# Parallel Computing Using MPI

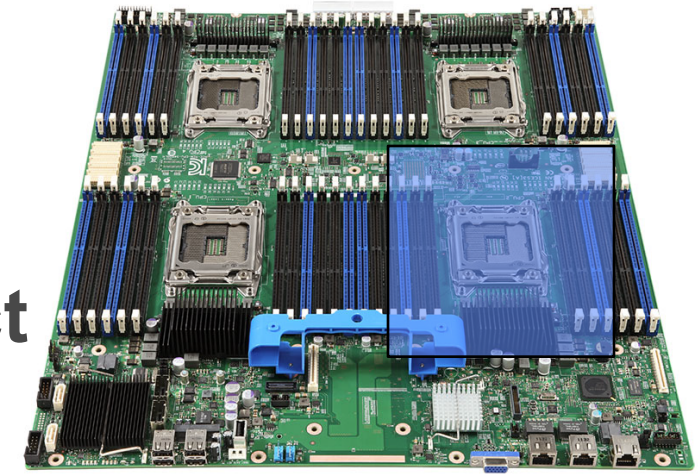
Mahidhar Tatineni

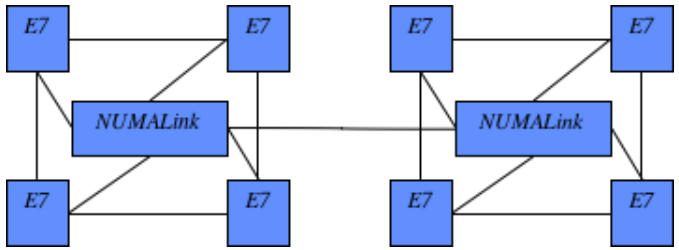
02/07/2020

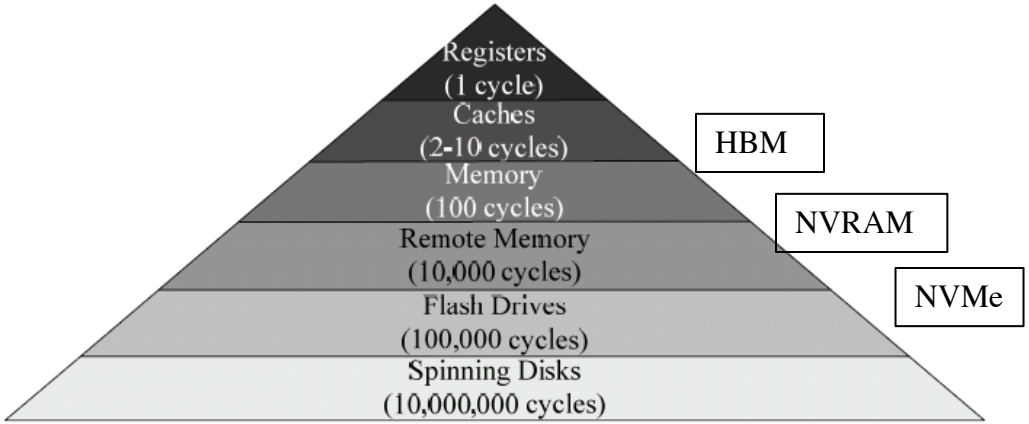
(mahidhar@sdsc.edu)

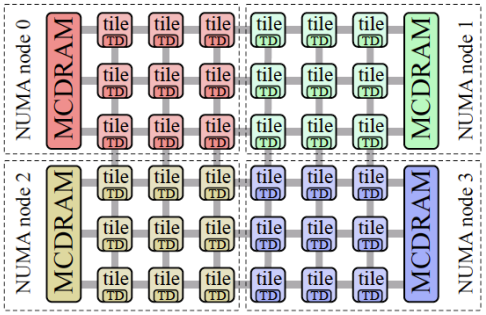
# Current Supercomputer Architectures

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



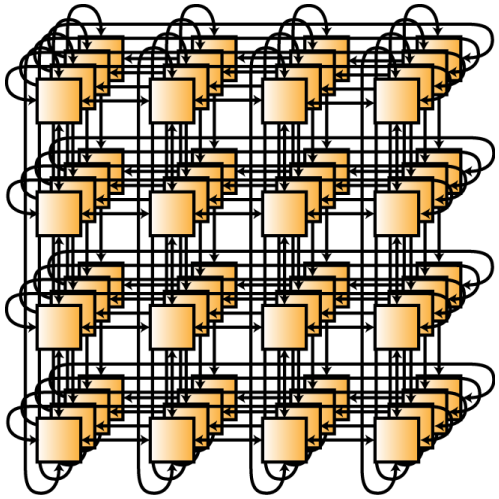


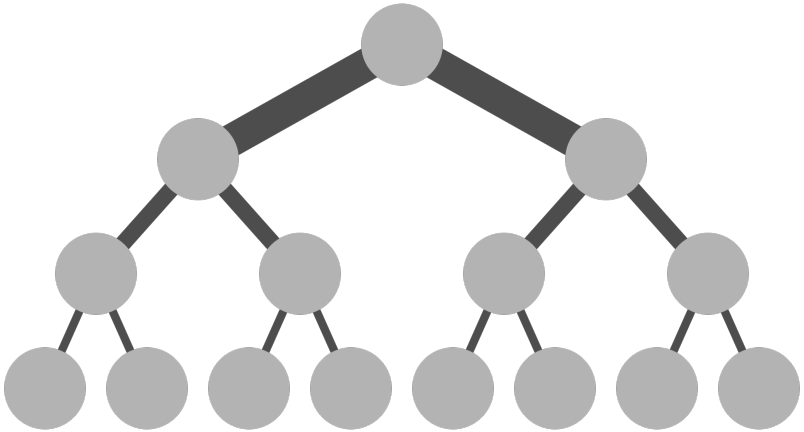



# Network Topologies

- **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 - Flynn's Taxonomy

| | |
|---|---|
| Single Instruction Single Data | Single Instruction Multiple Data |
| Multiple Instruction Single Data | Multiple Instruction Multiple Data |

- **Single Program Multiple Data (SPMD)**
- **Multiple Program Multiple Data (MPMD)**

- **Single Instruction, Single Data [Serial codes]**
- **Single Instruction, Multiple Data**
  - Processors run the same instructions, each operates on different data
  - Technically, Hadoop MapReduce fits this mode
  - GPUs
- **Multiple Instruction, Single Data**
  - Multiple instructions acting on single data stream. E.g. Different analysis on same set of data.
- **Multiple Instruction, Multiple Data**
  - 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

# 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

# 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-fact standard for parallel computing, portable, system specific optimizations without changing code interface
  - http://www.mpi-forum.org
  - Several implementations - e.g MVAPICH2 (default on Comet), Intel MPI, and 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

# Examples

- Copy the directory to your home directory:

```
cp -r /share/apps/examples/SCC/PARALLEL $HOME
```

# 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 Program to Compute PI

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

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

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

 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();
}
```

# 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.!

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

# Compiling and Running PI Example

**cd /home/$USER/PARALLEL/SIMPLE**

**Compile: mpicc -o pi_mpi.exe pi_mpi.c**

**Submit Job: sbatch --res=SCCRES pi_mpi.sb**

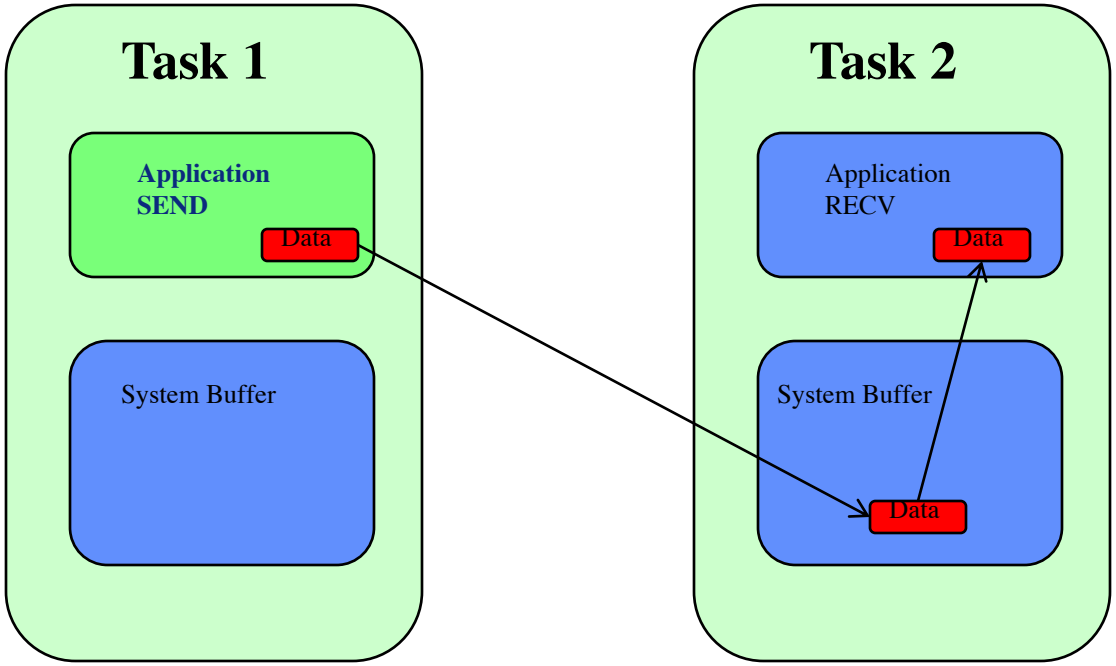
***Sample Output:***

```
Hello from MPI task= 12
Hello from MPI task= 14
Hello from MPI task= 8
Hello from MPI task= 3
Hello from MPI task= 2
Hello from MPI task= 4
Hello from MPI task= 13
Hello from MPI task= 9
Hello from MPI task= 5
Hello from MPI task= 1
Hello from MPI task= 11
Hello from MPI task= 10
Hello from MPI task= 15
Hello from MPI task= 7
Hello from MPI task= 6
Hello from MPI task= 0
Number of MPI tasks = 16
Pi = 3.141594606714
```

# 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

- 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.
- Blocking receive returns once the data is in the application buffer (receive data) and can by used by the application.

# Blocking Send, Recv Example (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] );
      }
   }
```

# Blocking Send, Recv Example

**Location:$HOME/PARALLEL/PTOP**

**Compile: mpicc -o blocking.exe blocking.c**

**Submit Job: sbatch --res=SCCRES blocking.sb**

**Output:**

**r_buf[0] = 0.000000**
**r_buf[1] = 4.000000**
**r_buf[2] = 8.000000**
**r_buf[3] = 12.000000**
**r_buf[4] = 16.000000**
**r_buf[5] = 20.000000**
**r_buf[6] = 24.000000**
**r_buf[7] = 28.000000**
**r_buf[8] = 32.000000**
**r_buf[9] = 36.000000**

# Deadlocking MPI Tasks

- Take care to sequence blocking send/recvs. Easy to deadlock processes waiting on each other with circular dependencies.
- Can also occur with control errors and unexpected symantics
- For example, take the following 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] );
    }
```

- The MPI_Ssend on both tasks will not complete till the MPI_Recv is posted (which will never happen given the order).

# Deadlock Example

- **Location: $HOME/PARALLEL/PTOP**
- **Compile: mpicc -o deadlock.exe deadlock.c**
- **Submit Job: sbatch --res=SCCRES deadlock.sb**
- **It should technically finish in less than a second since the data transferred is a few bytes. However, the code deadlocks and hits the wallclock limit (2 minutes in the script).**
- **Error info:**

```
[etrain68@comet-ln3 PTOP]$ more deadlock.31388191.comet-07-10.out
slurmstepd: *** JOB 31388191 ON comet-07-10 CANCELLED AT 2020-02-07T06:47:01
DUE TO TIME LIMIT ***
[comet-07-10.sdsc.edu:mpirun_rsh][signal_processor] Caught signal 15, killing job
Connection to comet-07-10 closed by remote host.
```

# Deadlock Example – Simple Fix

- Change the order on one of processes!
- For example, take the following 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] );
    }
```

- 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.

# Deadlock Example (Fix 1)

- Location: $HOME/PARALLEL/PTOP
- Compile: mpicc -o deadlock-fix1.exe deadlock-fix1.c
- Submit Job: sbatch --res=SCCRES deadlock-fix1.sb
- Fix works!

```
$ more deadlock-fix1.out
r_buf[0] = 0.000000
r_buf[1] = 4.000000
r_buf[2] = 8.000000
r_buf[3] = 12.000000
r_buf[4] = 16.000000
r_buf[5] = 20.000000
r_buf[6] = 24.000000
r_buf[7] = 28.000000
r_buf[8] = 32.000000
r_buf[9] = 36.000000
```

# 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.
- This non-blocking approaches allows computations (that don't depend on this data in transit) to continue while the communication operations are in progress. This allows for hiding the communication time with useful work and hence improves parallel efficiency.

# Non-Blocking Send, Recv Example

- **Example uses MPI_Isend, MPI_Irecv, MPI_Wait**
- **Code snippet:**

```
if(myid == source){
        s_buf=1024;
        MPI_Isend(&s_buf,count,MPI_INT,destination,tag,MPI_COMM_WORLD,&request);
    }
    if(myid == destination {
        MPI_Irecv(&r_buf,count,MPI_INT,source,tag,MPI_COMM_WORLD,&request);
    }
    MPI_Wait(&request,&status);
```

- **Compile & Run:**

**mpicc -o nonblocking.exe nonblocking.c**

**qsub nonblocking.cmd**

***Sample output:***

**processor 0 sent 1024**

**processor 1 got 1024**

# Deadlock Example – Non-Blocking Option

- Change the order on one of processes!
- For example, take the following code snippet:

```
if(myid == 0) {
    MPI_Isend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD, &request);
    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] );
    }
```

- 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.

# Deadlock Example (Fix 2)

- Location: $HOME/PARALLEL/PTOP
- Compile: mpicc -o deadlock-fix2-nb.exe deadlock-fix2-nb.c
- Submit Job: sbatch --res=SCCRES deadlock-fix2-nb.sb
- Fix works!

```
$ more deadlock-fix2-nb.out
r_buf[0] = 0.000000
r_buf[1] = 4.000000
r_buf[2] = 8.000000
r_buf[3] = 12.000000
r_buf[4] = 16.000000
r_buf[5] = 20.000000
r_buf[6] = 24.000000
r_buf[7] = 28.000000
r_buf[8] = 32.000000
r_buf[9] = 36.000000
```

# 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.**

# Broadcast Example

- **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:**
  - **mpif90 -o bcast.exe bcast.f90**
- **Run:**
  - **sbatch --res=SCCRES bcast.sb**
- **Output:**

```
processor       1  got       1       2       3       4
processor       0  got       1       2       3       4
processor       2  got       1       2       3       4
processor       3  got       1       2       3       4
```

# Reduction Example

- **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:**

```
mpif90 -o factorial.exe factorial.f90
```

- **Run:**

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

- **Output:**

```
8 ! =       40320
```

# MPI_Allreduce example

- **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:**

```
mpif90 -o allreduce.exe allreduce.f90
```

- **Run:**

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

- **Output:**

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

# 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 Reduction Operations

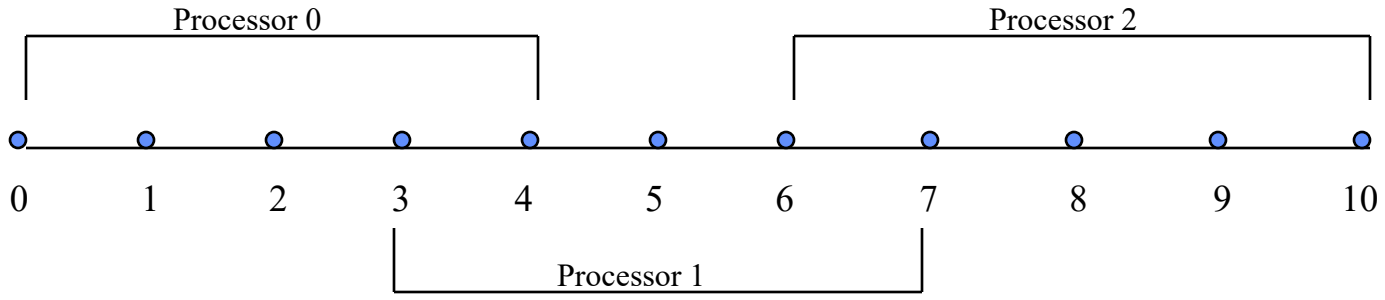
| NAME | OPERATION |
|---|---|
| MPI_MAX | Maximum |
| MPI_MIN | Minimum |
| MPI_SUM | Sum |
| MPI_PROD | Product |
| MPI_LAND | Logical AND |
| MPI_BAND | Bit-wise AND |
| MPI_LOR | Logical OR |
| MPI_BOR | Bit-wise OR |
| MPI_LXOR | Logical XOR |
| MPI_BXOR | Bit-wise XOR |
| MPI_MAXLOC | Maximum value and location |
| MPI_MINLOC | Minimum value and location |

# Decomposition and Mapping

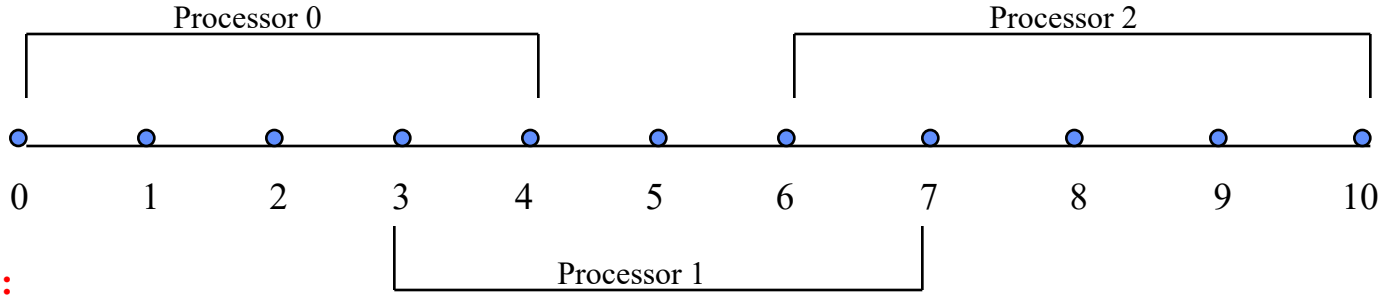
- **Data and work decomposition**
  - Map partitioned domain to processes
- **Mapping**
  - Processes/ranks topology
  - System/Domain/Data
- **How to share data?**
  - Exchange messages and replicate data
- **Load imbalance**
  - What if the system is not regular?
  - Is work proportional to size of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

# 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:
  $T^{(n+1)}_i - T^{(n)}_i = (\alpha\Delta t/\Delta x^2)(T^{(n)}_{i-1} - 2T^{(n)}_i + T^{(n)}_{i+1})$
  (*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:



# 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

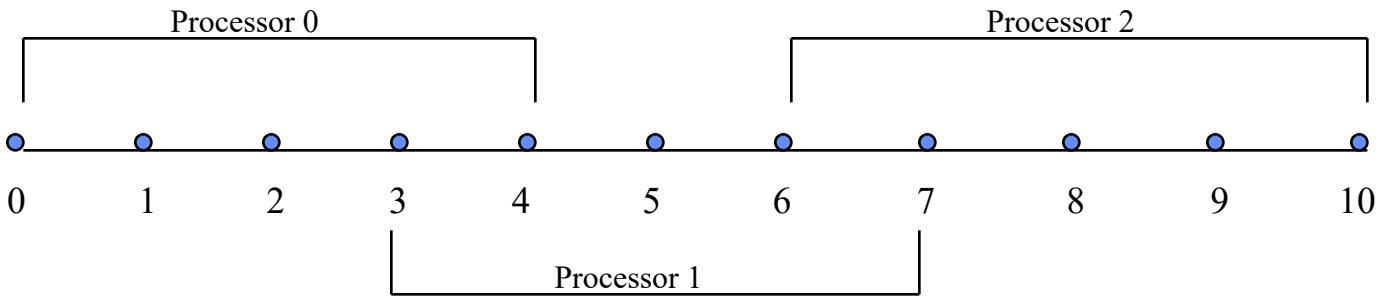
```
  PROGRAM HEATEQN
    implicit none
    include "mpif.h"
    integer :: iglobal, ilocal, itime
    integer :: ierr, nnodes, my_id
    integer :: dest, from, status(MPI_STATUS_SIZE),tag
    integer :: msg_size
    real*8 :: xalp,delx,delt,pi
    real*8 :: T(0:100,0:5), TG(0:10)
    CHARACTER(20) ::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

```
     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"
!************* 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
```

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

```
 msg_size = 1
      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
        call
MPI_SEND(T(itime,3),msg_size,MPI_DOUBLE_PRECISION,dest,
     +               tag,MPI_COMM_WORLD,ierr)
        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
        call
MPI_SEND(T(itime,1),msg_size,MPI_DOUBLE_PRECISION,dest,
     +               tag,MPI_COMM_WORLD,ierr)
        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
        call MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
     +               tag,MPI_COMM_WORLD,status,ierr)
        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
        call MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
     +               tag,MPI_COMM_WORLD,status,ierr)
      endif
      enddo

      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



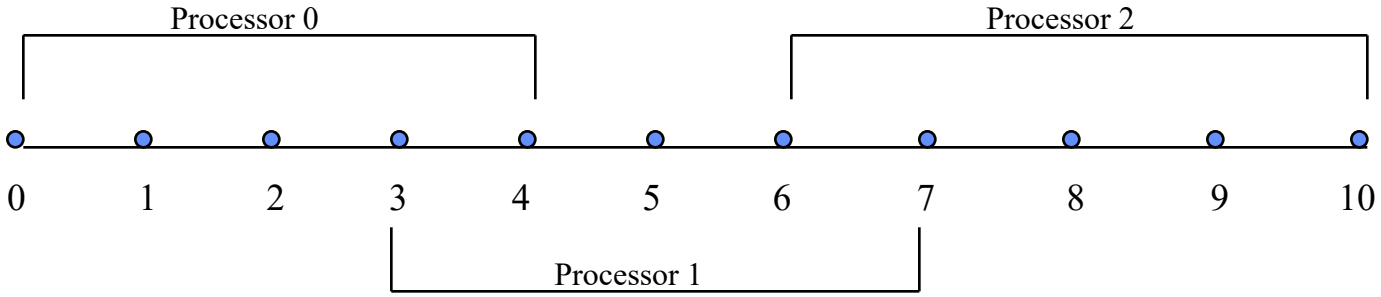

•Compilation

Fortran: mpif90 -nofree -o heat_mpi.exe heat_mpi.f90

•Run Job:

sbatch --res=SCCRES heat_mpi.sb

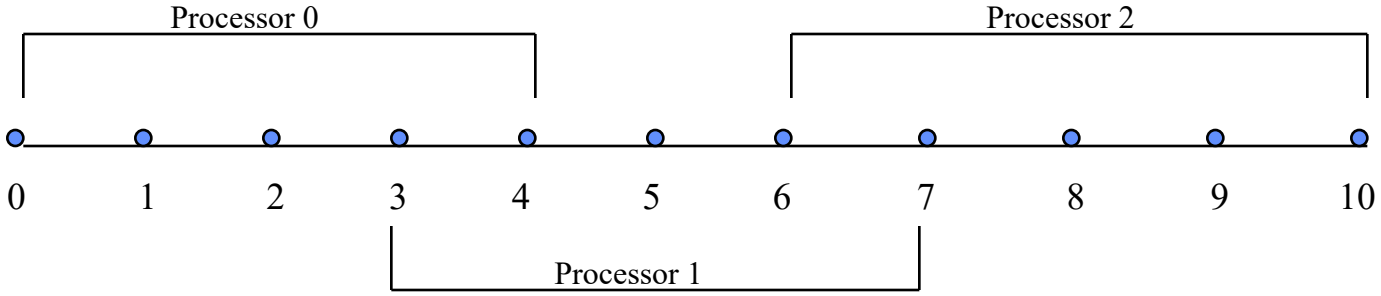
# Simple Application using MPI: 1-D Heat Equation




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
```

# 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.

# mpiP example

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

  mpif90 -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

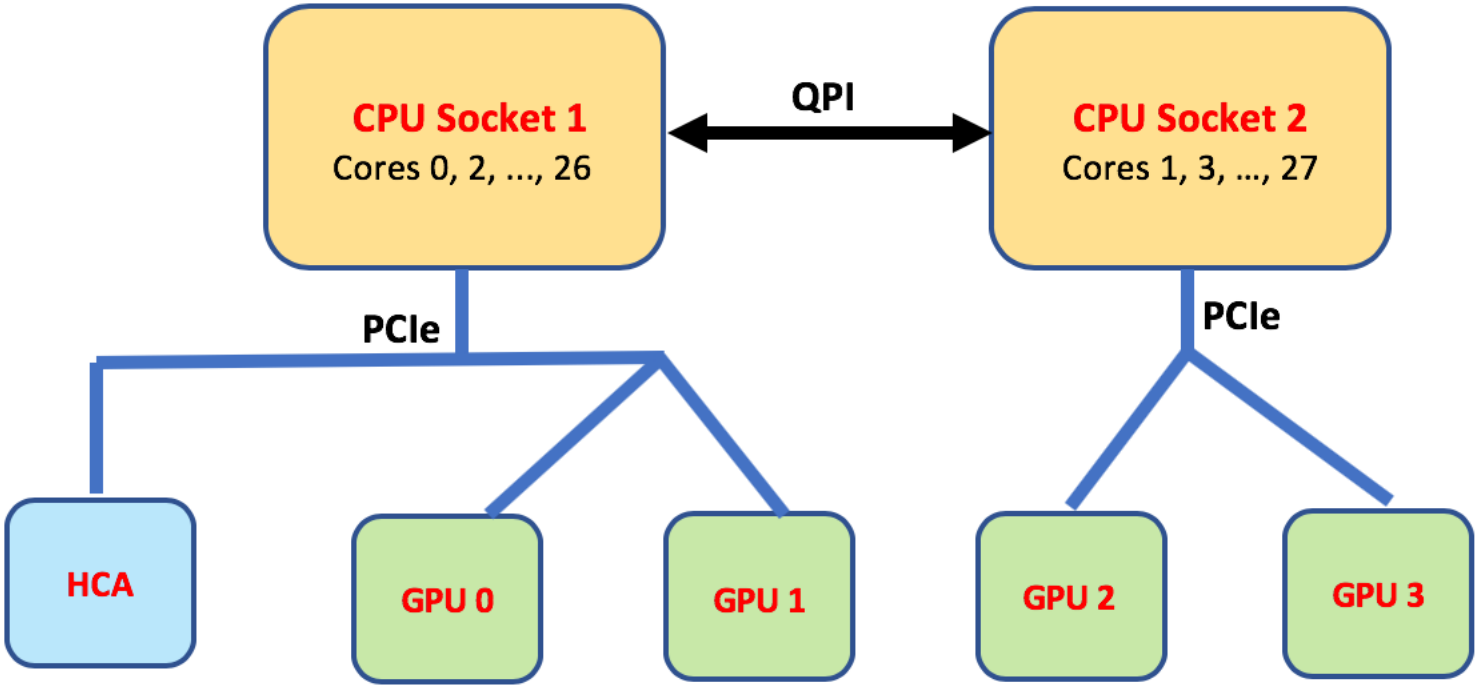
```
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 6 ----------------------
---------------------------------------------------------------------------
Name                 Site Rank  Count      Max       Mean        Min   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 --------------------------------------------------------
---------------------------------------------------------------------------
```

# More Complex routines

- Derived Data Types
- User defined reduction functions
- Groups/communicator management
- Parallel I/O
- One Sided Communication Routines (RDMA)
- MPI-3 Standard has over 400 routines(!).

# MVAPICH2-GDR

Comet P100 node layout

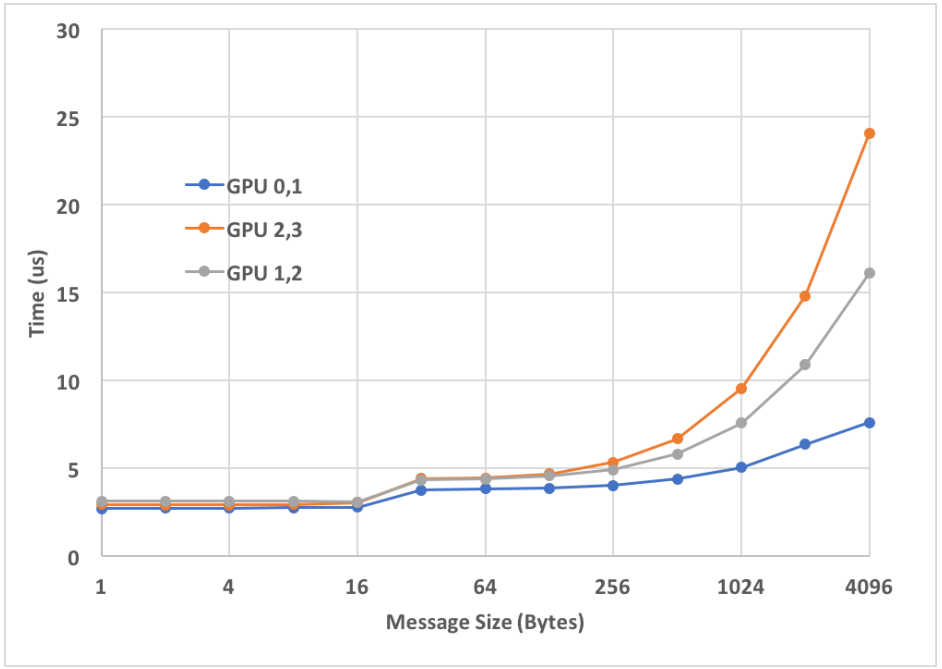


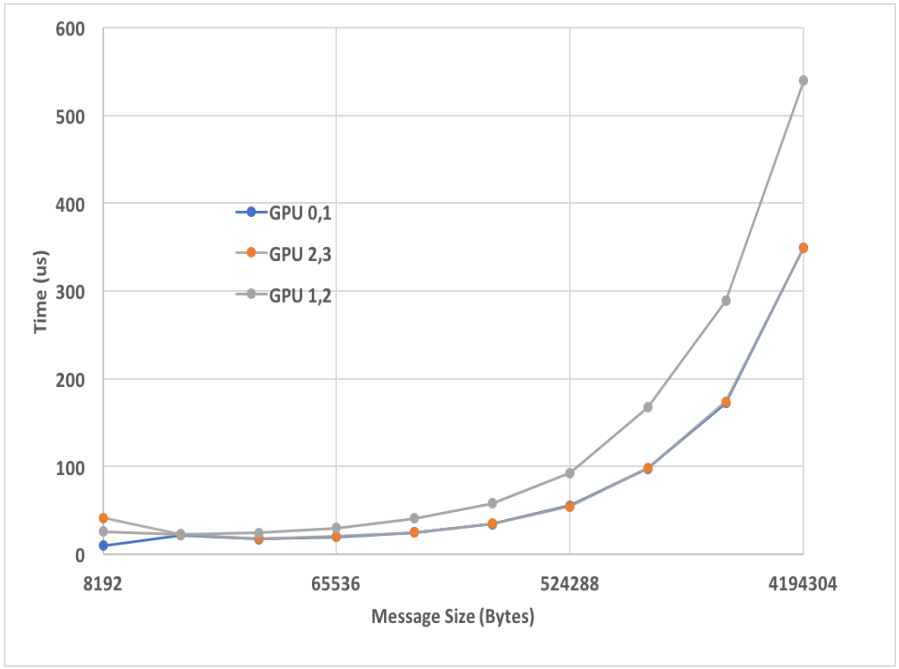
# Comet P100 node architecture

```
        GPU0    GPU1    GPU2    GPU3    mlx4_0  CPU Affinity
GPU0     X      PIX     SOC     SOC     PHB     0-0,2-2,4-4,6-6,8-8,10-10,12-12,14-14,16-16,18-18,20-20,22-22,24-24,26-26
GPU1    PIX      X      SOC     SOC     PHB     0-0,2-2,4-4,6-6,8-8,10-10,12-12,14-14,16-16,18-18,20-20,22-22,24-24,26-26
GPU2    SOC     SOC      X      PIX     SOC     1-1,3-3,5-5,7-7,9-9,11-11,13-13,15-15,17-17,19-19,21-21,23-23,25-25,27-27
GPU3    SOC     SOC     PIX      X      SOC     1-1,3-3,5-5,7-7,9-9,11-11,13-13,15-15,17-17,19-19,21-21,23-23,25-25,27-27
mlx4_0  PHB     PHB     SOC     SOC      X

Legend:

  X    = Self
  SOC  = Connection traversing PCIe as well as the SMP link between CPU sockets(e.g. QPI)
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing a single PCIe switch
  NV#  = Connection traversing a bonded set of # NVLinks
```

- 4 GPUs per node
- GPUs (0,1) and (2,3) can do P2P communication
- Mellanox InfiniBand adapter associated with first socket (GPUs 0, 1)

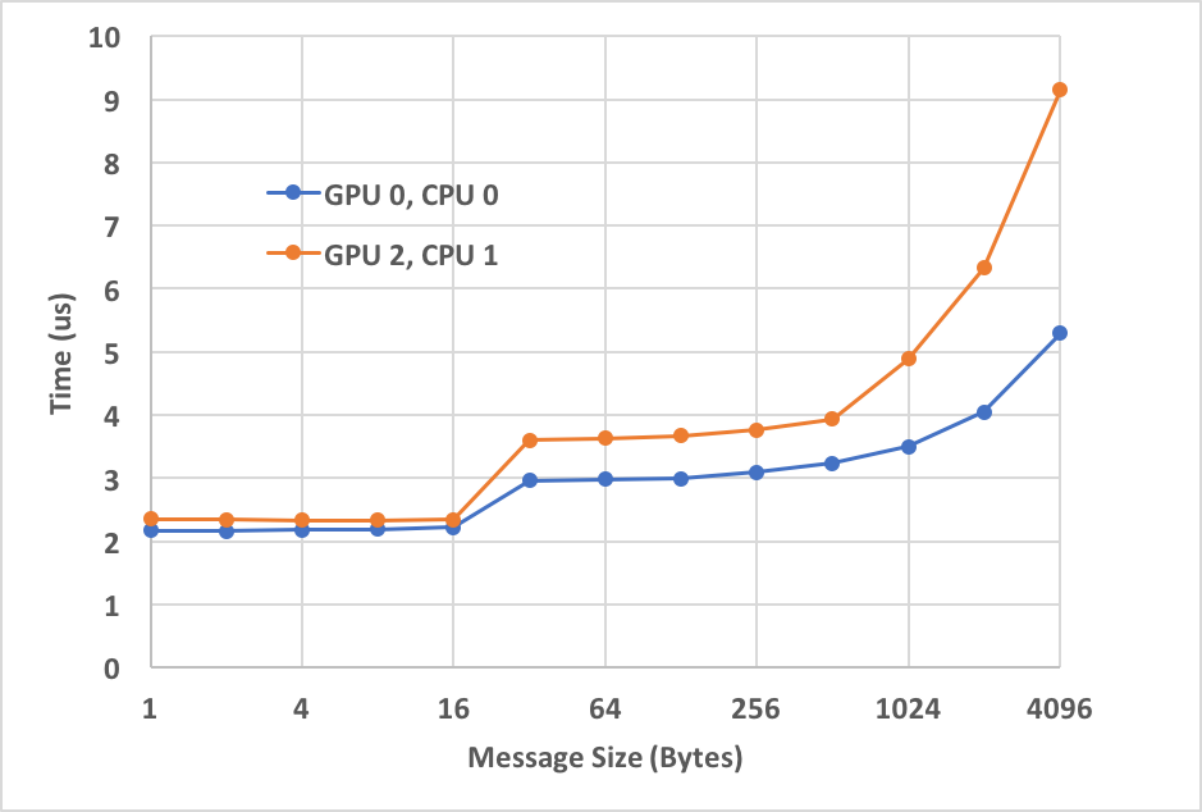
# OSU Latency (osu_latency) Benchmark Intra-node, P100 nodes




- Latency between GPU 0 , GPU 1: 2.73 $\mu$s
- Latency between GPU 2 , GPU 3: 2.95 $\mu$s
- Latency between GPU 1 , GPU 2: 3.13 $\mu$s

# OSU Latency (osu_latency) Benchmark Inter-node, P100 nodes




- Latency between GPU 0 , process bound to CPU 0 on both nodes: 2.17 $\mu$s
- Latency between GPU 2 , process bound to CPU 1 on both nodes: 2.35 $\mu$s

# Exercise

- Use MPI to do the Matrix-Vector Multiplication exercise from the OpenMP lecture (see snapshot below).
- Show parallel speedup and efficiency up to 2 nodes (48 tasks)

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.

# References

- **Excellent tutorials from LLNL:**
  - https://computing.llnl.gov/tutorials/mpi/
  - https://computing.llnl.gov/tutorials/openMP/
- **MPI for Python:**
  - http://mpi4py.scipy.org/docs/usrman/tutorial.html
- **MVAPICH2 User Guide:**
  - http://mvapich.cse.ohio-state.edu/userguide/