CPU Parallel Computing Using MPI

Mary Thomas

02/26/2021

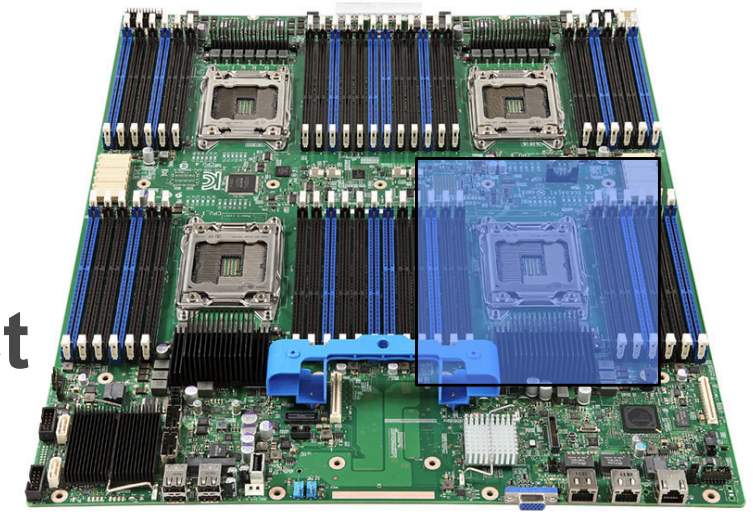
(mpthomas@ucsd.edu)

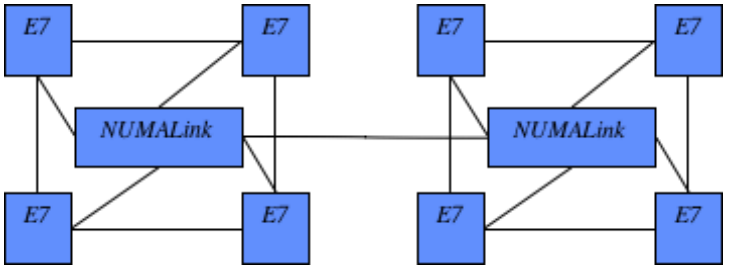
Outline

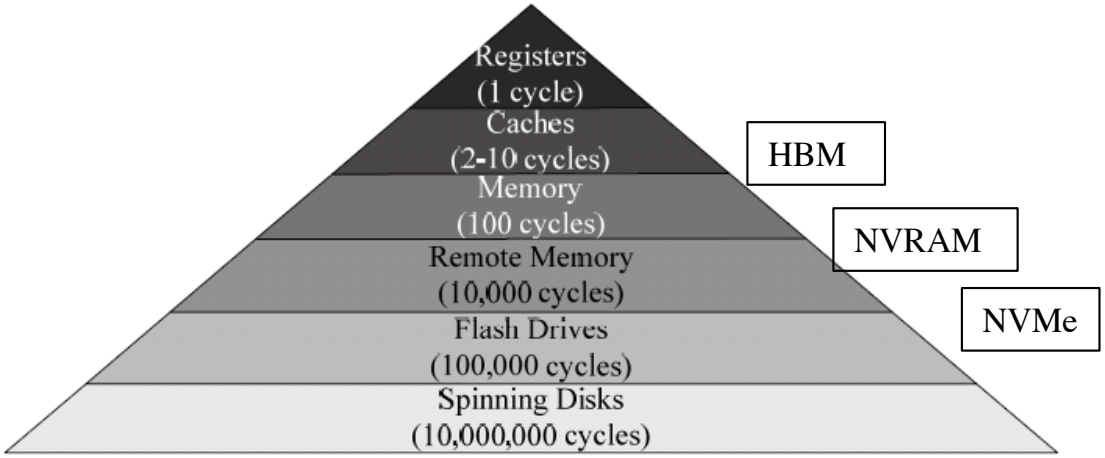
- Overview
- MPI Introduction
- Point-to-Point Communication
- MPI Collectives
- Decomposition and Mapping
- MPI Profiling & Tracing Tools
- More Complex Routines

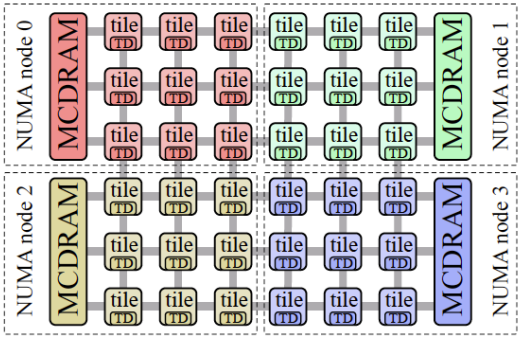
Current Supercomputer Architectures

- **Multi-socket server nodes**
  - NUMA
  - Accelerators
- **High performance interconnect**
  - e.g. Infiniband, OmniPath
- ***Scalable parallel approach needed to achieve performance***



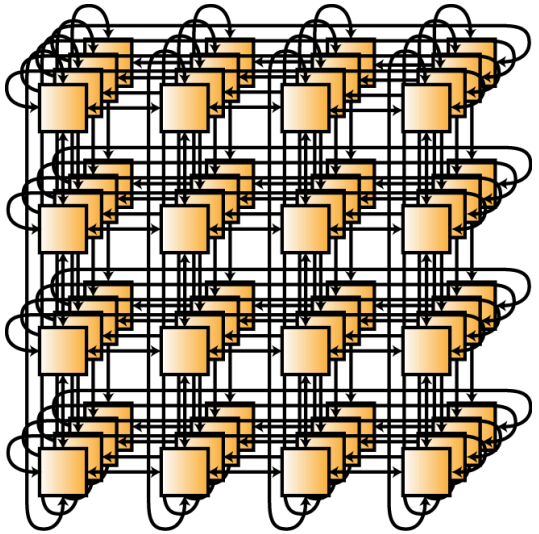


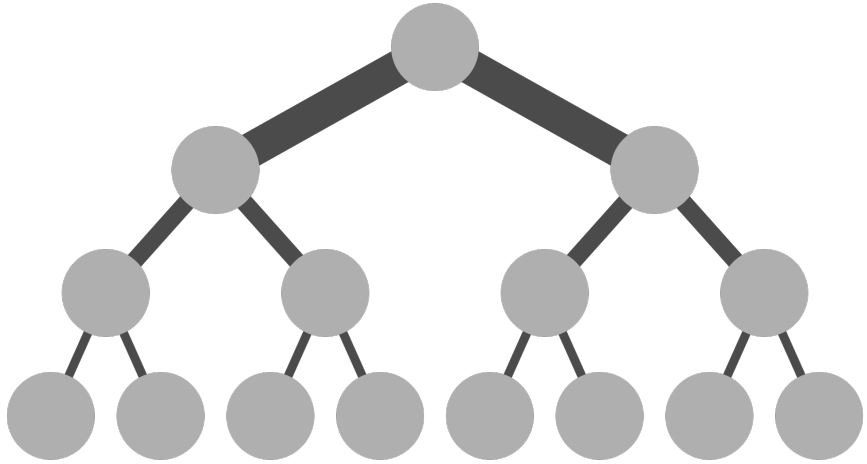




Network Topologies

- **Mesh, Torus, Hypercube**
- **Tree based**
  - Fat-tree
  - Clos
- **Dragonfly**
- **Metrics**
  - Bandwidth
  - Diameter, Connectivity
  - Bisection bandwidth



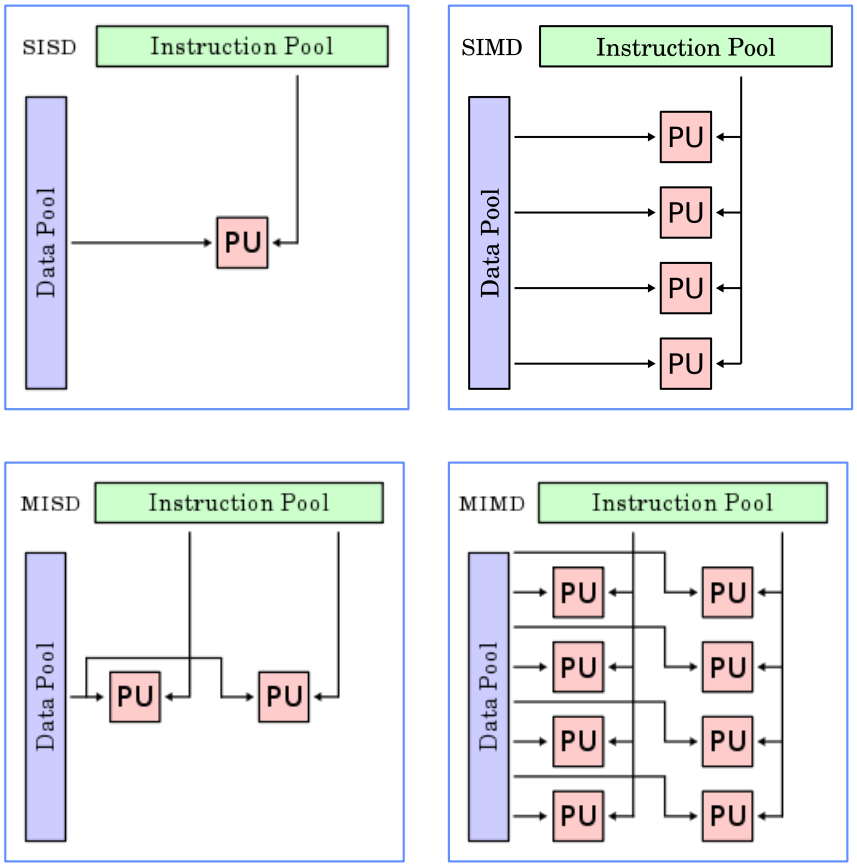


Parallel Computing

- **Executing instructions concurrently on physical resources (not time slicing)**
  - Multiple tightly coupled resources (e.g. cores) collaboratively solving a single problem
- **Benefits**
  - Capacity
    - Memory, storage
  - Performance
    - More instructions per unit of time (FLOPS)
    - Data streaming capability
- **Cost and Complexity**
  - Coordinate tasks and resources
  - Use resources efficiently

Classification of Computer Architectures: Flynn's Taxonomy

- **Single Instruction, Single Data (SISD)**
  - Serial codes
- **Single Instruction, Multiple Data (SIMD)**
  - Processors run the same instructions, each operates on different data
  - Technically, Hadoop MapReduce fit this mode
  - GPUs
- **Multiple Instruction, Single Data (MISD)**
  - Multiple instructions acting on single data stream. E.g. Different analysis on same set of data.
- **Multiple Instruction, Multiple Data (MIMD)**
  - Every processor may execute different instructions
  - Every processor may work on different parts of data
  - Execution can be synchronous or asynchronous, deterministic or non-deterministic
  - Since 2006, all top 10 and most of TOP500 systems are MIMD (Src: Wikipedia)



Memory, Communication, and Execution Models

- **Shared**
  - Communication model: shared memory
- **Distributed**
  - Communication model: exchange messages
- **Execution Models**
  - Fork-Join (e.g. Thread Level Parallelism)
  - Single Program Multiple Data (SPMD)
- **Parallelism enabled by decomposing work**
  - Tasks can be executed concurrently
  - Some tasks can have dependencies

HPC User Training Examples

- **HPC User Training GitHub examples, Week6:**
  - Clone the repo or do a “git pull” to update the repo.
  - https://github.com/sdsc-hpc-training-org/hpc-training-2021

- **Additional examples @ LLNL:**
  - https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_pi_send.c

Message Passing Interface (MPI)

- **Low level message passing abstraction**
  - SPMD execution model + messages
  - Designed for distributed memory. Implemented on hybrid distributed memory/shared memory systems.
- **MPI: API specification**
  - Portable: de-facto standard for parallel computing, portable, system specific optimizations without changing code interface
    - https://www.mpich.org/static/docs/latest/
  - http://www.mpi-forum.org
  - Implementations: MVAPICH2, Intel MPI, OpenMPI
  - High performance implementations available virtually on any interconnect and system
  - Point-to-point communication, datatypes, collective operations
  - One-sided communication, Parallel file I/O, Tool support, …

Typical MPI Code Structure

MPI Include File

Variable declarations, etc

Begin Program

…

Serial code

….

MPI Initialization

Parallel Code begins

MPI Rank (process identification)

…

Parallel code based on rank

…

MPI Communications between processes

…

Parallel code based on rank

…

MPI Communications between processes

MPI Finalize (terminate)

Parallel Code ends

Serial Code

Simple MPI Program – Compute PI

- **Initialize MPI (MPI_Init function)**
- **Find the number of tasks and taskids (MPI_Comm_size, MPI_Comm_rank)**
- **PI is calculated using an integral. The number of intervals used for the integration is fixed at 128000.**
- **Computes the sums for a different sections of the intervals in each MPI task.**
- **At the end of the code, the sums from all the tasks are added together to evaluate the final integral. This is accomplished through a reduction operation (MPI_Reduce function).**
- **Simple code illustrates decomposition of problem into parallel components.**

MPI Data Types

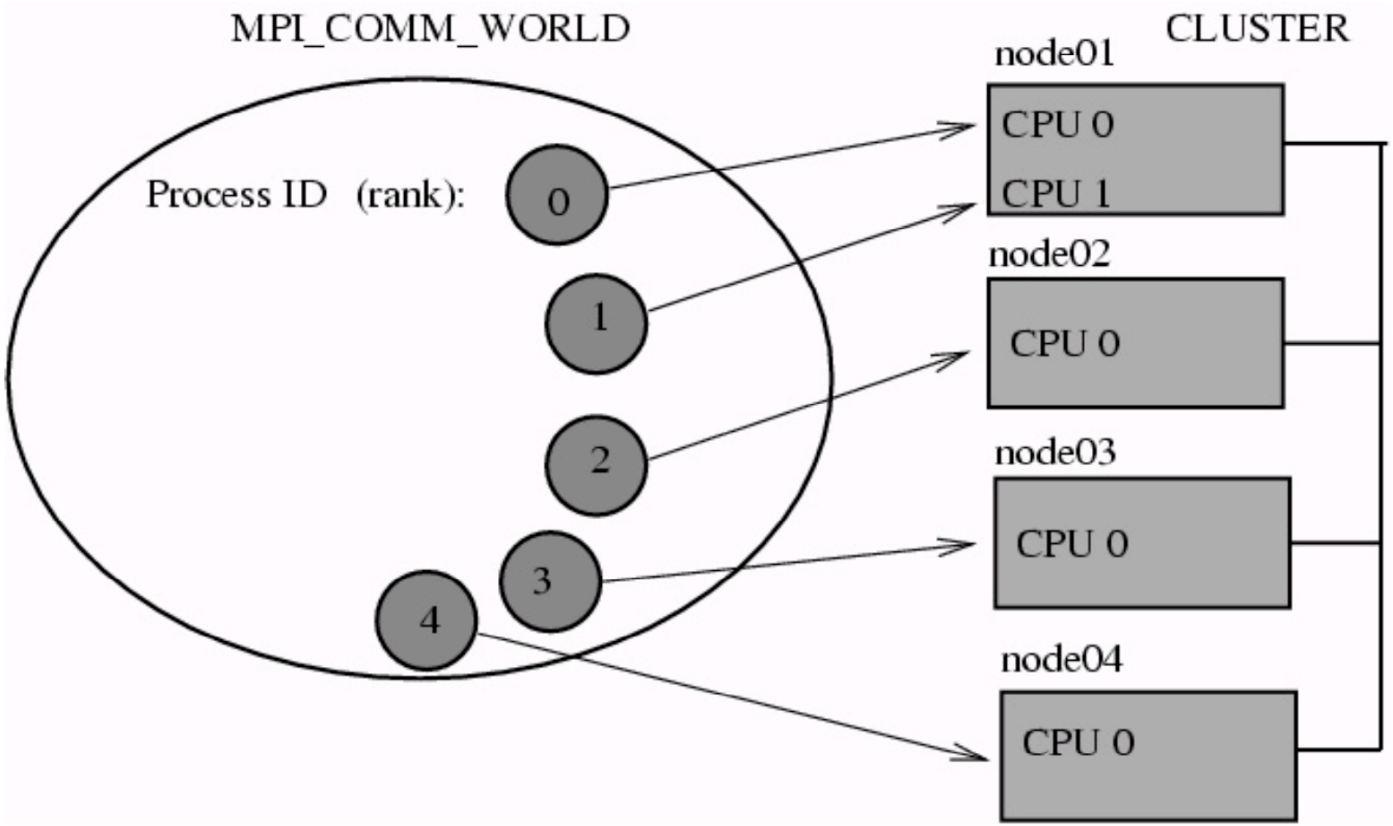
C Data Types		FORTRAN Data Types
MPI_CHAR	MPI_C_DOUBLE_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_BOOL	MPI_INTEGER1
MPI_INT	MPI_LOGICAL	MPI_INTEGER2
MPI_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_INT8_T	MPI_REAL
MPI_LONG_LONG	MPI_INT16_T	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT32_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT64_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_UINT8_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_UINT16_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT32_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT64_T	MPI_LOGICAL
MPI_DOUBLE	MPI_BYTE	MPI_BYTE
MPI_LONG_DOUBLE	MPI_PACKED	MPI_PACKED
MPI_C_COMPLEX		
MPI_C_FLOAT_COMPLEX		

MPI Program to Compute PI

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
  {
  int numprocs, rank, len;
  int i, iglob, INTERVALS, INTLOC;
  double n_1, x, pi, piloc;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,
    &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(hostname,
    &len);

  INTERVALS=128000;
  printf("Hello from MPI task= %d\n",
    rank);
  MPI_Barrier(MPI_COMM_WORLD);
```

```
 if (rank == 0) { printf("Number of MPI
     tasks = %d\n", numprocs);
 }

 INTLOC=INTERVALS/numprocs;
 piloc=0.0;
 n_1=1.0/(double)INTERVALS;
 for (i = 0; i < INTLOC; i++) {
   iglob = INTLOC*rank+i;
   x = n_1 * ((double)iglob  - 0.5);
   piloc += 4.0 / (1.0 + x * x);
 }

 MPI_Reduce(&piloc,&pi,1,MPI_DOUBLE,
      MPI_SUM,0,MPI_COMM_WORLD);
 if (rank == 0){
   pi *= n_1;
   printf ("Pi = %.12lf\n", pi);
 }

 MPI_Finalize();
 }
```

Message Passing Interface (MPI)



Communication on a multimode multicore cluster

PI Code : MPI Environment Functions

MPI_Init(&argc, &argv);

Initializes MPI, *must* be called (only once) in every MPI program before any MPI functions.

MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

Returns the total number of tasks in the communicator. MPI uses communicators to define which collections of processes can communicate with each other. The default MPI_COMM_WORLD includes all the processes. User defined communicators are an option.

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

Returns the rank (ID) of the calling MPI process within the communicator.

MPI_Finalize();

Ends the MPI execution environment. No MPI calls after this.!

Other routines in code are collectives and we will discuss them later in the talk.

PI Code : Other MPI Functions/Feartures

MPI_Get_processor_name(char *name, int *resultlen);

Gets the name of the processor

```
MPI_Get_processor_name(hostname, &len);
```

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

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

Compiling and Running PI Example

Compile:

```
[mthomas@login01 examples]$ ls -al
total 72
drwxr-xr-x 2 mthomas use300     7 Feb 25 22:28 .
drwxr-xr-x 3 mthomas use300     4 Feb 25 21:34 ..
-rwxr-xr-x 1 mthomas use300 20960 Feb 25 22:20 pi_mpi
-rw-r--r-- 1 mthomas use300   947 Feb 25 22:21 pi_mpi.c
-rw-r--r-- 1 mthomas use300   489 Feb 25 22:23 pi-mpi.sb
-rw-r--r-- 1 mthomas use300   229 Feb 25 22:20 README.txt

[mthomas@login01 examples]$ mpicc -o pi_mpi pi_mpi.c
[mthomas@login01 examples]$ sbatch pi-mpi.sb
Submitted batch job 1367377
[mthomas@login01 examples]$ !sq
squeue -u mthomas
JOBID    PARTITION NAME    USER     ST  . . .
1367380  compute   pi-mpi  mthomas  R . . .
[mthomas@login01 examples]$ ls -al
total 72
drwxr-xr-x 2 mthomas use300     7 Feb 25 22:28 .
drwxr-xr-x 3 mthomas use300     4 Feb 25 21:34 ..
-rwxr-xr-x 1 mthomas use300 20960 Feb 25 22:20 pi_mpi
-rw-r--r-- 1 mthomas use300   947 Feb 25 22:21 pi_mpi.c
-rw-r--r-- 1 mthomas use300   489 Feb 25 22:23 pi-mpi.sb
-rw-r--r-- 1 mthomas use300   229 Feb 25 22:20 README.txt
-rw-r--r-- 1 mthomas use300   575 Feb 25 22:32 pi-mpi.1367380.exp-2-10.out
```

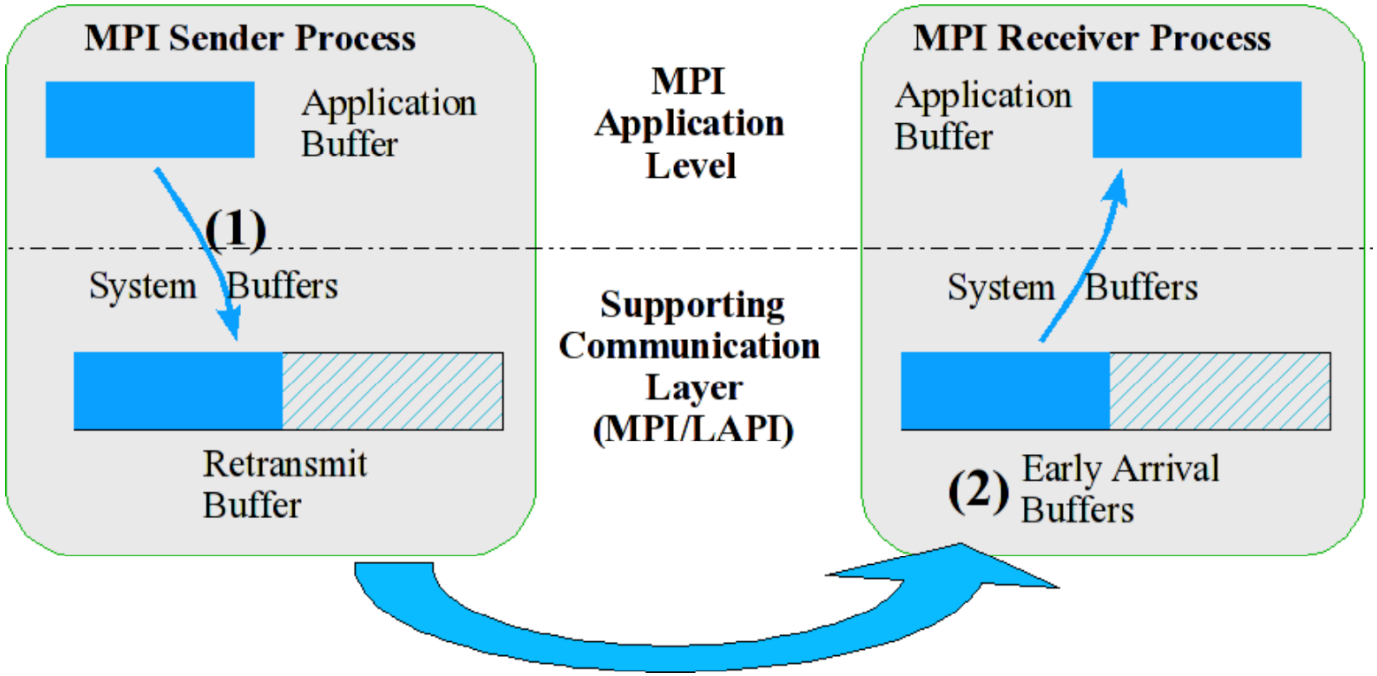
Sample Output:

```
[mthomas@login01 examples]$ cat pi-
mpi.1367374.exp-2-10.out
Hello from MPI task= 0 on exp-2-10
Hello from MPI task= 3 on exp-2-10
Hello from MPI task= 8 on exp-2-11
Hello from MPI task= 1 on exp-2-10
Hello from MPI task= 2 on exp-2-10
Hello from MPI task= 10 on exp-2-12
Hello from MPI task= 4 on exp-2-10
Hello from MPI task= 6 on exp-2-11
Hello from MPI task= 9 on exp-2-11
Hello from MPI task= 12 on exp-2-12
Hello from MPI task= 13 on exp-2-12
Hello from MPI task= 11 on exp-2-12
Hello from MPI task= 14 on exp-2-12
Hello from MPI task= 5 on exp-2-11
Hello from MPI task= 7 on exp-2-11
Number of MPI tasks = 15
Pi = 3.141575075349
```

Point to Point Communication

- Passing data between two, and only two different MPI tasks.
- Typically one task performs a send operation and the other task performs a matching receive.
- MPI Send operations have choices with different synchronization (when does a send complete) and different buffering (where the data resides till it is received) modes.
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides routines to probe status of messages, and “wait” routines.

Buffers



- Buffer space is used for data in transit – whether its waiting for a receive to be ready or if there are multiple sends arriving at the same receiving tasks.
- Typically a system buffer area managed by the MPI library (opaque to the user) is used. Can exist on both sending & receiving side.
- MPI also provides for user managed send buffer.

Blocking MPI Send, Receive Routines

- **int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
  - Blocking send call will return once it is safe for the application buffer (send data) to be reused.
  - This can happen as soon as the data is copied into the system (MPI) buffer on receiving process.
  - Synchronous if there is confirmation of safe send, and asynchronous otherwise.
- **int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**
  - Blocking receive returns once the data is in the application buffer (receive data) and can by used by the application.

Blocking Send, Recv (Code Snippet)

```
if(myid == 0) {
      for(i = 0; i < 10; i++) {
         s_buf[i] = i*4.0;
      }
      MPI_Send(s_buf, size, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
   }
   else if(myid == 1) {
      MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag, MPI_COMM_WORLD,
&reqstat);
      for (i = 0; i < 10; i++ ){
        printf("r_buf[%d] = %f\n", i, r_buf[i] );
      }
   }
```

MPI Send-Recv Communication Example

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

const int MAX_STRING = 100;
int main(void) {
   char  greeting[MAX_STRING];
   int     comm_sz,  my_rank;
   int     q;
   /* Start up MPI */
   MPI_Init(NULL, NULL);

   /* Get the number of processes */
   MPI_Comm_size(MPI_COMM_WORLD,
         &comm_sz);

   /* Get my rank */
   MPI_Comm_rank(MPI_COMM_WORLD,
       &my_rank);
```

```
   if (my_rank != 0) {
      /* I am a worker node - create a message */
      sprintf(greeting, "Greetings from prc %d of %d!",
            my_rank, comm_sz);
      /* Send message to process 0 */
      MPI_Send(greeting, strlen(greeting)+1,
            MPI_CHAR, 0, 0, MPI_COMM_WORLD);
} else {
      /* Print my message */
      printf("Greetings from Primary process %d of %d!\n",
                  my_rank, comm_sz);
      for (q = 1; q < comm_sz; q++) {
         /* Receive message from process q */
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         /* Print message from process q */
         printf("%s\n", greeting);
      }
   }


   MPI_Finalize();   /* Shut down MPI */

   return 0; }  /* main */
```

Blocking Send, Recv Example

Batch Script:

```
#!/bin/bash
#SBATCH --job-name="mpi-send-recv"
#SBATCH --output="mpi-send-recv.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=4
#SBATCH --export=ALL
#SBATCH -t 00:04:00
#SBATCH -A sds173


## Environment
## MODULE ENV: updated 01/28/2020 (MPT)
 module purge
 module load slurm
 module load cpu
 module load gcc/10.2.0
 module load openmpi/4.0.4


## This job can run on up to 3 nodes,
## and a max of 3*4=12  cores


## Use mpirun to run the job
mpirun -np 6  ./mpi-send-recv
```

Compile:

```
[mthomas@login02 point-to-point-
comm]$ mpicc -o mpi-send-recv mpi-
send-recv.c
```

Submit Job:

```
[mthomas@login02 point-to-point-
comm]$ sbatch mpi-send-recv.sb
Submitted batch job 1367654
```

Output:

```
[mthomas@login01 examples]$ cat pi-
mpi.1367374.exp-2-10.out

Greetings from process 0 of 6!
Greetings from process 1 of 6!
Greetings from process 2 of 6!
Greetings from process 3 of 6!
Greetings from process 4 of 6!
Greetings from process 5 of 6!
```

Deadlocking MPI Tasks

- Take care to properly sequence blocking send/recvs.
  - Possible to deadlock processes waiting on each other.
  - Can also occur with control errors and unexpected symantics
- Sample code snippet:

```
if(myid == 0) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
  }
  else if(myid == 1) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    for (i = 0; i < 10; i++ ){
      printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
```

- MPI_Ssend: blocking synchronous send
- The MPI_Ssend on both MPI tasks will not complete till the MPI_Recv is posted (which will never happen given the order above).

Deadlock Example – Simple Fix

- Change the order on one of the processes!
- Corrected code snippet:

```
if(myid == 0) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
  }
  else if(myid == 1) {
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    for (i = 0; i < 10; i++ ){
      printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
```

- MPI_Ssend: blocking synchronous send
- Now the MPI_Ssend on task 0 will complete since the corresponding MPI_Recv is posted first on task 1. (qsub deadlock-fix1.cmd)
- We will look at **Non-Blocking** options next.

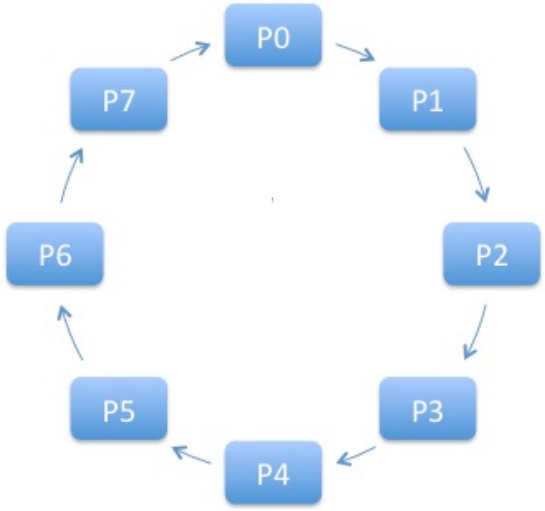
Non-Blocking MPI Send, Receive Routines

- Non-Blocking MPI Send, Receive routines return before there is any confirmation of receives or completion of the actual message copying operation.
- The routines simply put in the request to perform the operation.
- MPI wait routines can be used to check status and block till the operation is complete and it is safe to modify/use the information in the application buffer.
- Non-blocking approaches allow computations that don't depend on data in transit to continue while the communication operations are in progress.
- Allows for hiding the communication time with useful work and hence improves parallel efficiency.

MPI Ring-Topo

```
/* FILE: mpi_ringtopo.c
* DESCRIPTION: MPI tutorial example code: Non-Blocking Send/Receive
* AUTHOR: Blaise Barney;   LAST REVISED: 04/02/05 */
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]){
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);
printf("Task %d communicated with tasks %d & %d\n",rank,prev,next);

MPI_Finalize();
}
```

Virtual Ring Topology

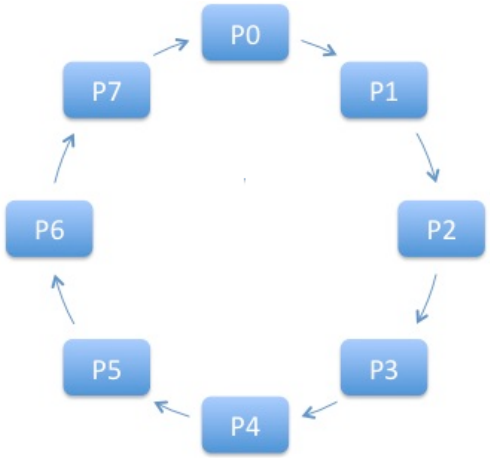


8 Processors arranged in a ring

Non-Blocking ISend, IRecv Example

Batch Script:

```
[point-to-point-comm]$ cat mpi-ringtopo.sb
#!/bin/bash
#SBATCH --job-name="mpi-ringtopo"
#SBATCH --output="mpi-ringtopo.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=1
#SBATCH --export=ALL
#SBATCH -t 00:04:00
#SBATCH -A sds173

## Environment
## MODULE ENV: updated 01/28/2020 (MPT)
 module purge
 module load slurm cpu gcc/10.2.0
 module load openmpi/4.0.4

## Use mpirun to run the job
mpirun -np 8  ./mpi_ringtopo
```



8 Processors arranged in a ring

Compile:

```
[mthomas@login02 point-to-point-comm]$ mpicc -o
mpi_ringtopo mpi_ringtopo.c
```

Submit Job:

```
[mthomas@login02 point-to-point-comm]$ sbatch mpi-
ringtopo.sb
Submitted batch job 1367573
```

Sample Output:

```
[mthomas@login02 point-to-point-comm]$ cat mpi-
ringtopo.1367573.exp-2-36.out
Task 2 communicated with tasks 1 & 3
Task 3 communicated with tasks 2 & 4
Task 4 communicated with tasks 3 & 5
Task 7 communicated with tasks 6 & 0
Task 5 communicated with tasks 4 & 6
Task 6 communicated with tasks 5 & 7
Task 0 communicated with tasks 7 & 1
Task 1 communicated with tasks 0 & 2
```

Collective MPI Routines

- Synchronization Routines: All processes in group/communicator wait till they get synchronized.
- Data Movement: Send/Receive data from all processes. E.g. Broadcast, Scatter, Gather, AlltoAll.
- Collective Computation (reductions): Perform reduction operations (min, max, add, multiply, etc.) on data obtained from all processes.
- Collective Computation and Data Movement combined (Hybrid).

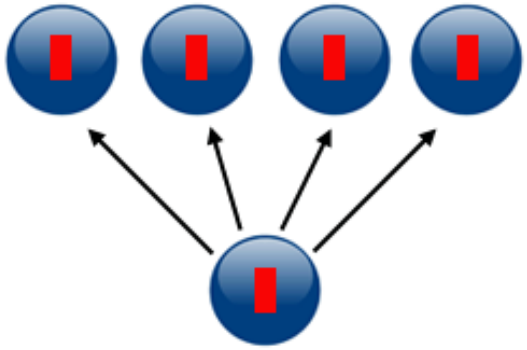
Synchronization Example

- **Our simple PI program had a synchronization example.**
- **Code Snippet:**

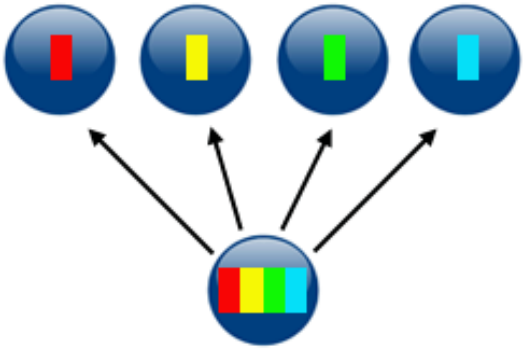
```
printf("Hello from MPI task= %d\n", rank);
MPI_Barrier(MPI_COMM_WORLD);
if (rank == 0)
  {
  printf("Number of MPI tasks = %d\n", numprocs);
  }
```

- **All tasks will wait till they are synchronized at this point.**

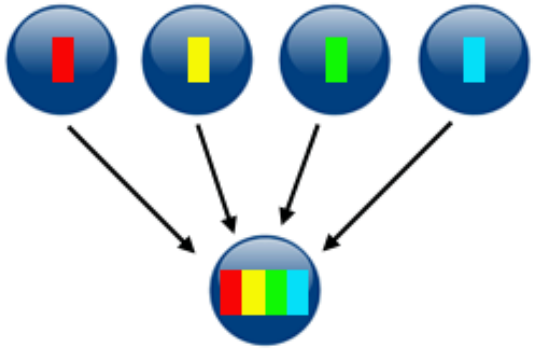
MPI Collectives



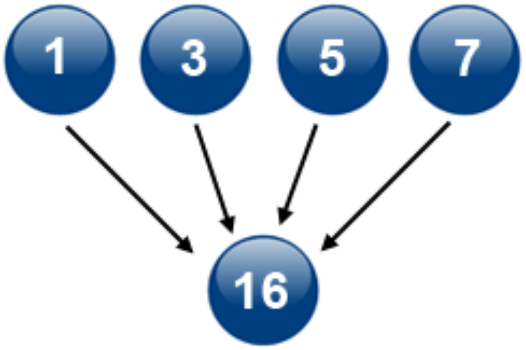
broadcast



scatter



gather



reduction

MPI Broadcast

- **Code Snippet (All collectives examples in $HOME/PARALLEL/COLLECTIVES):**

```
if(myid .eq. source)then
      do i=1,count
       buffer(i)=i
      enddo
  endif
  Call MPI_Bcast(buffer, count, MPI_INTEGER, source,&
        MPI_COMM_WORLD,ierr)
```

- **Compile:**

```
mpifort  -o bcast  bcast.f90
```

- **Run:**

```
sbatch bcast.sb
```

Compiling and Running bcast Example

```
program bcast
   include "mpif.h"
   integer myid, ierr,numprocs
   integer source,count
   integer buffer(4)
   integer status(MPI_STATUS_SIZE),request
   call MPI_INIT( ierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ie
)
   source=0
   count=4
   if(myid .eq. source)then
    do i=1,count
     buffer(i)=i
    enddo
   endif
   Call MPI_Bcast(buffer, count, MPI_INTEGER,source
        MPI_COMM_WORLD,ierr)
   write(*,*)"processor ",myid," got the ",buffer
   call MPI_FINALIZE(ierr)
   stop
end
```

```
[mthomas@login02 collectives]$ cat
bcast.1367736.exp-2-10.out
 processor  10 got  1  2  3  4
 processor   4 got  1  2  3  4
 processor   2 got  1  2  3  4
 processor   3 got  1  2  3  4
 processor   7 got  1  2  3  4
 processor   0 got  1  2  3  4
 processor   1 got  1  2  3  4
 processor  12 got  1  2  3  4
 processor  11 got  1  2  3  4
 processor  14 got  1  2  3  4
 processor  13 got  1  2  3  4
 processor   8 got  1  2  3  4
 processor   9 got  1  2  3  4
 processor   6 got  1  2  3  4
 processor   5 got  1  2  3  4
```

MPI Reduction

- **Reduction** is a classic concept from functional programming.
  - Data reduction involves reducing a set of numbers into a smaller set of numbers via a function.
- **Code Snippet:**

```
myidp1 = myid+1
call MPI_Reduce(myidp1,ifactorial,1,MPI_INTEGER,MPI_PROD,root,MPI_COMM_WORLD,ierr)
if (myid.eq.root) then
    write(*,*)numprocs,"! = ",ifactorial
endif
```

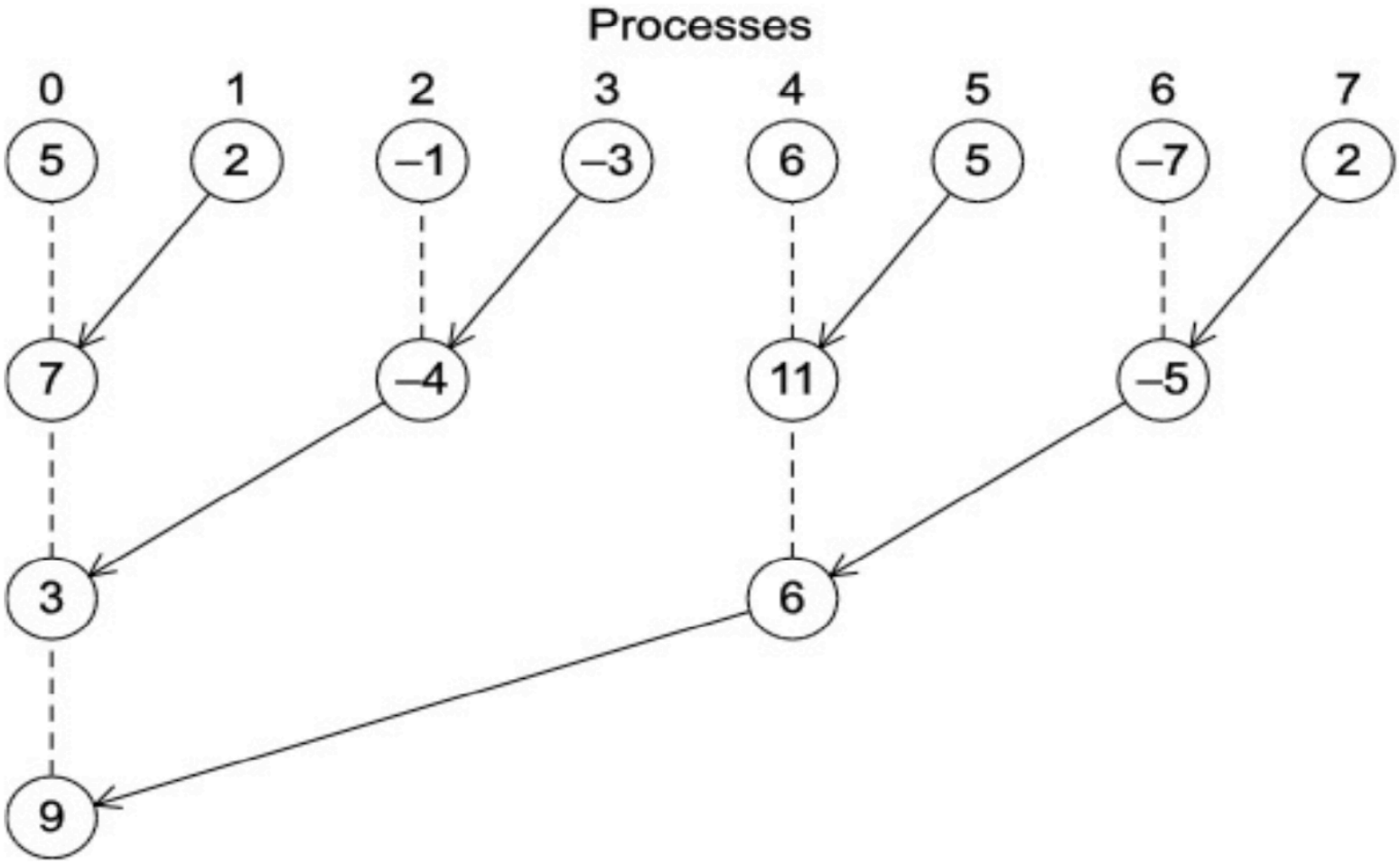
- **Compile:**

```
mpifort -o factorial.exe factorial.f90
```

- **Run:**

```
sbatch --res=SCCRES factorial.sb
```

MPI_Reduce (Add)



Compiling and Running Reduction Example

```
program factorial
   include "mpif.h"
   integer myid, ierr,numprocs
   integer myidp1, ifactorial, root
   integer status(MPI_STATUS_SIZE),request


   root = 0
   call MPI_INIT( ierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr
)
   myidp1 = myid+1
   call
MPI_Reduce(myidp1,ifactorial,1,MPI_INTEGER, MPI_PROD,
root,MPI_COMM_WORLD,ierr)

   if (myid.eq.root) then
      write(*,*)numprocs,"! = ",ifactorial
   endif
   call MPI_FINALIZE(ierr)
   stop
end
```

```
[mthomas@login02 collectives]$ cat
factorial.1367744.exp-1-34.out
           8 ! =        40320

[collectives]$ ./factorial_ser
           1           1
           2           2
           3           6
           4          24
           5         120
           6         720
           7        5040
           8       40320
           9      362880
          10     3628800
```

MPI_Allreduce

- **Code Snippet:**

```
imaxloc=IRAND(myid)
call MPI_ALLREDUCE(imaxloc,imax,1,MPI_INTEGER,MPI_MAX,MPI_COMM_WORLD,
mpi_err)
if (imax.eq.imaxloc) then
    write(*,*)"Max=",imax,"on task",myid
endif
```

- **Compile:**

```
mpifort -o allreduce.exe allreduce.f90
```

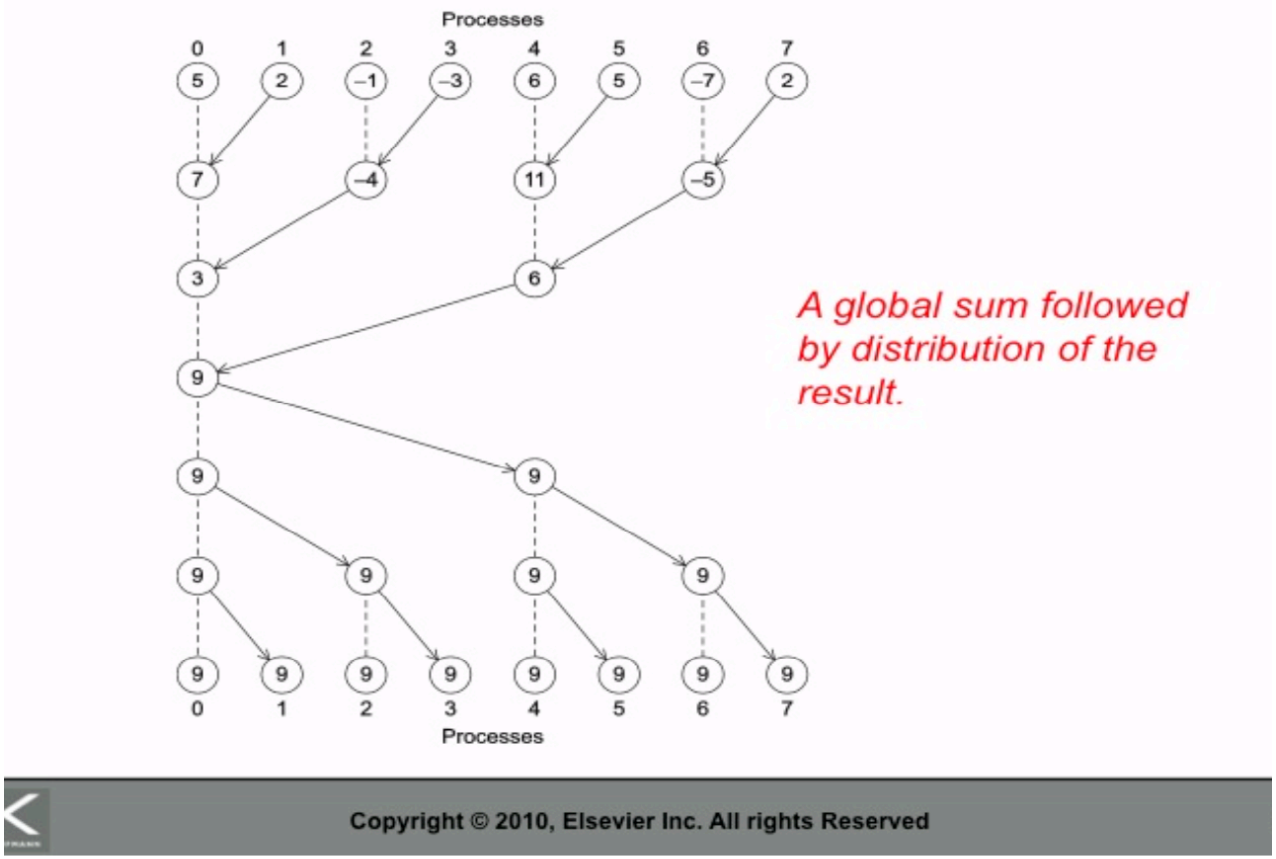
- **Run:**

```
sbatch --res=SCCRES allreduce.sb
```

- **Output:**

```
Max=      337897 on task          7
```

MPI_AllReduce: Butterfly pattern O(n)



Copyright © 2010, Elsevier Inc. All rights Reserved

MPI_AllReduce: Butterfly communication pattern, O(n)

Compiling and Running All_Reduce Example

```
program factorial
   include "mpif.h"
   integer myid, ierr,numprocs
   integer myidp1, ifactorial, root
   integer status(MPI_STATUS_SIZE),request


   root = 0
   call MPI_INIT( ierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr
)
   myidp1 = myid+1
   call
MPI_Reduce(myidp1,ifactorial,1,MPI_INTEGER,MPI_PROD,
root,MPI_COMM_WORLD,ierr)

   if (myid.eq.root) then
      write(*,*)numprocs,"! = ",ifactorial
   endif
   call MPI_FINALIZE(ierr)
   stop
end
```

```
[mthomas@login02 collectives]$ cat
factorial.1367744.exp-1-34.out
           8 ! =        40320

[collectives]$ ./factorial_ser
           1           1
           2           2
           3           6
           4          24
           5         120
           6         720
           7        5040
           8       40320
           9      362880
          10     3628800
```

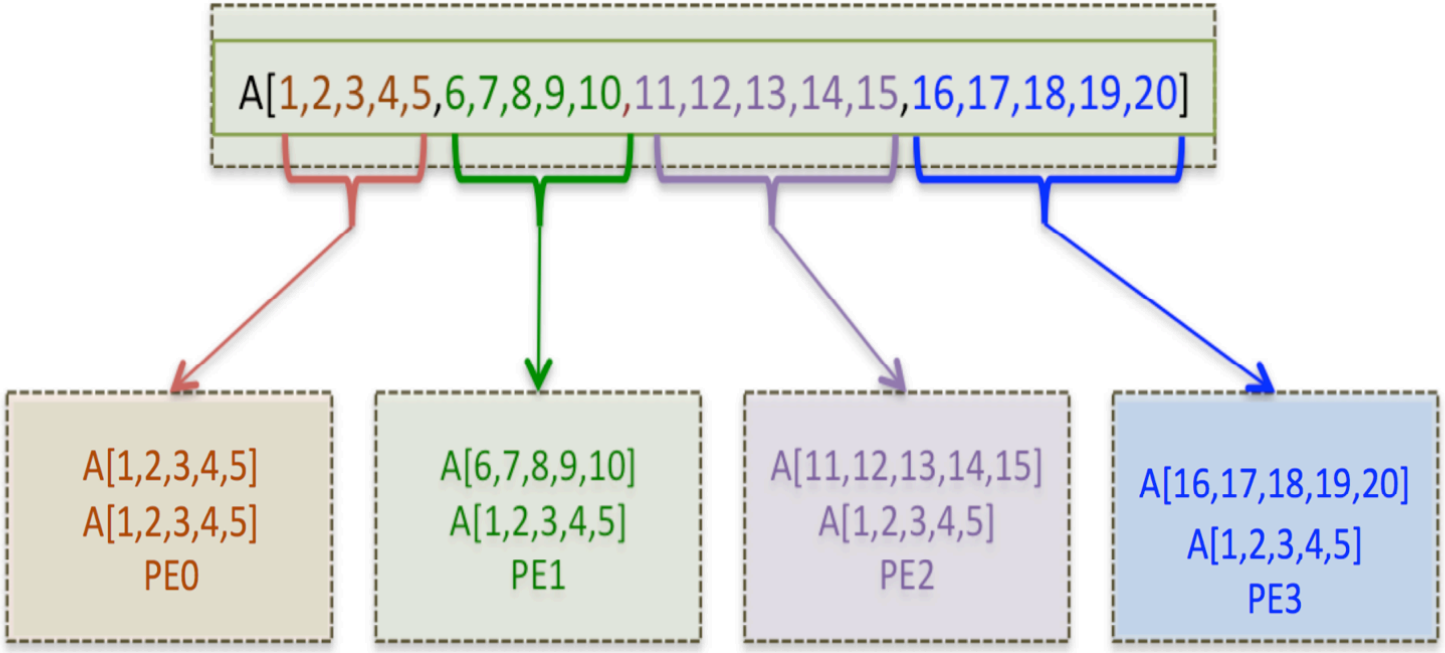
MPI Reduction Operations

NAME	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 XOR
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Decomposition and Mapping

- **Data and work decomposition**
  - Map partitioned domain to processes
- **Mapping**
  - Processes/ranks topology
  - System/Domain/Data
- **How to share data?**
  - Exchange messages and replicate data
- **Load imbalance**
  - What if the system is not regular?
  - Is work proportional to size of partitions?

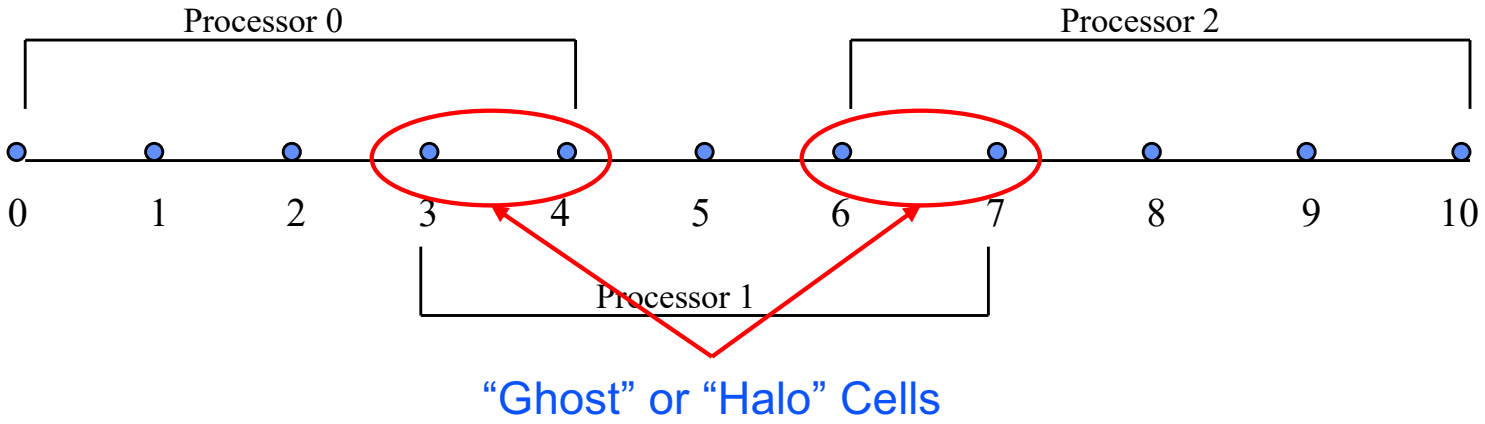
Distributing Data (1D Vector)



- 1D vector being distributed to 1D PE (logical) geometry.
- Distributing the ProbSize and Data
- Must be concerned about how the global problem maps onto local PE's
- MPI_Send/MPI_Recv required for data distribution, updates, collection.

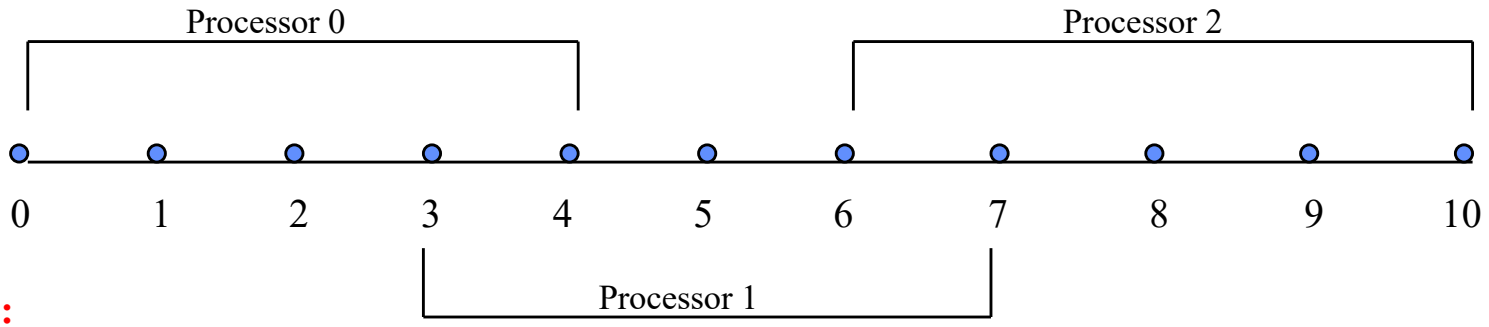
Simple Application using MPI: 1-D Heat Equation

- $\partial T/\partial t = \alpha(\partial^2 T/\partial x^2)$; T(0) = 0; T(1) = 0; (0 ≤x≤1)
  T(x,0) is know as an initial condition.
- Discretizing for numerical solution we get:
  $T^{(n+1)}_i - T^{(n)}_i = (\alpha\Delta t/\Delta x^2)(T^{(n)}_{i-1} - 2T^{(n)}_i + T^{(n)}_{i+1})$
  (central difference: *n* is the index in time and *i* is the index in space)
- In this example we solve the problem using 11 points and we distribute this problem over exactly 3 processors (for easy demo) shown graphically below:



"Ghost" or "Halo" Cells

Simple Application using MPI: 1-D Heat Equation



Processor 0:

Local Data Index : ilocal = 0 , 1, 2, 3, 4

Global Data Index: iglobal = 0, 1, 2, 3, 4

Solve the equation at (1,2,3)

Data Exchange: **Get 4 from processor 1; Send 3 to processor 1**

Processor 1:

Local Data Index : ilocal = 0, 1, 2, 3, 4

Global Data Index : iglobal = 3, 4, 5, 6, 7

Solve the equation at (4,5,6)

Data Exchange: **Get 3 from processor 0; Get 7 from processor 2; Send 4 to processor 0; Send 6 to processor 2**

Processor 2:

Local Data Index : ilocal = 0, 1, 2, 3, 4

Global Data Index : iglobal = 6, 7, 8, 9, 10

Solve the equation at (7,8,9)

Data Exchange: **Get 6 from processor 1; Send 7 to processor 1**

FORTRAN MPI CODE: 1-D Heat Equation

```
PROGRAM HEATEQN
  implicit none
  include "mpif.h"
  integer :: iglobal, ilocal, itime,ierr, nnodes, my_id
  integer :: dest, from, status(MPI_STATUS_SIZE),tag
  integer :: msg_size
  real*8 :: xalp,delx,delt,pi, T(0:100,0:5), TG(0:10)
  CHARACTER(20) :: FILEN

  delx = 0.1d0
  delt = 1d-4
  xalp = 2.0d0

  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nnodes, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)

  if (nnodes.ne.3) then
    if (my_id.eq.0) then
       print *, "This test needs exactly 3 tasks"
    endif
    call MPI_FINALIZE(ierr)
    STOP
  endif
 print *, "Process ", my_id, "of", nnodes ,"has started"
!*****Initial Conditions *********
      pi = 4d0*datan(1d0)
      do ilocal = 0, 4
        iglobal = 3*my_id+ilocal
        T(0,ilocal) = dsin(pi*delx*dfloat(iglobal))
      enddo
      write(*,*)"Processor", my_id, "has finished setting
     + initial conditions"
```

FORTRAN MPI CODE: 1-D Heat Equation

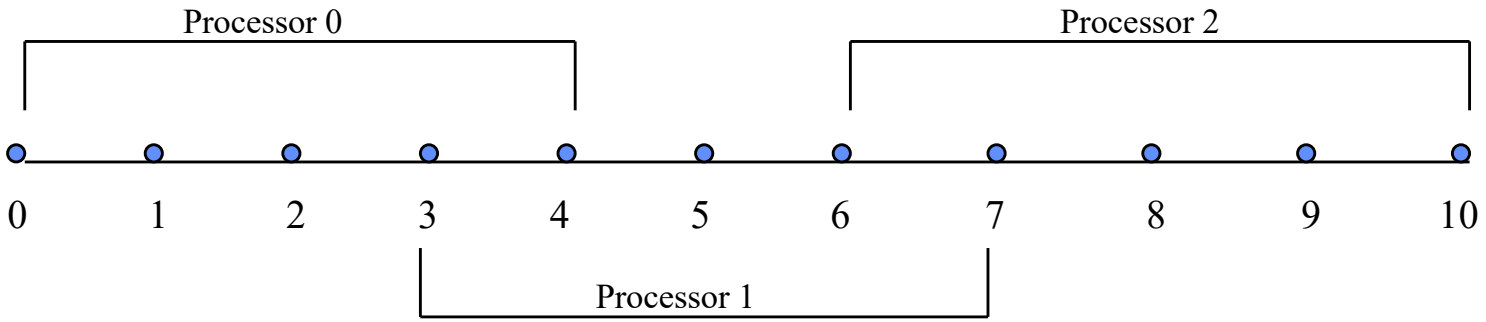
```
!****** Iterations ********
  do itime = 1 , 3
       if (my_id.eq.0) then
         write(*,*)"Running Iteration Number ", itime
       endif
       do ilocal = 1, 3
         T(itime,ilocal)=T(itime-1,ilocal)+
     +   xalp*delt/delx/delx*
     +   (T(itime-1,ilocal-1)-2*T(itime-
1,ilocal)+T(itime-1,ilocal+1))
       enddo
       if (my_id.eq.0) then
         write(*,*)"Sending and receiving overlap points"
         dest = 1
         msg_size = 1
         tag=2001
         call
MPI_SEND(T(itime,3),msg_size,MPI_DOUBLE_PRECISION,dest,
     +                 tag,MPI_COMM_WORLD,ierr)
       endif
       if (my_id.eq.1) then
         from = 0
         dest = 2
         msg_size = 1
         tag=2002
         call
MPI_SEND(T(itime,3),msg_size,MPI_DOUBLE_PRECISION,dest,
     +                 tag,MPI_COMM_WORLD,ierr)
         tag=MPI_ANY_TAG
         call
MPI_RECV(T(itime,0),msg_size,MPI_DOUBLE_PRECISION,from,
     +                 tag,MPI_COMM_WORLD,status,ierr)
       endif
```

```
       if (my_id.eq.2) then
         from = 1
         dest = 1
         msg_size = 1
         tag=2003
         call
MPI_SEND(T(itime,1),msg_size,MPI_DOUBLE_PRECISION,dest,
     +                 tag,MPI_COMM_WORLD,ierr)
         tag=MPI_ANY_TAG
         call
MPI_RECV(T(itime,0),msg_size,MPI_DOUBLE_PRECISION,from,
     +                 tag,MPI_COMM_WORLD,status,ierr)
       endif
       if (my_id.eq.1) then
         from = 2
          dest = 0
         msg_size = 1
         tag=MPI_ANY_TAG
         call
MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
     +                 tag,MPI_COMM_WORLD,status,ierr)
         tag=2004
         call
MPI_SEND(T(itime,1),msg_size,MPI_DOUBLE_PRECISION,dest,
     +                 tag,MPI_COMM_WORLD,ierr)
       endif
       if (my_id.eq.0) then
         from = 1
         msg_size = 1
         tag=MPI_ANY_TAG
         call
MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
     +                 tag,MPI_COMM_WORLD,status,ierr)
       endif
  enddo
```

Fortran MPI Code: 1-D Heat Equation (Contd.)

```
if (my_id.eq.0) then
  write(*,*)"SOLUTION SENT TO FILE AFTER 3 TIMESTEPS:"
endif
FILEN = 'data'//char(my_id+48)//'.dat'
open (5, file=FILEN)
write(5,*)"Processor ",my_id
do ilocal = 0 , 4
 iglobal = 3*my_id + ilocal
 write(5,*)"ilocal=",ilocal,";iglobal=",iglobal,";T=",T(3,ilocal)
enddo
close(5)
call MPI_FINALIZE(ierr)

END
```

Simple Application using MPI: 1-D Heat Equation



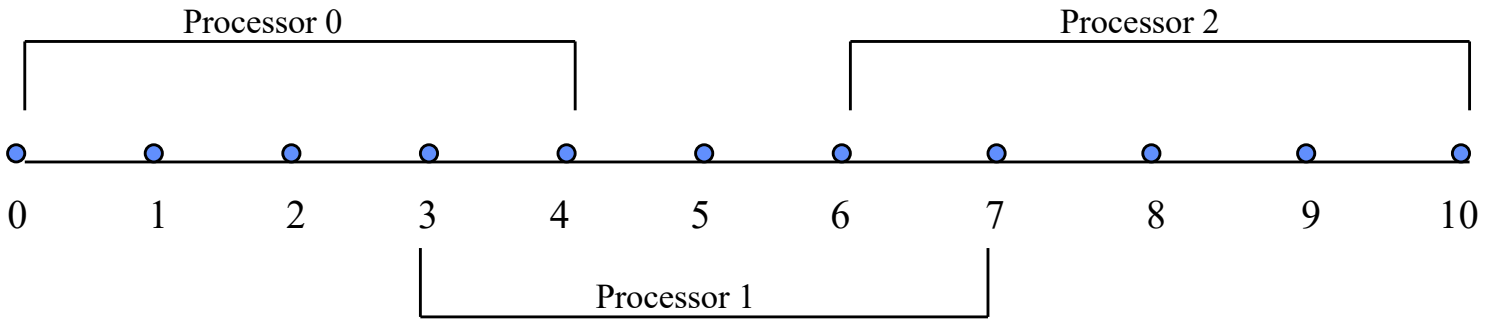
•Compilation

Fortran: mpifort -o heat_mpi.exe heat_mpi.f90

•Run Job:

sbatch --res=SCCRES heat_mpi.sb

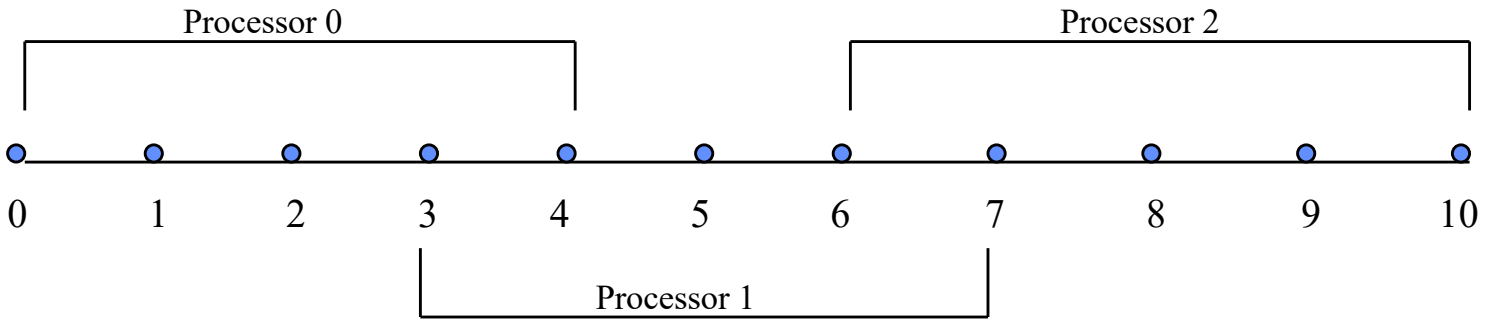
Simple Application using MPI: 1-D Heat Equation



OUTPUT FROM SAMPLE PROGRAM

```
Process  0 of 3 has started
 Processor 0 has finished                setting initial conditions
 Process  1 of 3 has started
 Processor 1 has finished                setting initial conditions
 Process  2 of 3 has started
 Processor 2 has finished                setting initial conditions
 Running Iteration Number  1
 Sending and receiving overlap points
 Running Iteration Number  2
 Sending and receiving overlap points
 Running Iteration Number  3
 Sending and receiving overlap points
 SOLUTION SENT TO FILE AFTER 3 TIMESTEPS:
```

Simple Application using MPI: 1-D Heat Equation

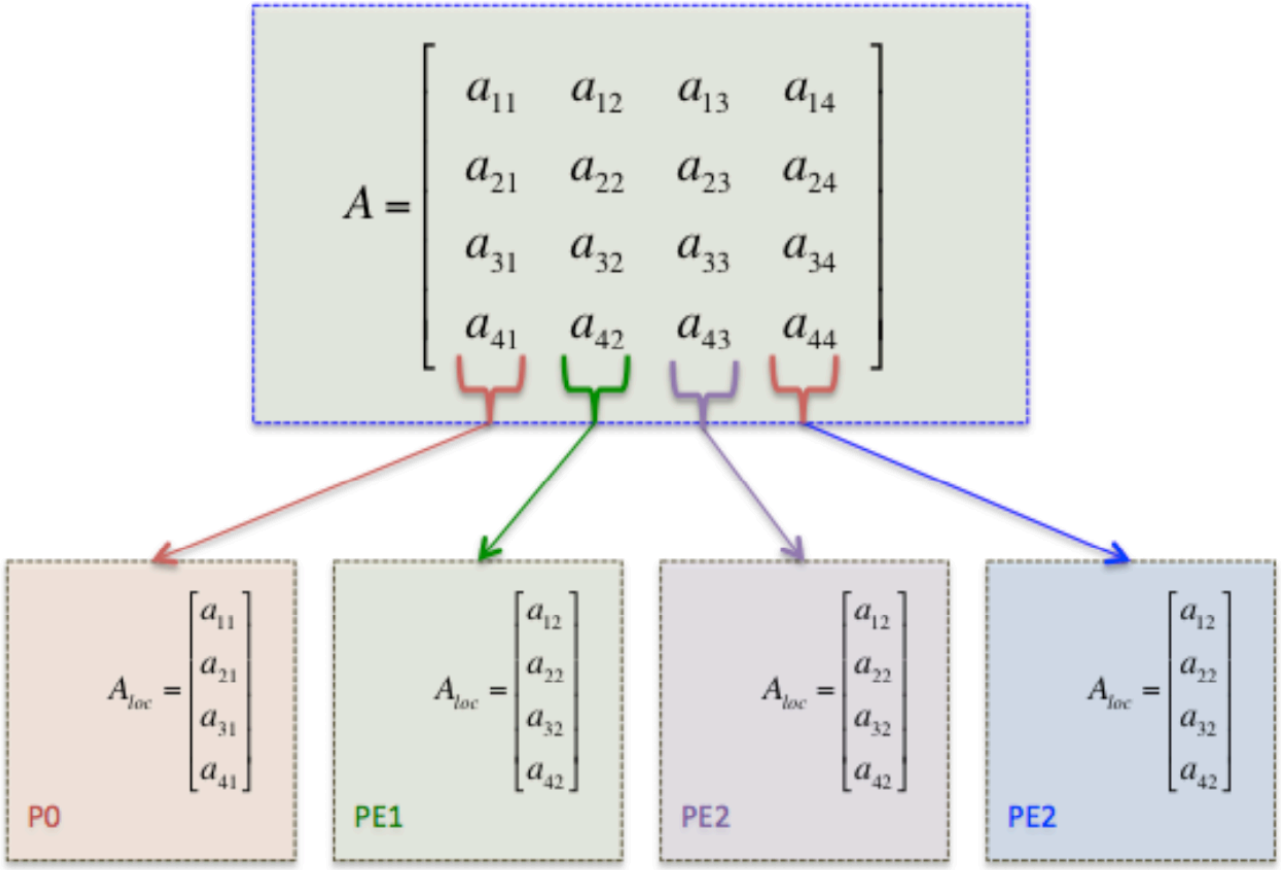



```
% more data0.dat
 Processor  0
 ilocal= 0 ;iglobal= 0 ;T= 0.0000000000000000E+00
 ilocal= 1 ;iglobal= 1 ;T= 0.307205621017284991
 ilocal= 2 ;iglobal= 2 ;T= 0.584339815421976549
 ilocal= 3 ;iglobal= 3 ;T= 0.804274757358271253
 ilocal= 4 ;iglobal= 4 ;T= 0.945481682332597884
```

```
% more data2.dat
 Processor  2
 ilocal= 0 ;iglobal= 6 ;T= 0.945481682332597995
 ilocal= 1 ;iglobal= 7 ;T= 0.804274757358271253
 ilocal= 2 ;iglobal= 8 ;T= 0.584339815421976660
 ilocal= 3 ;iglobal= 9 ;T= 0.307205621017285102
 ilocal= 4 ;iglobal= 10 ;T= 0.0000000000000000E+00
```

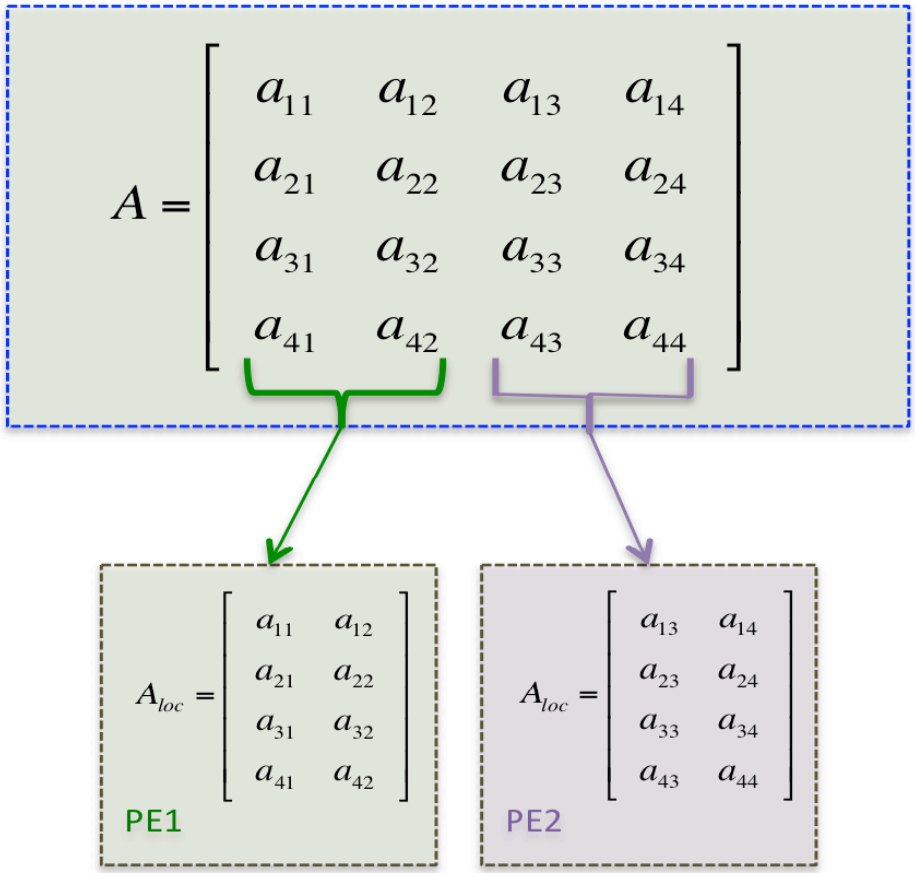
```
% more data1.dat
 Processor  1
 ilocal= 0 ;iglobal= 3 ;T= 0.804274757358271253
 ilocal= 1 ;iglobal= 4 ;T= 0.945481682332597884
 ilocal= 2 ;iglobal= 5 ;T= 0.994138272681972301
 ilocal= 3 ;iglobal= 6 ;T= 0.945481682332597995
 ilocal= 4 ;iglobal= 7 ;T= 0.804274757358271253
```

Matrix Data Distribution: 2D Matrix onto 4 PEs by columns



2D (4x4) matrix horizontal data Distribution onto a 1D processor arrangement using vertical slabs and 4 PE's.

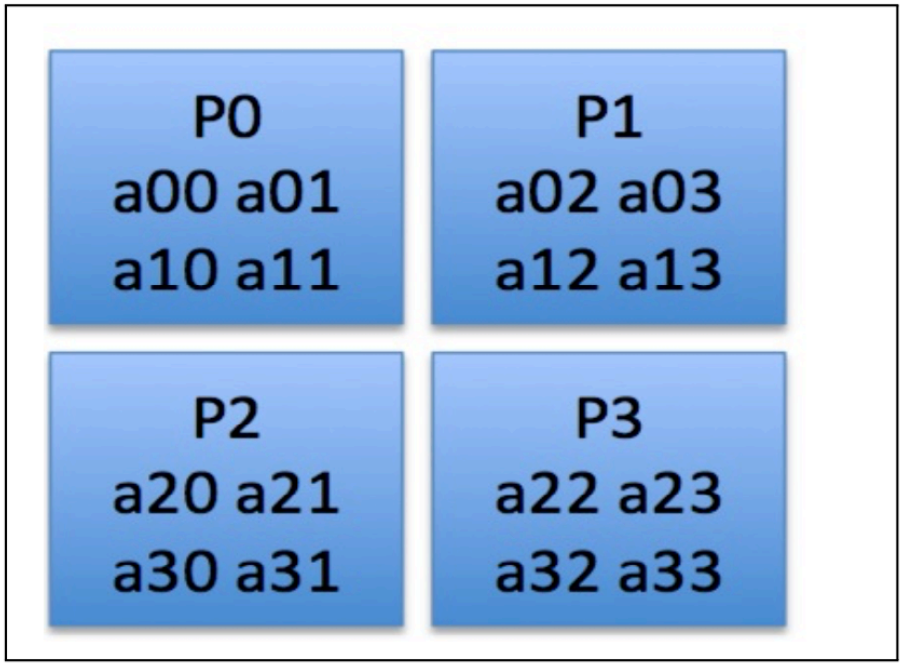
Data Distribution: 2D Matrix (2 PEs – using submatrices)



2D (4x4) matrix horizontal data Distribution onto a 1D processor arrangement using vertical slabs and 2 PE's.

2D "Checkerboard" Decomposition

- Use 2D cartesian mapping for Processors
- Use 2D cartesian mapping of the data
- Allocate space on each processor P_{ij} for subarrays of A, B, and C.
- Distribute A,B,C subarrays
- Calculate local data points for C
- Exchange A, B data as needed with neighbors



MPI – Profiling, Tracing Tools

- Several options available. On Comet we have mpiP, TAU, and Allinea MAP installed.
- Useful when you are trying to isolate performance issues.
- Tools can give you info on how much time is being spent in communication. The levels of detail vary with each tool.
- In general identify scaling bottlenecks and try to overlap communication with computation where possible.
- See recent Expanse Webinar: *Performance Tuning and Single Processor Optimization (*By Bob Sinkovits)
  - https://education.sdsc.edu/training/interactive/202102_perf_tuning_and_opt/index.html

mpiP example

- **Location: $HOME/PARALLEL/MISC**
- **Compile:**

```
mpifort -nofree -g -o heat_mpi_profile.exe heat_mpi.f90 -
L/share/apps/compute/mpiP/v3.4.1/mv2/lib -lmpiP -L/share/apps/compute/libiberty -liberty -
L/share/apps/compute/libunwind/v1.1/mv2/lib -lunwind
```

- **Executable already exists. Just submit heat_mpi_profile.sb.**
- **Once the job runs you get a .mpiP file.**

mpiP output

```
@ Command : ./heat_mpi_profile.exe
@ Version                  : 3.4.1
@ MPIP Build date          : Aug  9 2018, 12:07:41
@ Start time               : 2018 08 09 12:12:23
@ Stop time                : 2018 08 09 12:12:23
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 31588
@ Final Output Dir         : .
@ Report generation        : Single collector task
@ MPI Task Assignment      : 0 comet-06-13.sdsc.edu
@ MPI Task Assignment      : 1 comet-06-13.sdsc.edu
@ MPI Task Assignment      : 2 comet-06-13.sdsc.edu

---------------------------------------------------------------------------
@--- MPI Time (seconds) ------------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0      0.024    0.00989    41.22
   1     0.0243     0.0099    40.81
   2     0.0243   0.000126     0.52
```

mpiP Output

```
   *       0.0726       0.0199      27.44
---------------------------------------------------------------------------
@--- Callsites: 1 ------------------------------------------------------------
---------------------------------------------------------------------------
 ID Lev File/Address        Line Parent_Funct             MPI_Call
  1   0 0x40ca45                  [unknown]               Recv
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) -------------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Recv                    1       19.5   26.87   97.93    0.86
Send                    1      0.412    0.57    2.07    0.21
---------------------------------------------------------------------------
@--- Aggregate Sent Message Size (top twenty, descending, bytes) -------------
---------------------------------------------------------------------------
Call                 Site      Count      Total       Avrg  Sent%
Send                    1         12         96          8 100.00
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 6 -------------------------
---------------------------------------------------------------------------
Name                 Site Rank  Count      Max      Mean       Min   App%   MPI%
Recv                    1    0      3     9.73      3.25   0.00504  40.63  98.55
Recv                    1    1      6     9.69      1.62   0.00194  40.14  98.36
```

mpiP output

```
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 6 ----------------------
---------------------------------------------------------------------------
Name                 Site Rank  Count      Max      Mean       Min   App%   MPI%
Recv                    1    0      3     9.73      3.25   0.00504  40.63  98.55
Recv                    1    1      6     9.69      1.62   0.00194  40.14  98.36
Recv                    1    2      3    0.015   0.00664   0.00188   0.08  15.83
Send                    1    0      3    0.136    0.0477   0.00303   0.60   1.45
Send                    1    1      6    0.146    0.0272   0.00203   0.67   1.64
Send                    1    2      3   0.0999    0.0353   0.00298   0.44  84.17
Send                    1    *     24     9.73      0.83   0.00188  27.44 100.00
---------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) -------------------
---------------------------------------------------------------------------
Name                 Site Rank   Count      Max       Mean       Min       Sum
Send                    1    0       3        8          8          8        24
Send                    1    1       6        8          8          8        48
Send                    1    2       3        8          8          8        24
Send                    1    *      12        8          8          8        96
---------------------------------------------------------------------------
@--- End of Report --------------------------------------------------------
---------------------------------------------------------------------------
```

Exercise

- Use MPI to do the Matrix-Vector Multiplication for a Hilbert matrix. Analytic result available for verification.
- Show parallel speedup and efficiency up to 2 nodes (48 tasks)

A Hilbert matrix is a square matrix with elements that are unit fractions given by

$$H_{ij} = \frac{1}{i+j-1}$$

For example, the Hilbert matrix of dimension 4 is

$$\mathbf{H} = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

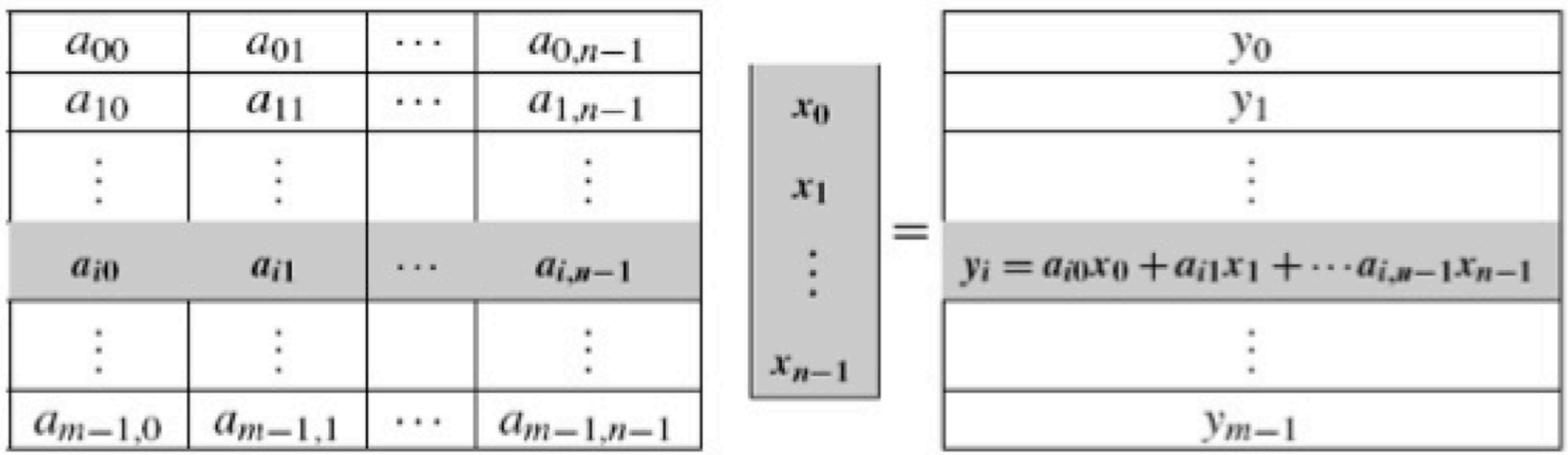
If $\mathbf{H}$ is a Hilbert matrix of dimension $n = 122880$ and $\mathbf{x} = [1 \cdots 1]^T$ is an all-ones vector of the same dimension, compute the resultant vector $\mathbf{y} = \mathbf{H}\mathbf{x}$.

Check your result by computing the sum of the elements of $\mathbf{y}$. For $\mathbf{H}$ and $\mathbf{x}$ of dimension n, the sum of the elements of $\mathbf{y}$ is given analytically by

$$\sum_{i=1}^{n} y_i = n + \sum_{j=1}^{n-1} \frac{n-j}{n+j}.$$

Extra credit: Optimize your code and recompute for both $n = 122880$ and $n = 1048576$. Plot the parallel speedup and efficiency of your code.

Matrix-Vector Multiplication



References

- **Excellent tutorials from LLNL:**
  - https://computing.llnl.gov/tutorials/mpi/
  - https://computing.llnl.gov/tutorials/openMP/
- **MPI for Python:**
  - https://mpi4py.readthedocs.io/en/stable/
- **MVAPICH2 User Guide:**
  - http://mvapich.cse.ohio-state.edu/userguide/