Message Passing Interfaces

A. Sahu
Dept of CSE, IIT Guwahati
Outline

• Introduction
  – Multicore System
  – HPC System
• Writing Parallel Program
• MPI Constructs and Example
• Running programming in IITG HPC system
• Reference and Other Resources
Trend of HPC

- HPC system
  - Multi Nodes/Computer/Blades
  - Programming Model MPI
Motivation for Parallel Programming

• Solving large problem (HPC)
  – Scientific simulation, computation, CFD, data analytics, ..
Programming Model

• Shared memory Programming Model
  – Pthread, Cilk, OpenMP, Vectorized

• Distributed Memory Programming Model
  – MPI
  – Large Scale
Writing Parallel Program using MPI
Writing Parallel Program

• Given a problem
• Design Solution/algorithm
• Design Solution with Parallel Algorithmic Technique
Generic Parallel Algorithm Design: Foster’s Methodology

• **Partitioning**: Process of dividing the computation and data into pieces
  – A good partitioning spilt both into many pieces
  – **Domain decomposition**: divide data into pieces and associate computation with data
Generic Parallel Algorithm Design: Foster’s Methodology

• **Partitioning**: Process of dividing the computation and data into pieces
  – A good partitioning splits both into many pieces
  – **Domain decomposition**: divide data into pieces and associate computation with data

• **Communication**: Identify communication pattern between partition and intra partition
Generic Parallel Algorithm Design: Foster’s Methodology

- **Partitioning**: Process of dividing the computation and data into pieces
  - A good partitioning spilt both into many pieces
  - **Domain decomposition**: divide data into pieces and associate computation with data

- **Communication**: Identify communication pattern between partition and intra partition

- **Agglomeration**: Process of grouping partition/task into larger task to reduce communications

- **Mapping**: Assigning task to processor
Message Passing Interface (MPI)
The Message-Passing Model

- A *process* is a program with an execution context (PC and address space).
- Processes may have multiple *threads* (PCs and associated stacks) sharing a single address space.
The Message-Passing Model

• A process is program with ex. with PC and address space.

• Processes may have multiple threads (PCs and associated stacks) sharing a single addr. space.

• MPI is for communication among processes, which have separate address spaces.

• Interprocess communication consists of
  – Synchronization
  – Movement of data from one process’s address space to another’s.
How to install MPI in Linux machine

$sudo$ apt-get install mpich2 mpich2-doc

Or

$sudo$ dnf install openmpi
How to install MPI in Linux machine

$ sudo apt-get install mpich2 mpich2-doc

Or

$ sudo dnf install openmpi

$ mpicc hello_mpi.c –o hello_mpi
How to install MPI in Linux machine

$sudo apt-get install mpich2 mpich2-doc
Or
$sudo dnf install openmpi

$mpicc hello_mpi.c –o hello_mpi
$mpirun –np 4 ./hello_mpi

4 copy of hello_mpi process will run
The Message-Passing Programming Paradigm

• Sequential Programming Paradigm

A processor may run many processes

- data
- memory
- Processor/Process
- program
The Message-Passing Programming Paradigm

- Message-Passing Programming Paradigm
The Message-Passing Programming Paradigm

- A **process** is a program performing a task on a **processor**
- Each processor/process in a message passing program runs a **instance/copy** of a **program**
- Written in a conventional sequential language, e.g., C or Fortran,
The Message-Passing Programming Paradigm

• Typically a single program operating on multiple dataset
• The variables of each sub-program have
  – The same name
  – But different locations (distributed memory) and different data!
  – i.e., all variables are local to a process
• Communicate via special send & receive routines (*message passing*)
Every process of MPI are different

• Hi : single person : you do
  – Touch your nose by left hand
  – Hi : Touch you head by right hand

• Hi: all persons of this hall do:
  – Touch your nose
Every process of MPI are different

• How to do work collaboratively: MPI program
• Assume 10 persons: want to do sum of n numbers
• First person: manager responsible for I/O
  – Get input from KBD
  – Send one data to each person
  – Get Sum from 2\textsuperscript{nd} person
  – Display the SUM
• All persons: every person have rank/ID-number
  – Receive a data from master
  – Receive a SUM from rank+1 person if \( i<10 \)
  – If rank=10 \( \text{SUM} = \text{Number} \) else \( \text{SUM} = \text{SUM} + \text{Number} \)
  – Send the number of rank-1 person.
main(){
    int D, SUM, rank, data[N]; //private data
    if (rank==MASTER){
        Get_inputs_from_KBD()
        Send_one_data_to_each_person();//SCATER();
        Get_Sum_(SUM, From_2ND_person); S=NUM+SUM;
        Display_the_SUM();
    } else {
        Receive_a_data_from_master(D, MASTER);
        if i<10
            Receive_a_SUM_from_rank+1_person
        If (rank==N) SUM = D; else SUM=SUM+D
        Send_the_number(SUM, RANK-1);
    }
}

Do work collaboratively : MPI program
Data and Work Distribution

• To communicate together mpi-processes need identifiers: \textbf{rank} = \textbf{identifying number}

• all distribution decisions are based on the \textit{rank} – i.e., which process works on which data
What is SPMD

• **Single** Program, **Multiple** Data
• Same (sub-)program runs on each processor
• MPI allows also MPMD, i.e., **Multiple** Program, ...
  – but some vendors may be restricted to SPMD
  – MPMD can be emulated with SPMD
Emulation of MPMD

main(int argc, char **argv){
    if (myrank < XX){
        ocean( /* arguments */ );
    }
    else{
        weather( /* arguments */ );
    }
}

Message passing

• Messages are packets of data moving between sub-programs
• Necessary information for the message passing system:
  – sending process – receiving process i.e., the ranks
  – source location – destination location
  – source data type – destination data type
  – source data size – destination buffer size
Access

• A sub-program needs to be connected to a message passing system

• A message passing system is similar to:
  – phone line, mail box, fax machine, etc.

• MPI:
  – program must be linked with an MPI library
  – program must be started with the MPI startup tool
What is message passing?

• Data transfer
• Requires cooperation of sender and receiver
• Cooperation not always apparent in code
Blocking vs. Non-Blocking

• Blocking
  – The program will not continue
  – Until the communication is completed.

• Non-Blocking
  – The program will continue
  – Without waiting for the communication to be completed.
Features of MPI

• General
  – Communications combine context and group for message security.
  – Thread safety can’t be assumed for MPI programs.
Features that are NOT part of MPI

- Process Management
- Remote memory transfer
- Threads
- Virtual shared memory
Why to use MPI?

• MPI provides a
  – Powerful, efficient, and portable way to express parallel programs.
Why to use MPI?

• MPI provides a
  – Powerful, efficient, and portable way to express parallel programs.

• MPI was explicitly designed
  – To enable libraries which eliminate the need for many users to learn much inside of MPI.
Why to use MPI?

• MPI provides a
  – Powerful, efficient, and portable way to express parallel programs.

• MPI was explicitly designed
  – To enable libraries which eliminate the need for many users to learn much inside of MPI.

• Portable !!!!!!!!!!!!!!!!!!!!!!!!!!!!!

• Good way to learn about subtle
  – Issues in parallel computing
How big is the MPI library?

• Huge (125 Functions).

• Basic (6 Functions).
How to install MPI in Linux Cluster

MPI Library

LAM : Local Area Multiprocessor

MPI CH: Argon National Laboratory
#include <mpi.h>

void main( int argc, char **argv )
{
    MPI_Init( &argc, &argv );

    /* main part of the program */

    /*
     * Use MPI function call depend on your data partitioning and the
     * parallelization architecture
     */

    MPI_Finalize();
}

Skeleton MPI Program
Initializing MPI

- The initialization routine MPI_INIT is the first MPI routine called.
- MPI_INIT is called once

```c
int mpi_Init(
    int *argc,
    char **argv );
```
A minimal MPI program

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

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    return 0;
}
```
How to compile and run on a Linux Machine

$mpiccc hello_mpi.c –o hello_mpi
$mpirun –np 4 ./hello_mpi

4 copies of hello_mpi process will run
A minimal MPI program cont.

• `#include <mpi.h>`
  – Provides basic MPI definitions and types.
• `MPI_Init` starts MPI
• `MPI_Finalize` exits MPI
• **Note that all non-MPI routines are local; thus “printf” run on each process**
• `MPI` functions return
  – Error codes or `MPI_SUCCESS`
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
MPI  Basic Communication Constructs
MPI Concepts

• The default communicator is the `MPI_COMM_WORLD`.

• A process is identified
  – By its rank in the group associated with a communicator.
Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
Finding Out About the Environment

• MPI provides functions to answer these questions:
  - `MPI_Comm_size` reports the number of processes.
  - `MPI_Comm_rank` reports the rank, a number between 0 and size-1, identifying the calling process.
**MPI: Data Types**

- The data message which is sent or received is described by a triple
  - Address, count, data type
- The following data types are supported
  - Predefined data types
  - Arrays, sub blocks of a matrix and user defined
MPI blocking send

MPI_Send(void *start, int count,
          MPI_DATATYPE datatype,
          int Dest, int Tag,
          MPI_COMM comm );

MPI blocking send

MPI_Send(void *start, int count, MPI_DATATYPE datatype, int Dest, int Tag, MPI_COMM comm);

• The message buffer is described by (start, count, datatype).
• Dest is the rank of the target process in the defined communicator
• Tag is the message identification number.
MPI blocking receive

MPI_Recv(void *start, int count,
           MPI_DATATYPE datatype,
           int Source, int Tag,
           MPI_COMM comm, MPI_COMM *S);
MPI blocking receive

MPI_Recv(void *start, int count, MPI_DATATYPE datatype, int Source, int Tag, MPI_COMM comm, MPI_COMM *S);

• **Source** is the rank of the sender in the communicator.

• The receiver can specify
  – A wildcard value for source (MPI_ANY_SOURCE)
  – A wildcard value for tag (MPI_ANY_TAG),
  – Indicating that any source and/or tag are acceptable
MPI blocking receive

MPI_Recv(void *start, int count, MPI_DATATYPE datatype, int Source, int Tag, MPI_COMM comm, MPI_COMM *S);

- **Status** is used for extra information
  - About the received message if a wildcard receive mode is used
- If the count of the message received is <= described by the MPI receive command
  - Message is successfully received
  - Else it is considered as a buffer overflow error.
Sources of Deadlocks

• Send a large message from proc 0 to proc 1
  – If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What happens with

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

• This is called “unsafe” because it depends on the availability of system buffers
Collective Communication

- Optimised Communication routines involving a group of processes
  - Collective action over a communicator
    - i.e. all processes must call the collective routine
  - Synchronization may or may not occur
  - All collective operations are blocking
    - and No tags
  - Receive buffers must have exactly the same size as send buffers.
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

- rank of the sending process (i.e., root process)
- must be given identically by all processes

e.g., root=1

before bcast

after bcast

red red red red red red
Broadcast

P1: A B C D
P2: 
P3: 
P4: 

MPI_Bcast

P1: A B C D
P2: A B C D
P3: A B C D
P4: A B C D
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
Scatter

MPI_Scatter

P1 | A  | B  | C  | D  |
---|----|----|----|----|
P2 |     |    |    |    |
P3 |     |    |    |    |
P4 |     |    |    |    |

P1 | A  | B  | C  | D  |
P2 | B  |    |    |    |
P3 | C  |    |    |    |
P4 | D  |    |    |    |
Gather

```c
int MPI_Gather(void *sendbuf,
               int sendcnt,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcnt,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

e.g., root=1
Gather

MPI_Gather

P1 P2 P3 P4
A B C D
## Gather to All

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Global Reduction Operations

- Perform a global reduce operation across all members of a group.

\[ d_0 \ o \ d_1 \ o \ d_2 \ o \ d_3 \ o \ ... \ o \ d_{s-2} \ o \ d_{s-1} \]

- \( d_i \) = data in process rank \( i \)
  - single variable, or vector
- \( o \) = associative operation
- Example:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation
MPI_Reduce

before MPI_REDUCE
- inbuf and result

A o o o o
• inbuf and result

root=1

after

A o Do Go Jo M

M N O
**MPI _Reduce**

<table>
<thead>
<tr>
<th>P1</th>
<th>A</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P3</td>
<td>C</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P4</td>
<td>D</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **P1**
  - A + B + C + D

- **P2**
  - A
  - B
  - C
  - D

- **P3**
  - A
  - B
  - C
  - D

- **P4**
  - A
  - B
  - C
  - D

---

MPI _Reduce

---
MPI_AllReduce

before MPI_ALLREDUCE
  • inbuf and result

A B C  D E F  G H I  J K L  M N O

A B C  D E F  G H I  J K L  M N O  A o D o G o J o M

after
MPI_AllReduce

P1  A  ...  ...  ...
P2  B  ...  ...  ...
P3  C  ...  ...  ...
P4  D  ...  ...  ...

P1  A+B+C+D  ...
P2  A+B+C+D  ...
P3  A+B+C+D  ...
P4  A+B+C+D  ...

MPI_AllReduce
MPI_Scan

before MPI_SCAN
• inbuf and result

done in parallel
MPI Examples
Example: Sum of N data

• Master Process
  – Data to be read by process 0 or MASTER
  – Divide the data in to N/M chunk size (N %M==0)
  – **SEND** respective chunk of data to other process
  – **Do local sum on each process (in master also)**
  – **RECV** sum of other process and calculate final sum

• Other Process
  – **RECV** data from Mater
  – **Do local sum on each process**
  – **SEND** local sum to MASTER
See the Code
Example: Matrix MUL

• c=axb: a[NRA][NCA], b[NCA][NCB], c[NRA][NCB]

• Work get divided: Based on Rows
Example: Matrix MUL

• c=axb: a[NRA][NCA], b[NCA][NCB], c[NRA][NCB]
• One Master Processor
• Many Workers, Assume NRA % NumWorker==0
  – Master divide the work between worker
  – Send respective rows of A and whole B to workers
  – RECV array C from all worker
• Every Worker
  – get some Row of A, Whole of B
  – calculate part of C
  – Send calculated C to Master
See the Code
Example: Compute $\pi$

$$\pi = \int_{0}^{1} \frac{4}{1 + x^2} \, dx$$
Example: Compute Pi

\[ \pi = \int_{0}^{1} \frac{4}{1 + x^2} \, dx \]
How to write Program?

• Divide the range in to N interval/piece
  – Piece of size $h = \text{Range}/N$;

• Calculate area under each piece
  – Calculate the function value at piece $X$ and multiply with piece size
  – $h \times F(X)$

• Sum all the piece
  – $\sum_{i=1}^{n} h \times F(X_i)$ with $X_i = \text{R}_{\text{min}} + i \times h$
How to write Program?

```c
printf("Enter Num intervals: ");
scanf("%d", &n);
h = 1.0 / (double)n;
sum = 0.0;
for (i=1; i<n; i++) {
    x = h*(i-0.5); Fx=4.0/(1.0+ x*x);
    sum = sum + Fx;
}
pi = h*sum;
printf("pi is approx %.16f", pi);
```
How to write Parallel Program?

• Divide the range into \( N \) interval/piece
  
  – Piece of size \( h = \text{Range}/N; \)
  
  – \textbf{Suppose} \( N = 1000, \ \text{NumProcessor} = 4 \)

• In Parallel: Calculate area under each piece

\[
\begin{align*}
  hF_{x1} + hF_{x2} + hF_{x3} + hF_{x4} + hF_{x5} + hF_{x6} + hF_{x7} + hF_{x8} + \ldots + hF_{x999}
\end{align*}
\]
How to write Parallel Program?

• Divide the range in to N interval/piece
  – Piece of size $h = \text{Range}/N$;
  – Suppose $N = 1000$, NumProcessor = 4

• In Parallel: Calculate area under each piece

\[
\begin{align*}
&hF_{x1} + hF_{x2} + hF_{x3} + hF_{x4} + hF_{x5} + hF_{x6} + hF_{x7} + hF_{x8} + \ldots + hF_{x999} \\
&+ (hF_{x1} + hF_{x5} + \ldots + hF_{x997}) + (hF_{x2} + hF_{x6} + \ldots + hF_{x998}) + \ldots + (hF_{x4} + hF_{x8} + \ldots + hF_{x996})
\end{align*}
\]
Example: Compute Pi

```c
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[]){
    int n, myid, Nproc, i;
    double lsum, pi, h, sum, x, a;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &Nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
        printf("Enter Num intervals: \n");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT,0,
               MPI_COMM_WORLD);
```
Example: Compute PI

```c
h = 1.0 / (double)n; sum = 0.0;
for (i=myid+1; i<=n; i+= Nproc) {
    x = h*((double)i – 0.5);
    sum += 4.0 / (1.0 + x * x);
}
lsum = h*sum;
MPI_Reduce(&lsum, &pi, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0)
    printf("pi is approx %.16f\n", pi);
MPI_Finalize();
return 0;
```

Domain Decomposition Example: SOR or Thermal model of grid

- Next temperature of the cell will depend on
  - Current temp of the cell
  - Effect of heat generation
  - Cooling rate
  - Effect of temp neighbor cell/core/grid
Domain Decomposition Example:
SOR or Thermal model of grid
Thermal Model of Grid/CMP

\[ T_{i,j}(t+1) = T_{i,j}(t) + \sum_{k=i-1,i+1\& l=j-1,j+1} \alpha(T_{k,l} - T_{i,j}) + \beta H_{i,j} + \gamma(T_A - T_{i,j}) \]
Thermal Model of Grid/CMP

\[ T_{i,j}(t+1) = T_{i,j}(t) + \sum_{k=i-1,i+1 \text{ and } l=j-1,j+1}^{\alpha} (T_{k,l} - T_{i,j}) + \beta H_{i,j} + \gamma (T_A - T_{i,j}) \]

- Current Temp of \((i,j)^{th}\) core/cell
- Heat generated at cell/core \((i,j)\)
- Ambient Temp \(T_A\)
- Temp. of all the neighbor cores/cell
\[ T_{i,j}(t+1) = T_{i,j}(t) + \sum_{k=i-1,i+1 \& l=j-1,j+1} \alpha (T_{k,l} - T_{i,j}) + \beta H_{i,j} + \gamma (T_A - T_{i,j}) \]

- \( T_{i,j} \): Current Temp of \((i,j)\)th core/cell
- \( T_A \): Ambient Temp
- \( H_{i,j} \): Heat generated at cell/core \((i,j)\)
- \( \alpha \): neighbor effect
- \( \gamma \): relative cooling rate
- \( \beta \): temp rise coefficient of heat generation
Thermal Model of Grid/CMP

\[ T_{i,j}(t+1) = T_{i,j}(t) + \sum_{k=i-1,i+1;l=j-1,j+1} \alpha (T_{k,l} - T_{i,j}) + \beta H_{i,j} + \gamma (T_{A} - T_{i,j}) \]

- \( T_{i,j}(t+1) \): Current Temp of \((i,j)^{th}\) core/cell
- \( T_{i,j}(t) \): Temp. of all the neighbor cores/cell
- \( \alpha \): neighbor effect
- \( \gamma \): relative cooling rate
- \( \beta \): temp rise coefficient of heat generation
- \( H_{i,j} \): Heat generated at cell/core \((i,j)\)
- \( T_{A} \): Ambient Temp
- \( T_{i,j} \): Temp. of all the neighbor cores/cell
 Thermal Model of MxN Grid

```c
float T[M][N]; TO[M][N];
for (time<MaxSimTime){
    for (i=1;i<M-1;i++)
        for (j=1;j<N-1;j++){
            C1= alpha*(TO[i-1][j]+TO[i+1][j] +TO[i][j-1]+TO[i][j+1]);
            C2= beta*H[i][j]
            C3= gamma(TO[i][j]-TA);
            T[i][j] = C1+ C2 + C3;
        }
    for (i=0;i<M;i++)
        for (j=0;j<N;j++) TO[i][j]=T[i][j];
    time++;
}
```
Parallel Version of Thermal Simulation

- **P0** requires Col 4 (from P1)
- **P1** requires Col 3 (from P0) and Col 8 (from P2)
- **P2** requires Col 7 (from P1) and Col 12 (from P3)
- **P3** requires Col 11 (from P3)
Parallel Version of Thermal Simulation

P0 P1 P2 P3

P0 -> P1
P1 -> P2
P2 -> P3
P3 -> P0
Parallel Version of Thermal Simulation

```c
if (left != NONE) {
    MPI_Send(ColumData to the left proc);
    source = left;  msgtype = LTAG;
    MPI_Recv(ColumData from the left proc);
}

if (right != NONE) {
    MPI_Send(ColumData to the right proc);
    source = right;  msgtype = RTAG;
    MPI_Recv(ColumData from the right proc);
}
```
Parallel Version of Thermal Simulation

Complete code: See demos
IITG HPC clusters: Spec

- 4 login nodes
- 126 compute node
- 16 GPU compute nodes
- 16 Phi compute nodes
- Total 126+16+16= 158 nodes
  - Each node 12 cores * 2 threaded
  - Effective 24*158 =3792 cores
Running MPI program on IITG HPC clusters

• Logic to one login nodes: non GPU/PHI
  – param.-ishan.iitg.ernet.in (172.17.0.7)

• Compile MPI-code
Running MPI program on IITG HPC clusters

• Logic to one login nodes : non GPU/PHI
  – param.-ishan.iitg.ernet.in (172.17.0.7)

• Compile MPI-code

• Run using srun or sbatch
  – In s batch specify number of node, task per node
  – Total process

• SLURM : Simple Linux Util for Resce Mngt
  – Scheduler the JOB efficiently, user need not to worry where it is scheduling
Resources

- [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)
  - [metis software]