Introduction to Code Parallelization Using MPI

Ping Luo
TAMU HPRC

March 29, 2019

HPRC Short Course – Spring 2019
Outline

- Parallel programming models
  - OpenMP for Shared Memory System
  - MPI for Distributed Memory System
  - MPI+OpenMP for hybrid systems
- Layout of an MPI program
- Compiling and running an MPI program
- Basic MPI concepts
  - size, rank, communicator, message, MPI datatype, tag
- Point-to-point communication
- Collective communication
Parallel Computing Systems

We use a processor or a CPU core to refer to the smallest physical processing unit where a program is executed.

- Shared Memory Systems
- Distributed Memory Systems
- Hybrid Systems
Parallel Computing Systems

- **Shared Memory System** – an abstraction to a parallel system where all processors share the same memory subsystem

Example: a node on Ada – an IBM NextScale nx360 M4 dual socket server
Parallel Computing Systems

- **Distributed Memory System** – an abstraction to a parallel system where each processor has its own local memory and the processors don’t share a global memory subsystem.
Parallel Computing Systems

- Hybrid System

Node

Interconnect network
# Parallel Programming Models

- Mapping from the parallel programming models to the parallel computing systems

<table>
<thead>
<tr>
<th>Model</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>Shared Memory System</td>
</tr>
<tr>
<td>MPI</td>
<td>Distributed Memory System</td>
</tr>
<tr>
<td>MPI+OpenMP</td>
<td>Hybrid System</td>
</tr>
</tbody>
</table>
Ada and Terra

- Ada has 852 nodes and each node has 20 cores.
- Terra has 304 compute nodes and each node has 28 cores.
- OpenMP can only run within a cluster node.
- MPI and Hybrid can run on multiple nodes.
Resources

- Two books: Using MPI and Using MPI2


- List of all MPI routines: https://www.mpich.org/static/docs/v3.2/

- Examples for the course are on Ada: /general/public/training/mpi/Spring2019
Example 1: Hello World

```c
#include <stdlib.h>

int main(int argc, char **argv){
    printf("Hello, world\n");
}
```

```fortran
program hello
implicit none

print *, "Hello, world"
end program hello
```
Example 1: Hello World

**C**

```c
#include <stdlib.h>

int main(int argc, char **argv){
    printf("Hello, world\n");
}
```

**Fortran**

```fortran
program hello
implicit none
print *, "Hello, world"
end program hello
```

**C**

```c
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    printf("Hello, world\n");
    MPI_Finalize();
}
```

**Fortran**

```fortran
program hello
use mpi
implicit none
print *, "Hello, world"
call MPI_Finalize(ierr)
end program hello
```
# Layout of an MPI Program

**C**

```c
#include <mpi.h>
int main(
    int argc, char **argv)
{
    ... no mpi calls
    MPI_Init(&argc, &argv);
    mpi calls happen here
    ... no mpi calls
    MPI_Finalize();
    ... no mpi calls
}
```

**Fortran**

```fortran
PROGRAM SAMPLE1
USE MPI !F90
!f77: include "mpif.h"
integer ierr
... no mpi calls
CALL MPI_INIT(ierr)
... mpi calls happen here
     multiple concurrent processes execute at their own pace unless synchronization is applied.
CALL MPI_FINALIZE(ierr)
... no mpi calls
END PROGRAM SAMPLE1
```

The mpi calls happen here, and mpi calls happen here.
Compiling and Linking MPI Programs

module load intel/2017b

mpicc  prog.c  [flags]  -o prog.exe  (C)
mpicpc prog.cpp [flags]  -o prog.exe  (C++)
mpiifort prog.f  [flags]  -o prog.exe  (Fortran)

(Intel compilers)

mpicc  prog.c  [flags]  -o prog.exe  (C)
mpicxx prog.cpp [flags]  -o prog.exe  (C++)
mpif90 prog.f  [flags]  -o prog.exe  (Fortran)

(GNU compilers)

We will use the Intel MPI and the Intel compilers in ensuing examples. See HPRC user guide for more information: https://hprc.tamu.edu/wiki/Ada:Compile:All#MPI_Programs
Running an MPI Program on Login Nodes

- Load the modules first

```
module load intel/2017b
```

- Run the mpi program on the login nodes for testing and debugging. **No more than 8 MPI tasks** can be launched at once per our policy.

```
mpirun -np n [options] \ prog.exe [prog_args]
(n is number of MPI tasks and n<=8)

-ppn/-perhost, -hosts, -hostfile, -h
```

- Useful MPI options

- When testing the code on the login nodes, make sure the code works on multiple login nodes.

```
mpirun -np 4 -hosts \ login1,login2,login3,\ login4 -ppn 1 mympi.exe
```
Running an MPI Program in Batch

- **Batch Examples**

<table>
<thead>
<tr>
<th>Ada</th>
<th>Terra</th>
</tr>
</thead>
<tbody>
<tr>
<td>#BSUB -J MPIBatchExample</td>
<td>#!/bin/bash</td>
</tr>
<tr>
<td>#BSUB -L /bin/bash</td>
<td>#SBATCH --export=NONE</td>
</tr>
<tr>
<td>#BSUB -W 24:00</td>
<td>#SBATCH --get-user-env=L</td>
</tr>
<tr>
<td>#BSUB -n 40</td>
<td>#SBATCH --job-name=MPIBatchExample</td>
</tr>
<tr>
<td>#BSUB -R &quot;span[ptile=20]&quot;</td>
<td>#SBATCH --time=24:00:00</td>
</tr>
<tr>
<td>#BSUB -R &quot;rusage[mem=2560]&quot;</td>
<td>#SBATCH --ntasks=56</td>
</tr>
<tr>
<td>#BSUB -M 2560</td>
<td>#SBATCH --ntasks-per-node=28</td>
</tr>
<tr>
<td>#BSUB -o MPIBatchExample.%J</td>
<td>#SBATCH --mem=56000M</td>
</tr>
<tr>
<td></td>
<td>#SBATCH --output=MPIBatchExample.%j</td>
</tr>
<tr>
<td>module load intel/2017b</td>
<td>module load intel/2017b</td>
</tr>
<tr>
<td>mpirun prog.exe</td>
<td>mpirun prog.exe</td>
</tr>
</tbody>
</table>

bsub < mpibatch.job

sbatch mpibatch.job
What is MPI

- Message Passing Interface: a specification for the library interface that implements message passing in parallel programing.

- Is standardized by the MPI forum for implementing portable, flexible, and reliable codes for distributed memory systems, regardless of the underneath architecture.
  - MPI-4.0 is under discussion

- Has C/C++/Fortran bindings.
  - C++ binding deprecated since MPI-2.2

- Different implementations (libraries) available: Intel MPI, MPICH, OpenMPI, etc.

- It is the most widely used parallel programing paradigm for large scale scientific computing.
Basic MPI Concepts

Communicator, size, rank

- **Communicator**: MPI_COMM_WORLD
  - Size: 8
  - Rank: 0, 1, ..., 7

- **Communicator**: comm1
  - Size: 2
  - Rank: 0, 1

- **Communicator**: comm2
  - Size: 4
  - Rank: 0, 1, 2, 3

- **Communicator**: comm3
  - Size: 3
  - Rank: 0, 1, 2

All MPI communication must specify a communicator.
Basic MPI Concepts

message, point-to-point communication, collective communication
### Example 2 – One Sender and One Receiver

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;stdlib.h&gt;</td>
<td>program simple</td>
</tr>
<tr>
<td>#include &lt;mpi.h&gt;</td>
<td>use mpi</td>
</tr>
<tr>
<td>int main(int argc, char **argv){</td>
<td>implicit none</td>
</tr>
<tr>
<td>int np, rank, number;</td>
<td>integer ierr, np, rank, number, status ;</td>
</tr>
<tr>
<td>MPI_Status status;</td>
<td>call MPI_INIT(ierr)</td>
</tr>
<tr>
<td>MPI_Init(&amp;argc, &amp;argv);</td>
<td>call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)</td>
</tr>
<tr>
<td>MPI_Comm_size(MPI_COMM_WORLD, &amp;np);</td>
<td>call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)</td>
</tr>
<tr>
<td>MPI_Comm_rank(MPI_COMM_WORLD, &amp;rank);</td>
<td>if (rank == 0) then</td>
</tr>
<tr>
<td></td>
<td>number = 1234</td>
</tr>
<tr>
<td></td>
<td>call MPI_SEND(number, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)</td>
</tr>
<tr>
<td></td>
<td>print *, &quot;process &quot;, rank, &quot; sends &quot;, number</td>
</tr>
<tr>
<td></td>
<td>}else if(rank == 1){</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv(&amp;number, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, status, ierr)</td>
</tr>
<tr>
<td></td>
<td>print *, &quot;process &quot;, rank, &quot; receives &quot;, number</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>MPI_Finalize();</td>
</tr>
<tr>
<td></td>
<td>end program simple</td>
</tr>
</tbody>
</table>
Communicator

- In MPI, a communicator is a software structure through which we specify a group of processes.

- Each process in a communicator is assigned a unique rank (an integer) ranging from 0 to (group_size - 1). group_size is the size of the communicator.

- The constant MPI_COMM_WORLD (obtained from MPI include file) is an initial communicator that includes all the MPI processes activated in the running of a program. MPI_COMM_WORLD is typically the most used communicator.

- Communicators are especially useful in characterizing the tasks that different groups of processes carry out.
Size and Rank

- How many processes in a communicator?

<table>
<thead>
<tr>
<th>C</th>
<th>int  MPI_Comm_size(MPI_Comm comm, int *size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SUBROUTINE MPI_COMM_SIZE(comm, size, ierr)</td>
</tr>
<tr>
<td></td>
<td>integer comm, size, ierr</td>
</tr>
</tbody>
</table>

- What’s the rank (identity) of each process in a communicator?

<table>
<thead>
<tr>
<th>C</th>
<th>int  MPI_Comm_rank(MPI_Comm comm, int *rank)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SUBROUTINE MPI_COMM_RANK(comm, rank, ierr)</td>
</tr>
<tr>
<td></td>
<td>integer comm, rank, ierr</td>
</tr>
</tbody>
</table>
Tag is an integer used in a message to differentiate one message from other messages.

Where to fetch/store the data

Type of the data to be sent or received. Datatype can be predefined or user defined. Commonly used predefined datatypes, also called MPI basic datatypes in Fortran:

- MPI_INTEGER
- MPI_REAL
- MPI_REAL8
- MPI_CHARACTER
- MPI_LOGICAL
program simple
use mpi
implicit none
integer ierr, np, rank, number, status ;

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

if (rank == 0) then
  number = 1234
  call MPI_SEND(number, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
  print *, "process ", rank, " sends ", number
else if (rank == 1) then
  call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
  print *, "process ", rank, " receives ", number
endif

call MPI_Finalize(ierr)
end program simpleC

Send and Receive a Message

Data

MPI_SEND(number, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)

MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)

Envelope

source

tag

communicator

data

Buffer

starting

address

MPI datatype

MPI destination

Data

Envelope

if (rank == 0) then
  number = 1234
  call MPI_SEND(number, 1, MPI_INTEGER 1, 0, MPI_COMM_WORLD, ierr)
  print *, "process ", rank, " sends ", number
else if (rank == 1) then
  call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
  print *, "process ", rank, " receives ", number
endif

call MPI_Finalize(ierr)
end program simpleC
Point-to-Point Communication

- Blocking
- Non-blocking
- Send-Receive

- `MPI_Send`, `MPI_Recv`
- `MPI_Isend`, `MPI_Irecv`
- `MPI_Sendrecv`
## Blocking Send

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td><code>int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)</code></td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td><code>MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;type&gt; buf(*)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>integer count, datatype, dest, tag, comm, ierr</code></td>
<td></td>
</tr>
</tbody>
</table>

- **buf**: initial address of send buffer
- **count**: number of elements in send buffer
- **datatype**: datatype of each send buffer element
- **dest**: rank of destination
- **tag**: message tag
- **comm**: communicator
Comments on Blocking Send

MPI_SEND(buf, count, datatype, dest, tag, comm)

- The calling process causes count many contiguous elements of type datatype to be sent, starting from buf.
- The message sent by MPI_SEND can be received by either MPI_RECV or MPI_IRECV.
- MPI_SEND doesn’t return (i.e., blocked) until it is safe to write to the send buffer.
  - Safe means the message has been copied either into a system buffer, or into the receiver’s buffer, depending on which mode the send call is currently working under.
## Blocking Receive

### C

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Status *status)
```

### Fortran

```fortran
MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)
```

C

- `buf` (*type*)
- `count`
- `datatype`
- `source`
- `tag`
- `comm`
- `status`
- `ierr`

### Parameters

- **buf**: initial address of receive buffer
- **count**: number of elements in receive buffer
- **datatype**: datatype of each receive buffer element
- **source**: rank of source or `MPI_ANY_SOURCE`
- **tag**: message tag or `MPI_ANY_TAG`
- **comm**: communicator
- **status**: status object

*Note: `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are MPI defined wildcards.*
Comments on Blocking Receive

MPI_RECV(buf,count,datatype,source,tag,comm,status)

- The calling process attempts to receive a message with specified envelope (source, tag, communicator).
  - MPI_ANY_SOURCE and MPI_ANY_TAG are valid values.
- When the matching message arrives, elements of the specified datatype are placed in the buffer in contiguous locations, starting at the address of buf.
- The buffer starting at buf is assumed pre-allocated and has capacity for at least count many datatype elements.
  - An error returns if buf is smaller than data received.
Comments on Blocking Receive

**MPI_RECV***(buf, count, datatype, source, tag, comm, status)***

- **MPI_RECV** can receive a message send by **MPI_SEND** or **MPI_ISEND**.
- Agreement in **datatype** between the send and receive is required.
- **MPI_RECV** is **blocked** until the message has been copied into **buf**.
- The actual size of the message received can be extracted with **MPI_GET_COUNT**.
Return Status

- The argument `status` in `MPI_Recev` provides a way of retrieving message source, message tag, and message error from the message.

- `status` is useful when MPI wildcards (MPI_ANY_SOURCE, MPI_ANY_TAG) are used in `MPI_Recev`.

- `status` can be ignored with `MPI_STATUS_IGNORE`

### Return Status

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
</table>
| C: ```
MPI_Status status
...
MPI_Recv(...,&status)
Source_id = status(MPI_SOURCE)
tag = status(MPI_TAG)
``` | Fortran: ```
integer status(MPI_STATUS_SIZE)
...
CALL MPI_RECV(...,status,ierr)
source_id = status(MPI_SOURCE)
tag = status(MPI_TAG)
``` |
Example 3

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

int main(int argc, char **argv) {
    int number, size, rank;
    int i;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2) {
        MPI_Abort(MPI_COMM_WORLD, 99);
    }
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        printf("Type any number from the input: ");
        scanf("%d", &number);
        for (i=1; i<size; i++)
            MPI_Send(&number, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("My id is %d. I received %d\n", rank, number);
    }
    MPI_Finalize();
}
```

program ex3
use mpi
implicit none
integer rank, np, ierr, number, i

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

if (np < 2) then
    call MPI_ABORT(MPI_COMM_WORLD, 99, ierr)
endif

if (rank == 0) then
    print *, "Type an integer from the input"
    read *, number
    do i=1, np-1
        call MPI_SEND(number, 1, MPI_INTEGER, i, 0, MPI_COMM_WORLD, ierr)
    enddo
else
    call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, &
                   MPI_STATUS_IGNORE, ierr)
    print "(2(A,I6))", "Process ", rank, " received ", number
Endif
call MPI_FINALIZE(ierr)
end program ex3

• One sender and multiple receivers
• The sender sends out a number to each and every receiver

• One sender and multiple receivers
• The sender sends out a number to each and every receiver
Non-blocking Send

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>buf</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
</tr>
<tr>
<td>IN</td>
<td>dest</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
</tr>
</tbody>
</table>
Non-blocking Receive

MPI_IRecv(buf, count, datatype, source, tag, comm, request)

<table>
<thead>
<tr>
<th>IN</th>
<th>buf</th>
<th>initial address of send buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in receive buffer (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each receive buffer element</td>
</tr>
<tr>
<td>IN</td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator</td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
<td>communication request (a handle that can be used later to refer the outstanding receive)</td>
</tr>
</tbody>
</table>
Non-blocking Send/Receive

- A non-blocking send/receive call initiates the send/receive operation, and returns immediately with a request handle, before the message is copied out/into the send/receive buffer.

- A separate send/receive complete call is needed to complete the communication before the buffer can be accessed again.

- A non-blocking send can be matched by a blocking receive; a non-blocking receive can be matched by a blocking send.

- Used correctly, non-blocking send/receive can improve program performance.

- They also make the point-to-point transfers “safer” by not depending on the size of the system buffers.
  - No deadlock caused by unavailable buffer
  - No buffer overflow
Auxiliary Routines for Non-blocking Send/Receive

- Auxiliary routines are used to complete a non-blocking communication or communications.
- Commonly used auxiliary routines:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Wait(request, status)</td>
<td>The calling process waits for the completion of a non-blocking send/receive identified by request.</td>
</tr>
<tr>
<td>MPI_Waitall(count, requests, statuses)</td>
<td>The calling process waits for all pending operations in a list of requests.</td>
</tr>
<tr>
<td>MPI_Test(request, flag, status)</td>
<td>The calling process tests a non-blocking send/receive specified by request has completed delivery/receipt of a message.</td>
</tr>
</tbody>
</table>
### MPI_WAIT

**MPI_WAIT(request, status)**

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>request request;</td>
<td>integer request</td>
</tr>
<tr>
<td>MPI_Status status;</td>
<td>integer status(MPI_STATUS_SIZE)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MPI_Irecv(recv_buf,count,...,comm,&amp;request);</td>
<td>call MPI_Irecv(recv_buf,count,...&amp;comm,request,ierr)</td>
</tr>
<tr>
<td>...do some computations ...</td>
<td>... do some computations ...</td>
</tr>
<tr>
<td>MPI_Wait(&amp;request, &amp;status);</td>
<td>call MPI_WAIT(request,status(ierr)</td>
</tr>
</tbody>
</table>

- **request**  
- **status**  

**status** can be ignored with MPI_STATUS_IGNORE
MPI_WAITALL

MPI_WAITALL(count, requests, statuses)

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer reqs(4)</td>
<td>MPI_Request reqs[4];</td>
</tr>
<tr>
<td>integer statuses(MPI_STATUS_SIZE,4)</td>
<td>MPI_Status status[4];</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>call MPI_ISEND(...,reqs(1),ierr)</td>
<td>call MPI_Isend(...,&amp;reqs[0]);</td>
</tr>
<tr>
<td>call MPI_Irecv(...,reqs(2),ierr)</td>
<td>call MPI_Irecv(...,&amp;reqs[1]);</td>
</tr>
<tr>
<td>call MPI_ISEND(...,reqs(3),ierr)</td>
<td>call MPI_Isend(...,&amp;reqs[2]);</td>
</tr>
<tr>
<td>call MPI_Irecv(...,reqs(4),ierr)</td>
<td>call MPI_Irecv(...,&amp;reqs[3]);</td>
</tr>
<tr>
<td>... do some computations ...</td>
<td>... do some com computations ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>call</td>
<td>call</td>
</tr>
<tr>
<td>MPI_WAITALL(4,reqs,statuses,ierr)</td>
<td>MPI_Waitall(4,reqs,statuses);</td>
</tr>
</tbody>
</table>

`statuses` can be ignored with MPI_STATUSES_IGNORE
### Example 4

**C**

```c
MPI_Request *requests;

if (rank == 0) {
    printf("Type any number from the input: ");
    scanf("%d", &number);
    requests = (MPI_Request *)(malloc(npof(MPI_Request)*(np-1)));

    for (i=1; i<np; i++)
        MPI_Isend(&number, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &requests[i-1]);

    MPI_Waitall(np-1, requests, MPI_STATUSES_IGNORE);
    free(requests);
} else {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("My id is %d. I received %d \n", rank, number);
}
```

**Fortran**

```fortran
integer, allocatable::requests(:)

if (rank == 0) then
    print *, "Type an integer from the input"
    read *, number
    allocate(requests(np-1))
    do i=1, np-1
        call MPI_ISEND(number, 1, MPI_INTEGER, i, 0, &
                        MPI_COMM_WORLD, ierr)
    enddo
    call MPI_WAITALL(np-1, requests, MPI_STATUSES_IGNORE, ierr)
    deallocate(requests)
else
    call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, &
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
    print "(2(A,I6))","Process ",rank, " received ", number
endif
```

- One sender and multiple receivers
- The sender sends out a number to each and every receiver
- The sender uses non-blocking send
Send-Receive

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

- Combines send and receive operations in one call
- The source and destination can be the same.
- The message sent out by send-receive can be received by blocking/non-blocking receive or another send-receive
- It can receive a message sent by blocking/non-blocking send or another send-receive.
- Useful for executing a **shift operation** across a chain of processes.
- Dependencies will be taken care of by the communication subsystem to eliminate the possibility of deadlock.
Collective Communication

- A collective communication refers to a communication that involves all processes in a communicator.
# Routines for Collective Communication

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BARRIER</td>
<td>All processes within a communicator will be blocked until all processes within the communicator have entered the call.</td>
</tr>
<tr>
<td>MPI_BCAST</td>
<td>Broadcasts a message from one process to members in a communicator.</td>
</tr>
<tr>
<td>MPI_REDUCE</td>
<td>Performs a reduction operation to the vector of elements in the sendbuf of the group members and places the result in recvbuf on root.</td>
</tr>
<tr>
<td>MPI_GATHER</td>
<td>Collects data from the sendbuf of all processes in comm and place them consecutively to the recvbuf on root based on their process rank.</td>
</tr>
<tr>
<td>MPI_GATHERV</td>
<td>Collects data from the sendbuf of all processes in comm and place them consecutively to the recvbuf on root based on their process rank.</td>
</tr>
<tr>
<td>MPI_SCATTER</td>
<td>Distribute data in sendbuf on root to recvbuf on all processes in comm.</td>
</tr>
<tr>
<td>MPI_SCATTERV</td>
<td>Distribute data in sendbuf on root to recvbuf on all processes in comm.</td>
</tr>
<tr>
<td>MPI_ALLREDUCE</td>
<td>Same as MPI_REDUCE, except the result is placed in recvbuf on all members in a communicator.</td>
</tr>
<tr>
<td>MPI_ALLGATHER</td>
<td>Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.</td>
</tr>
<tr>
<td>MPI_ALLGATHERV</td>
<td>Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.</td>
</tr>
<tr>
<td>MPI_ALLTOALL</td>
<td>The j-th block of the sendbuf at process i is send to process j and placed in the i-th block of the recvbuf of process j.</td>
</tr>
</tbody>
</table>
**MPI_BARRIER**

MPI_BARRIER(comm)

- Blocks all processes in `comm` until all processes have called it.
- Is used to synchronize the progress of all processes in `comm`. 
MPI_BCAST

MPI_BCAST(buffer, count, datatype, root, comm)

- Root process: sends a message to all processes (including root) in the communicator `comm`.
- Non-root processes: receives a message from the specified `root`.
- Each receiving process blocks until the message has arrived its `buffer`.
- All processes in `comm` must call this routine.
MPI_REDUCE

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

- Performs a reduction operation on all elements with same index in sendbuf on all processes and stores results in recvbuf of the root process.
- recvbuf is significant only at root.
- sendbuf and recvbuf cannot be the same.
- The size of sendbuf and recvbuf is equal to count.

C: MPI_Op op
Fortran: integer op

```
Fortran:
integer op
```

```
C:
MPI_Op op
```
## Predefined Reduction Operations

<table>
<thead>
<tr>
<th>PREDEFINED OPERATIONS</th>
<th>MPI DATATYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM, MPI_PROD</td>
<td>MPI_REAL8, MPI_INTEGER,</td>
</tr>
<tr>
<td></td>
<td>MPI_COMPLEX, MPI_DOUBLE,</td>
</tr>
<tr>
<td></td>
<td>MPI_INT, MPI_SHORT,</td>
</tr>
<tr>
<td></td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>MPI_MIN, MPI_MAX</td>
<td>MPI_INTEGER, MPI_REAL8,</td>
</tr>
<tr>
<td></td>
<td>MPI_INT, MPI_SHORT,</td>
</tr>
<tr>
<td></td>
<td>MPI_LONG, MPI_DOUBLE</td>
</tr>
<tr>
<td>MPI_LAND, MPI_LOR,</td>
<td>MPI_LOGICAL, MPI_INT,</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>MPI_SHORT, MPI_LONG</td>
</tr>
<tr>
<td>MPI_BAND, MPI_BOR,</td>
<td>MPI_INTEGER, MPI_INT,</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>MPI_SHORT, MPI_LONG</td>
</tr>
</tbody>
</table>
### MPI_ALLREDUCE

**MPI_ALLREDUCE**(*sendbuf, recvbuf, count, datatype, op, comm*)

![Diagram showing the operation of MPI_ALLREDUCE]

- **p0**
  - 1.2345
  - 3.1257
  - 9.2134

- **p1**
  - 2.4345
  - 7.0321
  - -0.9234

- **p2**
  - -0.2374
  - 3.2478
  - 0.2319

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
```
MPI_GATHER

MPI_GATHER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)

- Gathers together data from all process in comm and stores in root process.
- Data received by root are stored in rank order.
- recvcnt is number of elements received per process
-Recvbuf, recvcnt, recvtype are significant only at root.
MPI_GATHERV

MPI_GATHERV(sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm)

IN recvcnts an integer array of size of comm. recvcnts[i] = number of elements received from process i.

IN displs an integer array of size of comm. displs[i] = displacement from recvbuf for process i.

Fortran
integer recvcnts(*), displs(*)

C
int recvcnts[], displs[]
MPI_GATHERV (cont.)

MPI_GATHERV(sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm)
MPI_GATHERV (cont.)

MPI_GATHERV(sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm)

C:
Fortran:
MPI_GATHERV (cont.)

MPI_GATHERV(sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm)

C:
Fortran:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

displs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

recvbuf

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2345</td>
<td>2.4345</td>
<td>3.2478</td>
</tr>
<tr>
<td>3.1257</td>
<td>7.0321</td>
<td></td>
</tr>
<tr>
<td>-0.2374</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sendbuf

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2345</td>
<td>2.4345</td>
<td>3.2478</td>
</tr>
<tr>
<td>3.1257</td>
<td>7.0321</td>
<td></td>
</tr>
<tr>
<td>-0.2374</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P0 (root)

sendbuf

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2345</td>
<td>3.1257</td>
</tr>
</tbody>
</table>

recvbuf

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

P1

sendbuf

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4345</td>
<td>7.0321</td>
</tr>
<tr>
<td>3.2478</td>
<td></td>
</tr>
</tbody>
</table>

recvbuf

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

P2

sendbuf

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2478</td>
</tr>
</tbody>
</table>

recvbuf

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

C

Fortran:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
MPI_ALLGATHER/MPI_ALLGATHERV

MPI_ALLGATHER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, comm)

MPI_ALLGATHERV(sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, comm)

- Same as MPI_GATHER/MPI_GATHERV, except no root.
- Root is not needed since every process in the communicator stores the data gathered in its recvbuff.
MPI_SCATTER

MPI_SCATTER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)
MPI_SCATTERV

MPI_SCATTERV(sendbuf, sendcmts, displs, sendtype, recvbuf, recvcnt, recvtype, root, comm)

p0

1.2345
2.4345
3.2478

3.1257
2.4345
7.0321

-0.2374
-0.2374
-0.2374

p1

1.2345
2.4345
3.2478

3.1257
2.4345
7.0321

-0.2374
-0.2374
-0.2374

p2

1.2345
2.4345
3.2478

3.1257
2.4345
7.0321

-0.2374
-0.2374
-0.2374

C:
0 2 5
1 3 6

Fortran:
0 2 5
1 3 6
Timing Routine

MPI_WTIME()

- returns a floating-point number in seconds, representing elapsed wall clock time since some time in the past.

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>double t1, t2; double elapsed; t1 = MPI_Wtime(); ... // code segment to be timed ... t2 = MPI_Wtime(); elapsed = t2 - t1;</td>
<td>real<em>8 t1, t2 real</em>8 elapsed t1 = MPI_WTIME() ... ! Code segment to be timed ... t2 = MPI_WTIME() elapsed = t2 - t1</td>
</tr>
</tbody>
</table>
Example 5: Calculate PI

\[ \pi = \int_0^1 \frac{4.0}{1 + x^2} \, dx \]
Break
Outline for MPI Part II

- Exploring parallelism
  - Task-parallelism
  - Data-parallelism
- Matrix-vector multiplication
- Solving the 2D Poisson equation
- Domain decomposition
- MPI/OMP hybrid programming
- MPI/OMP: A Case Study
Important Note On Using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs
Exploring Parallelism

- Task-parallelism: The programmer identifies different tasks of a program and distribute the tasks among different processors.
- Data-parallelism: The programmer partitions the data used in a program and distribute them among different processors, each performing similar operations on the subset of data assigned.
3 chefs need to prepare a three-course menu for 12 guests

Preparing 12 salads
Preparing 12 steaks
Preparing 12 deserts

Task parallelism

Preparing 12 salads
Preparing 12 steaks
Preparing 12 deserts

Data parallelism

Task 1
Task 2
Task 3

4 meals
salad
steak
desert
salad
steak
desert
4 meals
salad
steak
desert
salad
steak
desert
4 meals
salad
steak
desert
salad
steak
desert
Example 6: Data Distribution

Cyclic Distribution

Data is distributed in a round robin manner among the processes.

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=myid+1; i&lt;N; i+=nprocs) { x = h*(i-0.5); sum += 4.0/(1.0+x<em>x); } sum = sum</em>h;</td>
<td>do i=myid+1, N, nprocs x = h*(i-0.5d0) sum = sum+4.0d0/(1.0d0+x<em>x) enddo sum = sum</em>h;</td>
</tr>
</tbody>
</table>

calc_PI_cyclic.c
Example 6: Data Distribution

Data is partitioned into $n$ contiguous parts, where $n$ is equal to the number of processes. Each process will take one part of the data.

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>block_map(1,N,nprocs,myid,&amp;l1,&amp;l2); for (i=l1; i&lt;=l2; i++){ x = h*(i-0.5); sum += 4.0/(1.0+x<em>x); } sum = sum</em>h;</td>
<td>call block_map(1,N,nprocs,myid,l1,l2) do i=l1, l2 x = h*(i-0.5d0) sum = sum + 4.0d0/(1.0d0+x<em>x) enddo sum = sum</em>h</td>
</tr>
</tbody>
</table>

Example of data distribution:

<table>
<thead>
<tr>
<th>Process id</th>
<th>data id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

C code: `calc_PI_block.c`
Example 6: Data Distribution

Block Distribution

| Process id | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |

```
C
void block_map(int n1, int n2, int nprocs, int myid, int *l1, int *l2)
{
    int block, rem;
    block = (n2-n1+1)/nprocs;
    rem   = (n2-n1+1)%nprocs;
    if (myid < rem){
        block++;
        *l1 = n1+myid*block;
    }else
        *l1 = n1+rem+block*myid;
    *l2 = *l1+block-1;
}
```

```
Fortran
subroutine block_map(n1,n2, nprocs, myid, l1, l2)
Integer n1, n2, nprocs, myid, l1, l2

integer block, rem
block = (n2-n1+1)/nprocs
rem   = mod(n2-n1+1, nprocs)
if (myid < rem) then
    block = block+1
    l1 = n1+myid*block
else
    l1 = n1+rem+block*myid
end if
l2 = l1+block-1
end subroutine block_map
```
Example 6: Data Distribution

Data is divided into chunks of contiguous blocks and the chunks are distributed in a round-robin manner.

### C

```c
for (i=myid*BLK+1; i<=N; i+=nprocs*BLK){
    for (j=i; j<=MIN(N,i+BLK-1); j++){
        x = h*(j-0.5);
        sum += 4.0/(1.0+x*x);
    }
}
sum = sum*h;
```

### Fortran

```fortran
do i=myid*BLK+1, N, nprocs*BLK
    do j=i, MIN(N,i+BLK-1)
        x = h*(j-0.5d0)
        sum = sum+4.0d0/(1.0d0+x*x)
    enddo
enddo
sum = sum*h
```

**calc_PI_bc.c**
Example 7: matvec-scatterv

\[
A\tilde{b} = (\tilde{a}_1 \ ... \ \tilde{a}_n)\tilde{b} = b_1\tilde{a}_1 + b_2\tilde{a}_2 + \cdots + b_n\tilde{a}_n = \tilde{c}
\]

\(\tilde{a}_i\) is a column vector.

In this program, strips of consecutive columns of A are distributed to all processes. Each process carries out a part of the linear vector sum

\[
b_i\tilde{a}_i + \cdots + b_j\tilde{a}_j
\]
Example 8: Solving the x-y Poisson Equation

Solve the partial differential equation

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)
\]

where \( x, y \in [0,1] \)

and

\( u = g(x, y) \) on boundary

Using the finite difference method.

Discretize the domain along \( x \) and \( y \) using \( n \) internal points in each direction.

The increment is

\( h = 1/(n + 1) \)

\( x_i = ih, y_j = jh \)

\( 0 \leq i, j \leq n + 1 \)

\( u_{ij} = u(x_i, y_j) = u(ih, jh) \)

\( 0 < i, j < n + 1 \)
Example 8: the x-y Poisson Equation

5-point finite difference stencil approximation:
\[ u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = h^2 f_{i,j} \quad (0 < i, j < n + 1) \]
\[ u_{i,j} = 0.25 * (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}) \]

k+1 Jacobi iteration step at \( x_i = ih; \ y_j = jh; \ i, j = 1: n \)
\[ u^{k+1}_{i,j} = 1/4(u^k_{i-1,j} + u^k_{i+1,j} + u^k_{i,j-1} + u^k_{i,j+1}) - h^2 f_{i,j} \]

Jacobi iteration across all points:
do j=1, n
\[ \text{do} i = 1, n \]
\[ \text{unew}(i, j) = 0.25*(u(i-1,j) + u(i, j+1) + u(i+1,j) + i(i, j-1)) - f(i,j)*h^2 \]
\[ \text{end do} \]
\[ \text{end do} \]
Example 8: Solving the x-y Poisson Equation

Boundary conditions (stay fixed):

\[ u(x_i, 0) = \frac{\cos(\pi x_i) - \pi^2}{\pi^2} \quad (0 \leq x \leq 1) \]

\[ u(x_i, 1) = \frac{\cos(\pi x_i) - \pi^2 \cos(x_i)}{\pi^2} \quad (0 \leq x \leq 1) \]

\[ u(0, y_j) = \frac{1}{\pi^2} - 1 \quad (0 \leq y \leq 1) \]

\[ u(1, y_j) = -\left( \frac{1}{\pi^2} + \cos(y_j) \right) \quad (0 \leq y \leq 1) \]

RHS: \[ f(x_i, y_j) = (x_i^2 + y_j^2)\cos(x_i y_j) - \cos(\pi x_i) \]
1d-Domain Decomposition

C: decompose along y axis

Fortran: decompose along x axis
Border columns need to be copied into the memory space of the neighboring processes for the five-point stencil calculation. Same for row-wise decomposition.
1d-Domain Decomposition

Exchanged border columns are stored in ‘ghost’ columns in each process to be used in five-point stencil calculation.
MPI/OpenMP Hybrid Programming

- Simplest and intuitive form: master-only: only master thread can execute MPI calls
  ```fortran
  Call MPI_INIT(ierr)
  ...
  Call MPI_SEND(...)
  ...
  !$OMP DO
  DO i=1, N
  ...
  ENDDO
  !$OMP END DO
  ...
  CALL MPI_FINALIZE(ierr)
  ```

- Starting MPI-2, the standard provides guidelines on how to interact MPI with threads

- Four levels of thread support
  - MPI_THREAD_SINGLE: Only one thread will execute
  - MPI_THREAD_FUNNELED: Only master thread will make MPI-calls
  - MPI_THREAD_SERIALIZED: Multiple threads may make MPI-calls, but only one at a time
  - MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions

```fortran
MPI_INIT_THREAD(required, provided, ierr)
```

mpiifort -qopenmp [options] prog.f90 -o prog.exe
Example 9: Hybrid Programming

Int MPI_Init_thread(int *argc, char **argv, int required, int *provided)

MPI_INIT_THREAD(required, provided, ierr)

- Four possible values for the parameter required:
  - MPI_THREAD_SINGLE
  - MPI_THREAD_FUNNELED
  - MPI_THREAD_SERIALIZED
  - MPI_THREAD_MULTIPLE

ex2_single.c
ex2_funnel.c
ex2_serialized.c
ex2_multiple.c
Example 9: poisson_1d_hybrid

- Master-only implementation

- Performance comparison using a 800x800 mesh

<table>
<thead>
<tr>
<th>Pure MPI Single-node</th>
<th>Pure MPI 2-node</th>
<th>Hybrid OMP_NUM_THREADS=2</th>
<th>Hybrid OMP_NUM_THREADS=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>np=8 80.37s</td>
<td>np=8,ppn=4 84.14s</td>
<td>np=4 161.18s</td>
<td>np=2 312.33s</td>
</tr>
<tr>
<td>np=16 43.24s</td>
<td>np=16,ppn=8 46.04s</td>
<td>np=8 84.57s</td>
<td>np=4 162.67s</td>
</tr>
<tr>
<td>np=20 36.48s</td>
<td>np=20,ppn=10 39.19s</td>
<td>np=10 72.21s</td>
<td>np=5 130.51s</td>
</tr>
<tr>
<td>np=40,ppn=20 25.88s</td>
<td>np=20,ppn=10 52.37s</td>
<td>np=10,ppn=5 74.02s</td>
<td></td>
</tr>
</tbody>
</table>

- Not all program can benefit from hybrid programming
MPI+OMP: A Case Study

- The nlace code developed by Prof. Sevan Goenezen’s group from the TAMU ME department is used to estimate the non-homogeneous elastic material properties using force and surface displacement data from multiple measurements.

- Was partially parallelized with OpenMP
  - The inverse solver has a good speedup with up to 3 CPU cores. Increasing number of cores won’t help to speedup the code further
  - Cannot process 3D cases due to extremely slow running time.
gprof is used for profiling in this project.
Amdahl’s Law

\[ S_n = \frac{1}{\frac{p}{n} + (1 - p)} \]

- \( S_n \) – speedup of the parallel code on \( n \) core vs the serial code
- \( p \) – percentage of the code that can be parallelized
- \( n \) – number of CPU cores used to run the code

According to Amdahl’s law, \( S_n \) is bounded by the serial part of the code that cannot benefit from increasing the number of CPU cores.

The speedup of the original code cannot exceed 4 due to the 25% of serial code.
Results

- Parallelized the serial portion of the code with OpenMP
- Parallelized the entire code with MPI which has successfully explored the data parallelism among different measurements.

<table>
<thead>
<tr>
<th></th>
<th>Num of Cores</th>
<th>Wall Clock Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original OpenMP</td>
<td>3</td>
<td>108m</td>
<td>1</td>
</tr>
<tr>
<td>Improved OpenMP</td>
<td>10</td>
<td>38m47s</td>
<td>2.8</td>
</tr>
<tr>
<td>OpenMP+MPI</td>
<td>100</td>
<td>3m20s</td>
<td>32.7</td>
</tr>
</tbody>
</table>
Acknowledgement

- Part of the content is adapted from former Supercomputing Facility staff Mr. Spiros Vellas’ introductory MPI short course