's Algorithm

Perform the initial alignment so that each block of **A** and **B** may be used for multiplication in parallel on each process



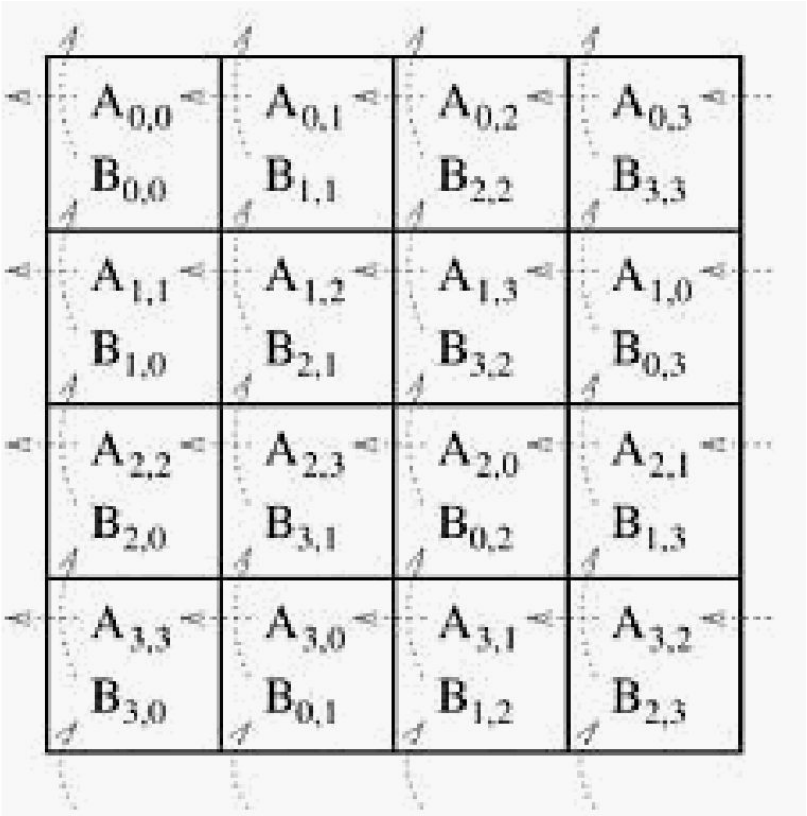
Cannon's Algorithm

Perform the initial alignment so that each block of **A** and **B** may be used for multiplication in parallel on each process



Now we can start multiplying

Cannon's Algorithm



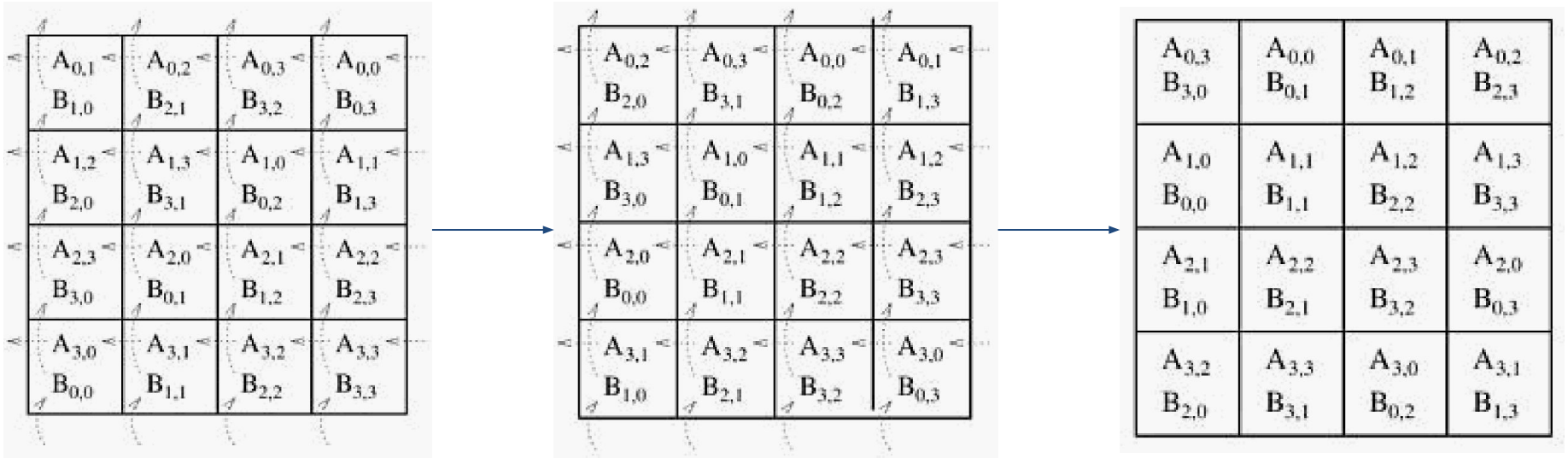
We see two steps represented in this image

(1) Each process first computes using the local blocks it holds in memory (e.g. P_0 computes $A_{0,1}B_{1,0}$)

(2) After performing its computation, each process communicates (dashed lines) it's local portion of **A** leftward, and **B** upward. Note that this communication always occurs between adjacent processes.

Cannon's Algorithm

We repeat those same two steps, but only after the previous communication has complete. This proceeds sequentially.



Cannon's Algorithm

We can therefore define Cannon's algorithm as

(1) *[In parallel]* Perform initial shift of elements of **A** and **B**

(2) *[sequential]* For loop over j=0,sqrt(p)-1

- (a) *[In parallel]* Compute local block-matrix ($A_{i,j}B_{j,k}$) on process $P_{i*sqrt(p) + k}$
- (b) *[In parallel]* Communicate blocks to adjacent processes

After execution, each processor will hold one block of **C**

$C_{0, 0}$	$C_{0, 1}$	$C_{0, 2}$	$C_{0, 0}$
$C_{1, 0}$	$C_{1, 1}$	$C_{1, 2}$	$C_{1, 0}$
$C_{2, 0}$	$C_{1, 0}$	$C_{2, 2}$	$C_{2, 0}$
$C_{0, 0}$	$C_{0, 1}$	$C_{0, 2}$	$C_{0, 0}$

There are two open questions with how we can use MPI to program Cannon's Algorithm

(1) How do we know which process is up/down/left/right?
(2) How can we be sure that processes are actually mapped to be adjacent on hardware?

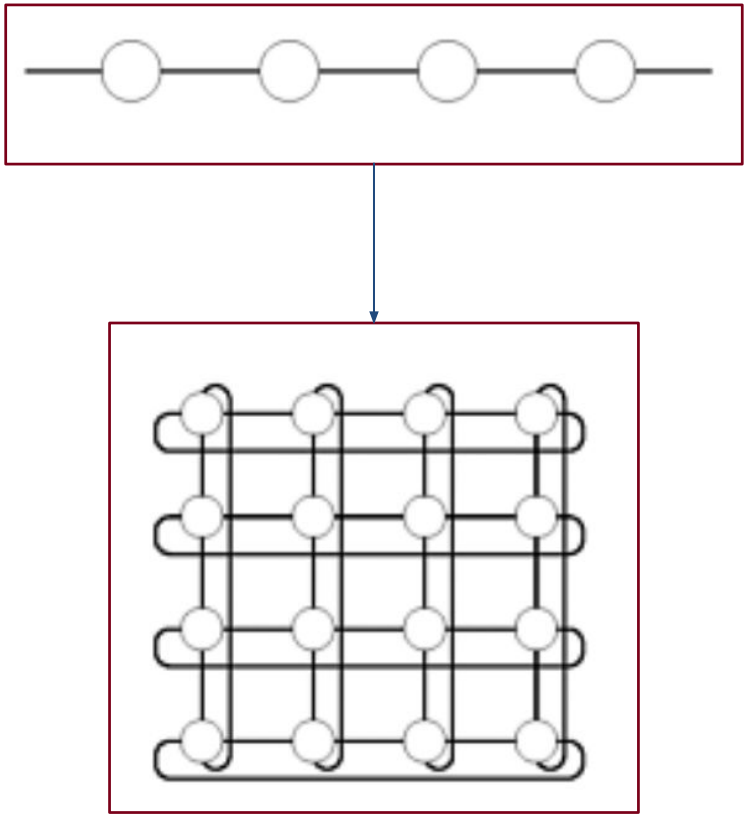
There are two open questions with how we can use MPI to program Cannon's Algorithm

(1) How do we know which process is up/down/left/right?
(2) How can we be sure that processes are actually mapped to be adjacent on hardware?

Use Topologies

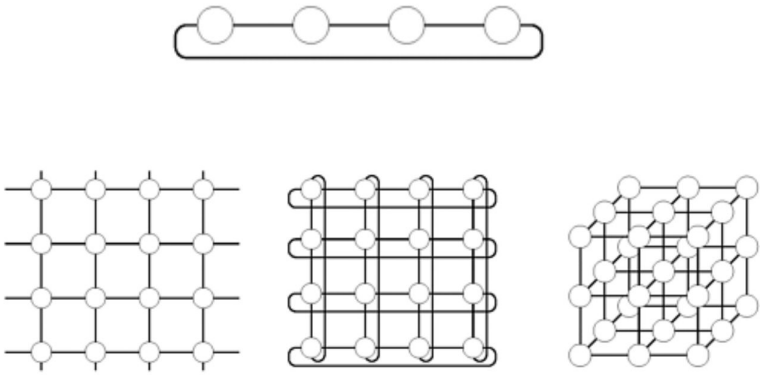
Topologies

- MPI initially views the processes as being arranged in an array by assigning a *rank* to each process, indicating the linear order of processes
- However, many programs, like the Cannon's algorithm we just described, make use of 2-d or 3-d topologies
- MPI exposes a set of routines which allows you to re-organize your processes into a different topology



Cartesian Topologies in MPI

- Cartesian topologies are one, 2, 3 or higher dimensional grids
- It is possible to define other topologies (graphs, distributed graphs, etc.), but we focus only only Cartesian Topologies



Creating a Cartesian Topology

- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

Creating a Cartesian Topology

- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

The old communicator we want to create a cartesian topology from. Often this is ***MPI_COMM_WORLD.***

Creating a Cartesian Topology

- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

The dimensionality of the grid we wish to create. 1 is a ring. 2 is a mesh. 3 is a 3-d grid. Log(p) is a hypercube, where *p* is the number of processes in ***comm_old***

Creating a Cartesian Topology

- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

The size of each of the ***ndims*** dimensions. For example, if ***ndims*** were set to 2, then it is typical to use *{sqrt(p), sqrt(p)}* for this array.

Creating a Cartesian Topology

- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

An array of size ndims indicating whether to include wraparound for each dimension.

Creating a Cartesian Topology

- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

Whether to allow MPI to reorder the processes so that the new grid has adjacent processes close together in hardware. **MAKE SURE** that you set this to 1. There are few cases where not reordering is a good idea.

Creating a Cartesian Topology

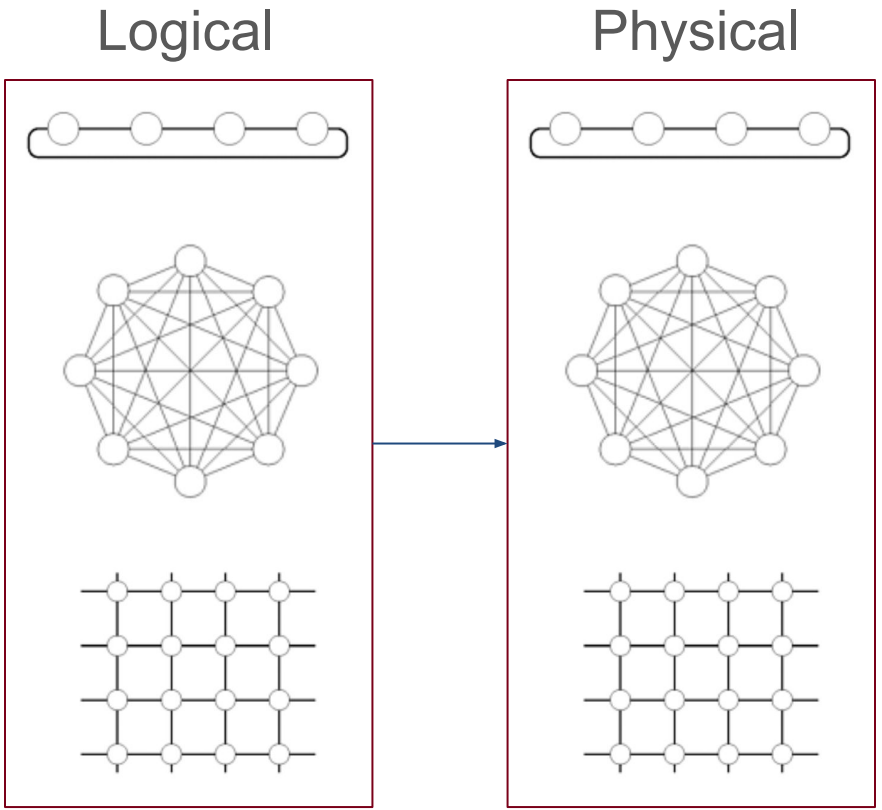
- We can create a Cartesian topology with ***MPI_Cart_create***
- Once called, this function will provide a mapping from the original ring of processes onto the grid which we specify
- Be sure to set reorder to 1 to ensure mapping results in adjacent processes being close together in hardware

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims, int *dims,
        int *periods, int reorder,
        MPI_Comm *comm_cart)
```

The new communicator - which ***MPI_Cart_create*** will save into after execution.

How does MPI choose a Mapping?

- MPI will typically attempt to map logical processes to be close to one another on the physical processors
- The actual implementation can vary quite - but in general heuristic methods are used
- Typically, logical processes which are adjacent to each other can be expected to be mapped close to each other in hardware to minimize contention & overhead



Getting Cartesian Rank Information

- In the new Cartesian topology, each process has an integer rank as well as a location in the n-dimensional grid (an array of integers)
- ***MPI_Cart_rank*** accepts as input the coordinates of a process and returns the rank of that process
- ***MPI_Cart_coord*** accepts as input the rank of a process and returns the coordinates

```
int MPI_Cart_rank(
        MPI_Comm comm_cart,
        int *coords, int *rank)
int MPI_Cart_coord(
        MPI_Comm comm_cart,
        int rank, int maxdims,
        int *coords)
```

Getting Cartesian Rank Information

We'll see some examples shortly

- In the new Cartesian topology, each process has an integer rank as well as a location in the n-dimensional grid (an array of integers)
- ***MPI_Cart_rank*** accepts as input the coordinates of a process and returns the rank of that process
- ***MPI_Cart_coord*** accepts as input the rank of a process and returns the coordinates

```
int MPI_Cart_rank(
        MPI_Comm comm_cart,
        int *coords, int *rank)
int MPI_Cart_coord(
        MPI_Comm comm_cart,
        int rank, int maxdims,
        int *coords)
```

Communication with Cartesian Topologies

- To communicate, we need to use processor ranks
- In particular, we often need to know the ranks of processes (1) above, (2) below, (3) to the left and (4) to the right of us
- ***MPI_Cart_shift*** enables us to find the ranks of these processes for later communications

```
int MPI_Cart_shift(
        MPI_Comm comm_cart,
        int dir, int s_step,
        int *rank_source,
        int *rank_dest)
```

The Cartesian communicator we wish to get process ranks for

Communication with Cartesian Topologies

- To communicate, we need to use processor ranks
- In particular, we often need to know the ranks of processes (1) above, (2) below, (3) to the left and (4) to the right of us
- ***MPI_Cart_shift*** enables us to find the ranks of these processes for later communications

```
int MPI_Cart_shift(
        MPI_Comm comm_cart,
        int dir, int s_step,
        int *rank_source,
        int *rank_dest)
```

Which dimension we want to get ranks for. If we have a two-dimensional communicator, then setting this to 0 means shift along columns. 1 Means shift along rows.

Communication with Cartesian Topologies

- To communicate, we need to use processor ranks
- In particular, we often need to know the ranks of processes (1) above, (2) below, (3) to the left and (4) to the right of us
- ***MPI_Cart_shift*** enables us to find the ranks of these processes for later communications

```
int MPI_Cart_shift(
        MPI_Comm comm_cart,
        int dir, int s_step,
        int *rank_source,
        int *rank_dest)
```

The size of the shift. If ‘1’ or ‘-1’ then the ranks of adjacent processes along the ***dir*** dimension are returned

Communication with Cartesian Topologies

- To communicate, we need to use processor ranks
- In particular, we often need to know the ranks of processes (1) above, (2) below, (3) to the left and (4) to the right of us
- ***MPI_Cart_shift*** enables us to find the ranks of these processes for later communications

```
int MPI_Cart_shift(
        MPI_Comm comm_cart,
        int dir, int s_step,
        int *rank_source,
        int *rank_dest)
```

Based on the shift step & direction, the rank of the process which will be sending data to the calling process

Communication with Cartesian Topologies

- To communicate, we need to use processor ranks
- In particular, we often need to know the ranks of processes (1) above, (2) below, (3) to the left and (4) to the right of us
- ***MPI_Cart_shift*** enables us to find the ranks of these processes for later communications

```
int MPI_Cart_shift(
        MPI_Comm comm_cart,
        int dir, int s_step,
        int *rank_source,
        int *rank_dest)
```

Based on the shift step & direction, the rank of the process which will be receiving data from the calling process

Cannon's Algorithm in MPI

Start by defining a local matrix multiplication operation. For the purposes of this program, we assume that **A** and **B** are *n x n* matrices & that n is divisible by sqrt(*p*)

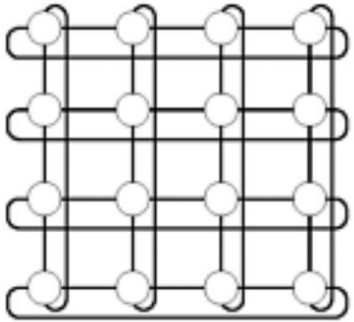
```
/* This function performs a serial matrix-matrix multiplication c = a*b */
MatrixMultiply(int n, double *a, double *b, double *c)
{
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      for (k=0; k<n; k++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```

Cannon's Algorithm in MPI

```
MatrixMatrixMultiply(int n, double *a, double *b, double *c,
                     MPI_Comm comm)
{
  int i;
  int nlocal;
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Cannon's Algorithm in MPI

2-d mesh with wraparound

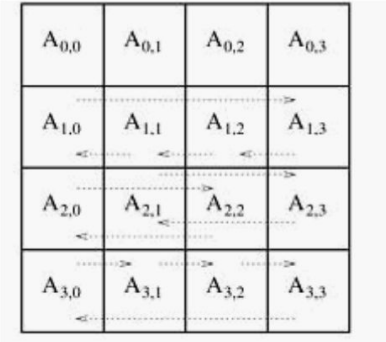


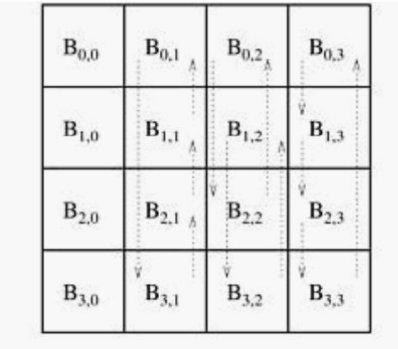
```
MatrixMatrixMultiply(int n, double *a, double *b, double *c,
                     MPI_Comm comm)
{
  int i;
  int nlocal;
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Cannon's Algorithm in MPI

```
/* Get the rank and coordinates with respect to the new topology */
 MPI_Comm_rank(comm_2d, &my2drank);
 MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Compute ranks of the up and left shifts */
 MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
 MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

/* Determine the dimension of the local matrix block */
nlocal = n/dims[0];

/* Perform the initial matrix alignment. First for A and then for B */
 MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest,
     1, shiftsource, 1, comm_2d, &status);

 MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

Cannon's Algorithm in MPI



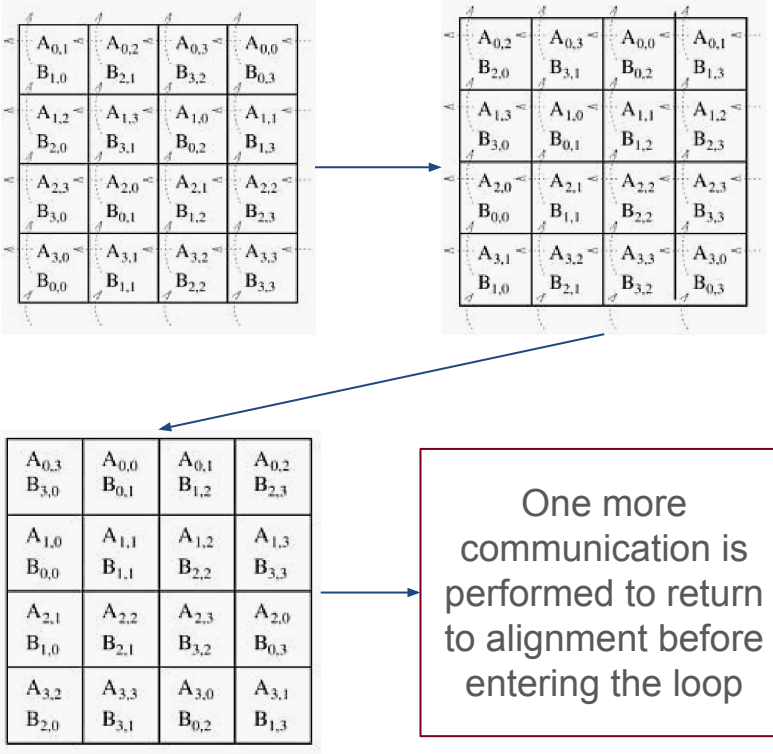



```
/* Get the rank and coordinates with respect to the new topology */
 MPI_Comm_rank(comm_2d, &my2drank);
 MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Compute ranks of the up and left shifts */
 MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
 MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

/* Determine the dimension of the local matrix block */
nlocal = n/dims[0];

/* Perform the initial matrix alignment. First for A and then for B */
 MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest,
     1, shiftsource, 1, comm_2d, &status);

 MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

Cannon's Algorithm in MPI

```
  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
        leftrank, 1, rightrank, 1, comm_2d, &status);

    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
        uprank, 1, downrank, 1, comm_2d, &status);
  }
  MPI_Comm_free(&comm_2d); /* Free up communicator */
}
```

Cannon's Algorithm in MPI

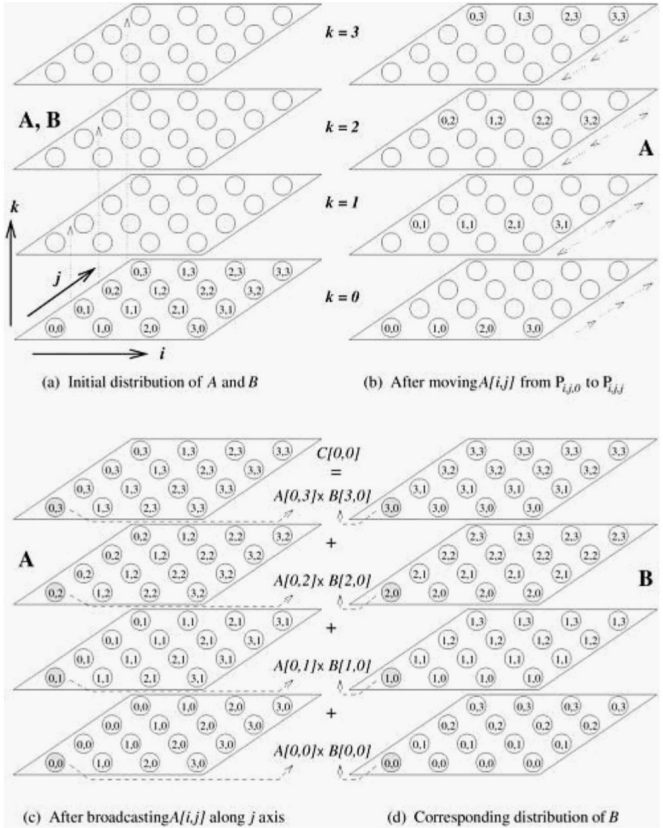


One more communication is performed to return to alignment before entering the loop

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

  /* Shift matrix a left by one */
   MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
       leftrank, 1, rightrank, 1, comm_2d, &status);

  /* Shift matrix b up by one */
   MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
       uprank, 1, downrank, 1, comm_2d, &status);
}
  MPI_Comm_free(&comm_2d); /* Free up communicator */
}
```

Other Means of Matrix Multiplication

- There are many other ways of performing matrix multiplication in parallel settings
- We will explore some of these in greater detail in the coming days in MPI and in the coming weeks in CUDA



LECTURE ENDED HERE. THE REMAINDER OF THE LECTURE WILL BE COVERED ON 10/13.

What is a downside of the current project to our implementation of Cannon's Algorithm?

What is a downside of the current project to our implementation of Cannon's Algorithm?

The CPU will idle during communication steps.

What is a downside of the current project to our implementation of Cannon's Algorithm?

The CPU will idle during communication steps.

We can resolve this by overlapping communication + computation with non-blocking MPI calls.

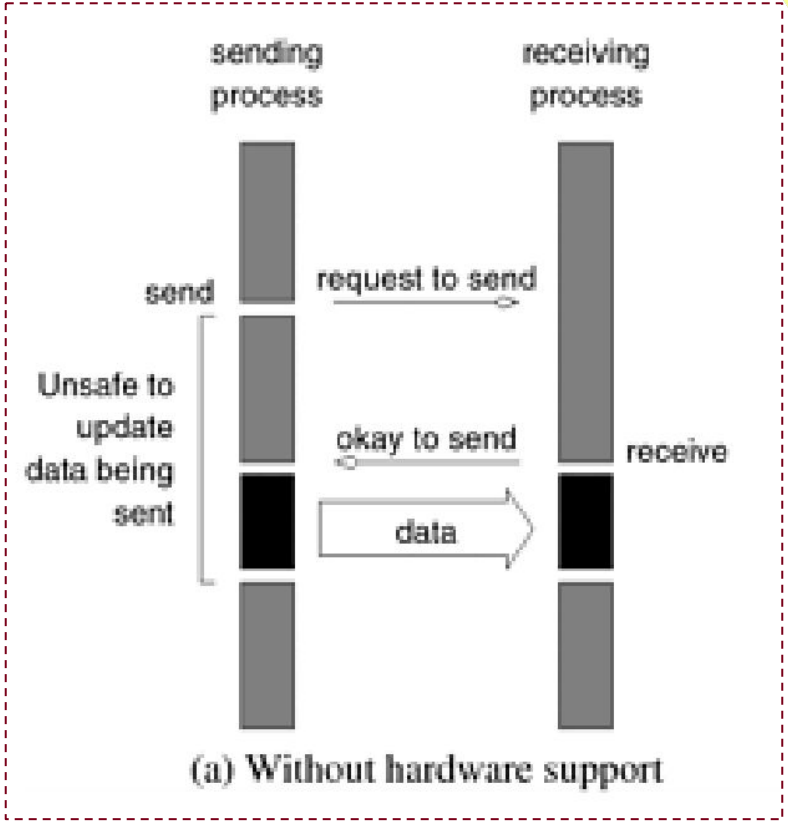
Lecture Overview

- Running an MPI Program in Practice
- Odd-Even Sort Example
- Topologies
- **Non-Blocking MPI Communications**

Non-Blocking MPI Sends & Receives

Recap

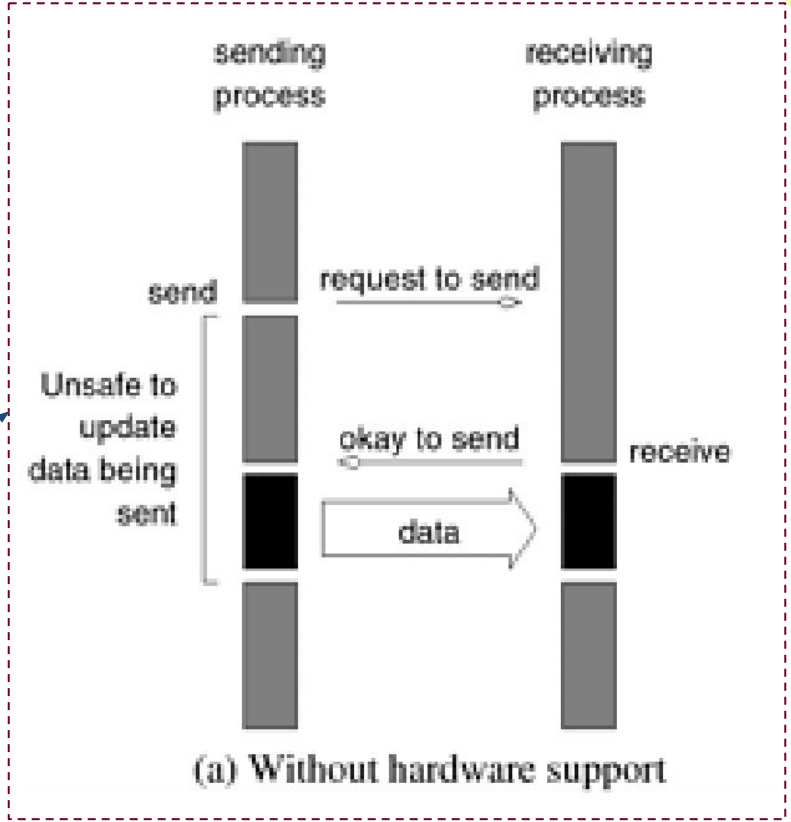
- A program using non-blocking Sends & Receives will ***immediately*** continue after finishing the Send or Receive function
- We ***do not*** have any guarantees that our program will not overwrite the data being sent
- This is dependent on whether or not the data will be buffered, and how that buffering occurs



Non-Blocking MPI Sends & Receives

Recap

- A program using non-blocking Sends & Receives will ***immediately*** continue after finishing the Send or Receive function
- We ***do not*** have any guarantees that our program will not overwrite the data being sent
- This is dependent on whether or not the data will be buffered, and how that buffering occurs

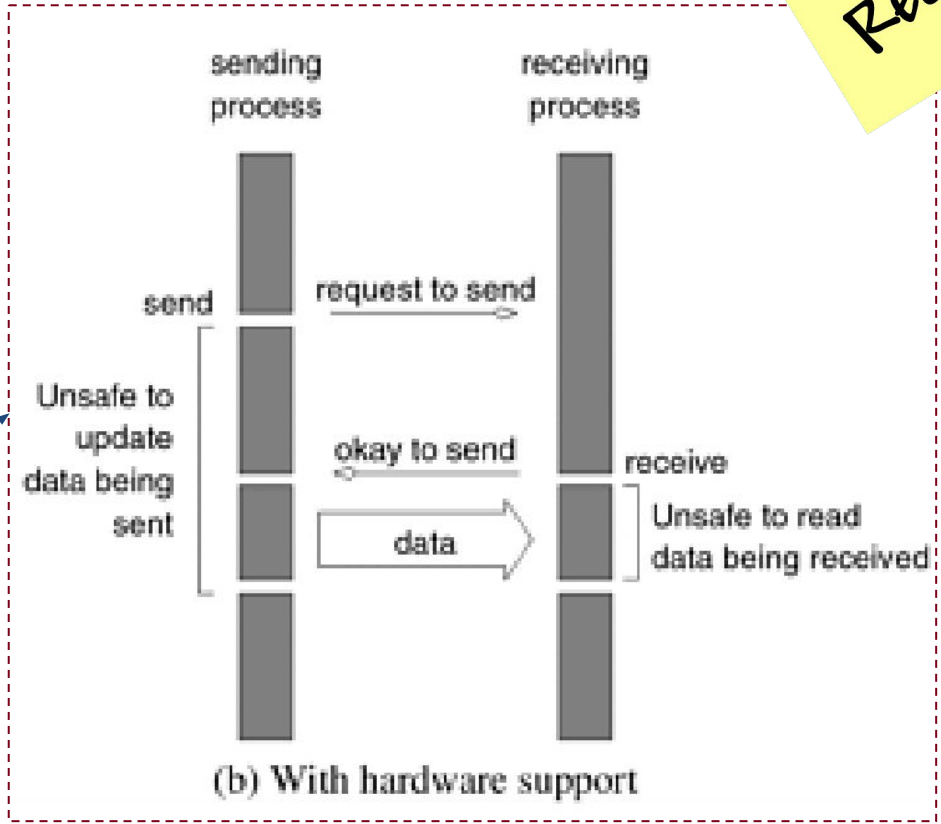


Non-Buffered

Non-Blocking MPI Sends & Receives

Recap

- A program using non-blocking Sends & Receives will ***immediately*** continue after finishing the Send or Receive function
- We ***do not*** have any guarantees that our program will not overwrite the data being sent
- This is dependent on whether or not the data will be buffered, and how that buffering occurs



Non-Buffered

Non-Blocking MPI Sends & Receives

- ***MPI_Isend*** & ***MPI_Irecv*** enable non-blocking communication with MPI
- With hardware support, these operations enable us to perfectly overlap communication and computation
- Without hardware support, we remove any idling, but the CPU must still participate in the network communication
- MPI_Isend may be buffered, MPI_Irecv is not
- As such, we must be careful when structuring program execution to not overwrite the buffers

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

Non-Blocking MPI Sends & Receives

- ***MPI_Isend*** & ***MPI_Irecv*** enable non-blocking communication with MPI
- With hardware support, these operations enable us to perfectly overlap communication and computation
- Without hardware support, we remove any idling, but the CPU must still participate in the network communication
- MPI_Isend may be buffered, MPI_Irecv is not
- As such, we must be careful when structuring program execution to not overwrite the buffers

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

Same arguments as used in MPI_Send & MPI_Recv

Non-Blocking MPI Sends & Receives

- ***MPI_Isend*** & ***MPI_Irecv*** enable non-blocking communication with MPI
- With hardware support, these operations enable us to perfectly overlap communication and computation
- Without hardware support, we remove any idling, but the CPU must still participate in the network communication
- MPI_Isend may be buffered, MPI_Irecv is not
- As such, we must be careful when structuring program execution to not overwrite the buffers

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

The ***request*** object allows us to monitor for when the corresponding send or receive has completed (we will explore this more in upcoming slides)

Non-Blocking MPI Sends & Receives

```
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```


```
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
   MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
}
...
```

Non-Blocking MPI Sends & Receives

- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

Non-Blocking MPI Sends & Receives

- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

The ***request*** object - returned from the earlier ***MPI_ISend*** or ***MPI_Irecv*** calls

Non-Blocking MPI Sends & Receives

- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

A variable indicating whether or not the given communication operation has completed. This returns ‘1’ if it has, ‘0’ otherwise.

Non-Blocking MPI Sends & Receives

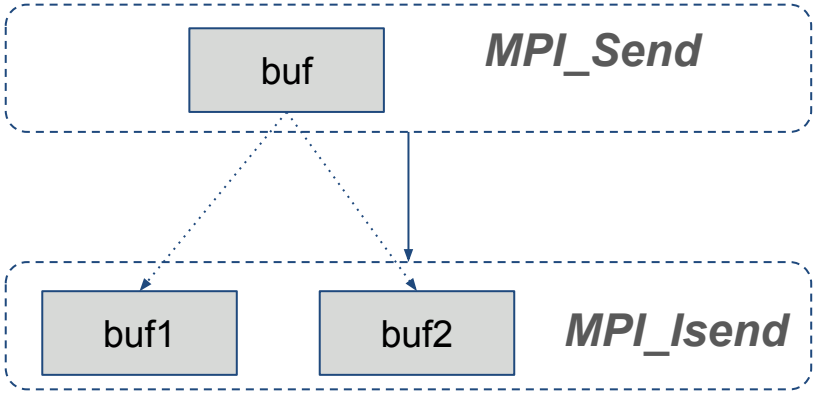
- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

The ***status*** of the communication operation. This is the same object returned by ***MPI_Revc***.

Cannon's Algorithm in MPI

- We make the following changes to ensure that our MPI program works with non-blocking operations
  - Use ***MPI_Isend, MPI_Irecv, MPI_Wait***
  - Duplicate the local buffers ***a*** and ***b*** on each process
- In general, you will typically want to explicitly use duplicate buffers to ensure that the buffers you are writing to during computation are different from those you are currently using to communicate

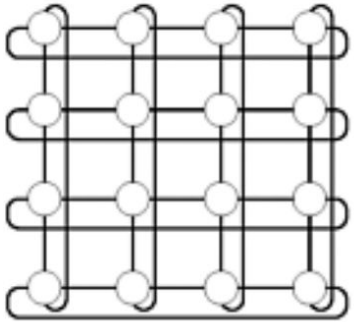


Non-Blocking Cannon's Algorithm in MPI

```
MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
                                 double *c, MPI_Comm comm)
{
  int i, j, nlocal;
  double *a_buffers[2], *b_buffers[2];
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;
  MPI_Request reqs[4];

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Non-Blocking Cannon's Algorithm in MPI

2-d mesh with wraparound

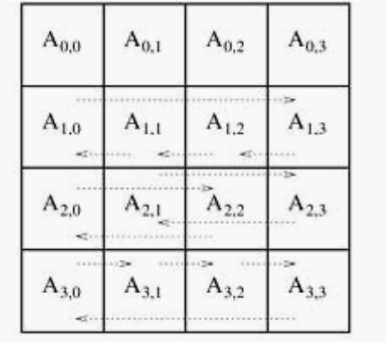


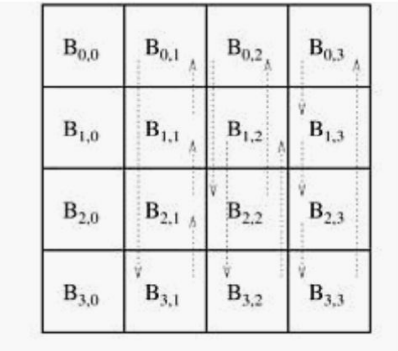
```
MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
                                 double *c, MPI_Comm comm)
{
  int i, j, nlocal;
  double *a_buffers[2], *b_buffers[2];
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;
  MPI_Request reqs[4];

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Non-Blocking Cannon's Algorithm in MPI

```
/* Get the rank and coordinates with respect to the new topology */
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Compute ranks of the up and left shifts */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

/* Determine the dimension of the local matrix block */
nlocal = n/dims[0];

/* Setup the a_buffers and b_buffers arrays */
a_buffers[0] = a;
a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
b_buffers[0] = b;
b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));

/* Perform the initial matrix alignment. First for A and then for B */
MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

Non-Blocking Cannon's Algorithm in MPI



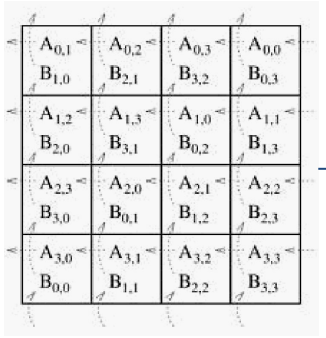


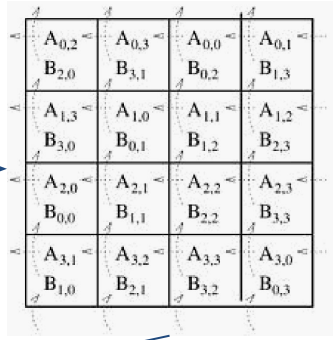

```
/* Get the rank and coordinates with respect to the new topology */
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Compute ranks of the up and left shifts */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

/* Determine the dimension of the local matrix block */
nlocal = n/dims[0];

/* Setup the a_buffers and b_buffers arrays */
a_buffers[0] = a;
a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
b_buffers[0] = b;
b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));

/* Perform the initial matrix alignment. First for A and then for B */
MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

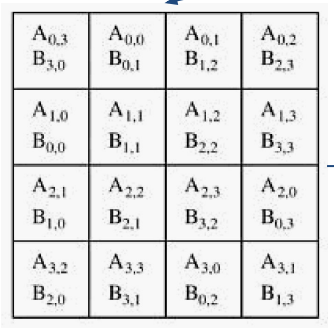
Non-Blocking Cannon's Algorithm in MPI

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
   MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
       leftrank, 1, comm_2d, &reqs[0]);
   MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
       uprank, 1, comm_2d, &reqs[1]);
   MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
       rightrank, 1, comm_2d, &reqs[2]);
   MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
       downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
   MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
 MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);

 MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);

 MPI_Comm_free(&comm_2d); /* Free up communicator */

 free(a_buffers[1]);
```

Non-Blocking Cannon's Algorithm in MPI



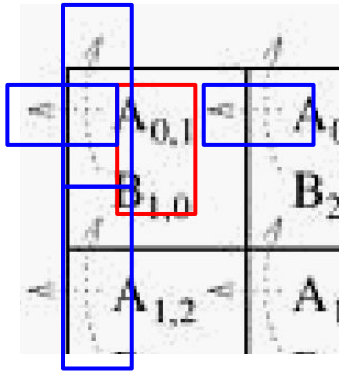




One more communication is performed to return to alignment before entering the loop

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Comm_free(&comm_2d); /* Free up communicator */

free(a_buffers[1]);
```

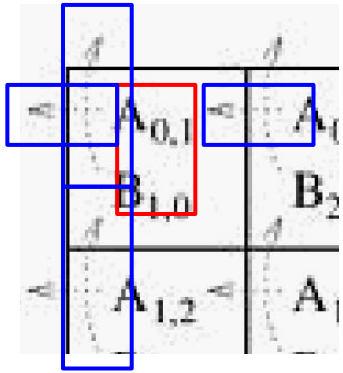
Non-Blocking Cannon's Algorithm in MPI



On each process, computations (red) + communications (blue) are done at the same time.

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Comm_free(&comm_2d); /* Free up communicator */

free(a_buffers[1]);
```

Non-Blocking Cannon's Algorithm in MPI



If there is no dedicated hardware, then the program will *poll* at regular intervals to see if it may communicate & the CPU will pause computation to carry out the necessary communication

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Comm_free(&comm_2d); /* Free up communicator */

free(a_buffers[1]);
```