Introduction to Code Parallelization Using MPI

Ping Luo
TAMU HPRC

March 23, 2018
HPRC Short Course – Spring 2018
Outline

- Why parallel programming
- 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
Why Parallel Programming

- The semiconductor industry has long ago switched from boosting a single core CPU performance to producing multi-core systems
  - Free performance gain for a serial program has come to an end
- Almost all of today’s computers are multi-core systems, ranging from desktops to HPC clusters
- A serial code won’t automatically benefit from the multiple cores
- Some applications may take years to get a solution if running in serial
- Parallel programming is the key!
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
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.

![Diagram of Distributed Memory System]

- Interconnect network
Parallel Computing Systems

- A Hybrid System

Node

Interconnect network
Parallel Programming Models

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

<table>
<thead>
<tr>
<th>Parallel Programming Models</th>
<th>Parallel Computing Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>Shared Memory Systems</td>
</tr>
<tr>
<td>MPI</td>
<td>Distributed Memory Systems</td>
</tr>
<tr>
<td>MPI+OpenMP</td>
<td>Hybrid Systems</td>
</tr>
</tbody>
</table>
Ada and Terra

- Ada has 852 nodes and 17340 cores
- Terra has 304 compute nodes and 8512 cores
- Both support all three parallel programming models
  - OpenMP at node level
  - MPI at node and cluster level
  - Hybrid at node and cluster level
Resources

- Two books: "Using MPI" and "Using MPI II"
- [https://www.mpich.org/static/docs/v3.2/](https://www.mpich.org/static/docs/v3.2/)
- Examples are stored at /general/public/training/mpi/Spring2018
Books

Using MPI
Portable Parallel Programming with the Message Passing Interface, Second Edition

William Gropp
Ewing Lusk
Anthony Skjellum

Using MPI 2
Advanced Features of the Message Passing Interface

William Gropp
Ewing Lusk
Rajeev Thakur
### Example 1: Hello World

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;stdlib.h&gt;</td>
<td>program hello</td>
</tr>
<tr>
<td>int main(int argc, char **argv){</td>
<td>implicit none</td>
</tr>
<tr>
<td>printf(&quot;Hello, world\n&quot;);</td>
<td>print *, “Hello, world”</td>
</tr>
<tr>
<td>}</td>
<td>end program hello</td>
</tr>
</tbody>
</table>
## Example 1: Hello World

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;stdlib.h&gt;</code></td>
<td><code>program hello</code></td>
</tr>
<tr>
<td><code>int main(int argc, char **argv){</code></td>
<td><code>implicit none</code></td>
</tr>
<tr>
<td><code>    printf(&quot;Hello, world\n&quot;);</code></td>
<td><code>call MPI_INIT(ierr)</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>print *, &quot;Hello, world&quot;</code></td>
</tr>
<tr>
<td></td>
<td><code>call MPI_Finalize(ierr)</code></td>
</tr>
<tr>
<td></td>
<td><code>end program hello</code></td>
</tr>
</tbody>
</table>

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

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

```fortran
program hello
  use mpi
  implicit none

  print *, "Hello, world"

  end program hello
```

## Example 1: Hello World with MPI

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;stdlib.h&gt;</code></td>
<td><code>program hello</code></td>
</tr>
<tr>
<td><code>#include &lt;mpi.h&gt;</code></td>
<td><code>implicit none</code></td>
</tr>
<tr>
<td><code>int main(int argc, char **argv){</code></td>
<td><code>call MPI_INIT(ierr)</code></td>
</tr>
<tr>
<td><code>    MPI_Init(&amp;argc, &amp;argv);</code></td>
<td><code>print *, &quot;Hello, world&quot;</code></td>
</tr>
<tr>
<td><code>    printf(&quot;Hello, world\n&quot;);</code></td>
<td><code>call MPI_Finalize(ierr)</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>end program hello</code></td>
</tr>
</tbody>
</table>

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

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

```fortran
program hello
  use mpi
  implicit none

  call MPI_INIT(ierr)
  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
    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)
... no mpi calls
CALL MPI_FINALIZE(ierr)
... no mpi calls
END PROGRAM SAMPLE1
```

Multiple concurrent processes execute at their own pace unless synchronization is applied.
Compiling and Linking MPI Programs

```
module load intel/2017A
mpicc prog.c [options] -o prog.exe (C)
mpicpc prog.cpp [options] -o prog.exe (C++)
mpiifort prog.f [options] -o prog.exe (Fortran)
```

(Intel compilers)

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

(GNU compilers)
Running MPI Programs

- Load the modules first
  ```bash
  module load intel/2017A
  ```

- Run the mpi program interactively
  ```bash
  mpirun -np n [options] prog.exe [prog_args]
  ```

- Useful options
  - `ppn/-perhost`
  - `hosts`
  - `hostfile`
  - `h`
### Running MPI Programs

#### 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></td>
<td>module load intel/2017A</td>
</tr>
<tr>
<td></td>
<td>module load intel/2017A</td>
</tr>
<tr>
<td></td>
<td>mpirun prog.exe</td>
</tr>
<tr>
<td></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 programming.
- 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 programming paradigm for large scale scientific computing.
Example 2

C

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

int main(int argc, char **argv){
    int np, rank, number;
    MPI_Status status;

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

    if (rank == 0){
        number = 1234;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process %d sends out %d to process 1\n", rank, number);
    }else if(rank == 1){
        MPI_Recv(&number, 1, MPI_INT, 0, 0,
                  MPI_COMM_WORLD, &status);
        printf("Process %d receives %d from process 0\n", rank, number);
    }
    MPI_Finalize();
}
```

Fortran

```fortran
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 simple
```
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 \((\text{group}\_\text{size} - 1)\). \text{group}\_\text{size} is the size of the communicator.
- The constant \texttt{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. \texttt{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 <code>MPI_Comm_size(MPI_Comm comm, int *size)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SUBROUTINE <code>MPI_COMM_SIZE(comm, size, ierr)</code></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 <code>MPI_Comm_rank(MPI_Comm comm, int *rank)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SUBROUTINE <code>MPI_COMM_RANK(comm, rank, ierr)</code></td>
</tr>
<tr>
<td></td>
<td>integer comm, rank, ierr</td>
</tr>
</tbody>
</table>
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 communications must specify a communicator.
MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
Send and Receive a Message

C

MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status);

Fortran

call MPI_SEND(number, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
Point-to-Point Communication

- Blocking
  - MPI_Send, MPI_Recv
- Non-blocking
  - MPI_Isend, MPI_Irecv
- Send-Receive
  - MPI_Ssendrecv
## Blocking Send

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

\texttt{MPI\_SEND(buf, count, datatype, dest, tag, comm)}

- The calling process causes \texttt{count} many contiguous elements of type \texttt{datatype} to be sent, starting from \texttt{buf}.

- The message sent by \texttt{MPI\_SEND} can be received by either \texttt{MPI\_RECV} or \texttt{MPI\_IRECV}.

- \texttt{MPI\_SEND} doesn’t return (i.e., \texttt{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**

<table>
<thead>
<tr>
<th>C</th>
<th>int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)</td>
</tr>
</tbody>
</table>

```c
<type> buf(*)
integer count, datatype, source, tag, comm, ierr, status[MPI_STATUS_SIZE]
```

- `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.*
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.
The argument status in MPI_Recv 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_Recv.

status can be ignored with MPI_STATUS_IGNORE

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_Status status</strong></td>
<td><strong>integer status(MPI_STATUS_SIZE)</strong></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>MPI_Recv(...,&amp;status)</strong></td>
<td><strong>CALL MPI_RECV(...,status,ierr)</strong></td>
</tr>
<tr>
<td>Source_id = status.MPI_SOURCE</td>
<td>source_id = status(MPI_SOURCE)</td>
</tr>
<tr>
<td>tag = status.MPI_TAG</td>
<td>tag = status(MPI_TAG)</td>
</tr>
</tbody>
</table>
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();
}
```

Example 3

```fortran
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
```
Non-blocking Send

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>buf</td>
<td>initial address of send buffer</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in send buffer (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each send buffer element</td>
</tr>
<tr>
<td>IN</td>
<td>dest</td>
<td>rank of destination</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message 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 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**

**MPI_WAIT(request, status)**

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_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** - request (handle)
- **status** - status object (Status)

*status can be ignored with MPI_STATUS_IGNORE*
MPI_WAITALL(count, requests, statuses)

C

```c
integer reqs[4]
integer statuses(MPI_STATUS_SIZE,4)
...
call MPI_ISEND(...,reqs(1),ierr)
call MPI_IRECV(...,reqs(2),ierr)
call MPI_ISEND(...,reqs(3),ierr)
call MPI_IRECV(...,reqs(4),ierr)
...
... do some computations ...
... 
call MPI_WAITALL(4,reqs,statuses,ierr)
```

Fortran

```fortran
MPI_Request reqs[4];
MPI_Status status[4];
...
MPI_Isend(...,&reqs[0]);
MPI_Irecv(...,&reqs[1]);
MPI_Isend(...,&reqs[2]);
MPI_Irecv(...,&reqs[3]);
...
... do some com computations ...
...
MPI_Waitall(4,reqs,statuses);
```

statuses can be ignored with MPI_STATUSES_IGNORE
Example 4

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Request *requests;</td>
<td>integer, allocatable::requests(:)</td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
<tr>
<td>if (rank == 0){</td>
<td>if (rank == 0) then</td>
</tr>
<tr>
<td>printf(&quot;Type any number from the</td>
<td>print *, &quot;Type an integer from the</td>
</tr>
<tr>
<td>input: &quot;);</td>
<td>input&quot;</td>
</tr>
<tr>
<td>scanf(&quot;%d&quot;, &amp;number);</td>
<td>read *, number</td>
</tr>
<tr>
<td>requests = (MPI_Request <em>)malloc(npof(MPI_Request)</em>(np-1));</td>
<td>allocate(requests(np-1))</td>
</tr>
</tbody>
</table>
| for (i=1; i<np; i++)  
|   MPI_Isend(&number, 1, MPI_INT, 0, MPI_COMM_WORLD, &requests[i-1]); | do i=1, np-1 
|   MPI_Waitall(np-1, requests, MPI_STATUSES_IGNORE); |   call MPI_ISEND(number, 1, MPI_INTEGER, i, 0, &
|   free(requests);                      |   MPI_COMM_WORLD, ierr)                 |
| }else{                                 | enddo                                    |
|   MPI_Recv(&number, 1, MPI_INT, 0, MPI_COMM_WORLD, 
|     MPI_STATUS_IGNORE);                | call MPI_WAITALL(np-1, requests, MPI_STATUSES_IGNORE, ierr) |
|     printf("My id is %d. I received %d\n", rank, number); | 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                                   |
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><strong>MPI_BARRIER</strong></td>
<td>All processes within a communicator will be blocked until all processes within the communicator have entered the call.</td>
</tr>
<tr>
<td><strong>MPI_BCAST</strong></td>
<td>Broadcasts a message from one process to members in a communicator.</td>
</tr>
<tr>
<td><strong>MPI_REDUCE</strong></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><strong>MPI_GATHER</strong></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><strong>MPI_GATHERV</strong></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><strong>MPI_SCATTER</strong></td>
<td>Distribute data in sendbuf on root to recvbuf on all processes in comm.</td>
</tr>
<tr>
<td><strong>MPI_SCATTERV</strong></td>
<td>Distribute data in sendbuf on root to recvbuf on all processes in comm.</td>
</tr>
<tr>
<td><strong>MPI_ALLREDUCE</strong></td>
<td>Same as MPI_REDUCE, except the result is placed in recvbuf on all members in a communicator.</td>
</tr>
<tr>
<td><strong>MPI_ALLGATHER</strong></td>
<td>Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.</td>
</tr>
<tr>
<td><strong>MPI_ALLGATHERV</strong></td>
<td>Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.</td>
</tr>
<tr>
<td><strong>MPI_ALLTOALL</strong></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

<table>
<thead>
<tr>
<th>p0</th>
<th>p1 (root)</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2345</td>
<td>2.4345</td>
<td>-0.2374</td>
</tr>
<tr>
<td>3.1257</td>
<td>7.0321</td>
<td>3.2478</td>
</tr>
<tr>
<td>9.2134</td>
<td>-0.9234</td>
<td>0.2319</td>
</tr>
</tbody>
</table>

```
p0
1.2345
3.1257
9.2134

p1 (root)
2.4345
7.0321
-0.9234

p2
-0.2374
3.2478
0.2319

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

```
MPI_Op
MPI_SUM
```
## 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, MPI_COMPLEX, MPI_DOUBLE,</td>
</tr>
<tr>
<td></td>
<td>MPI_INT, MPI_SHORT, MPI_LONG</td>
</tr>
<tr>
<td>MPI_MIN, MPI_MAX</td>
<td>MPI_INTEGER, MPI_REAL8, 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)

\[
\begin{align*}
\text{p0} & : 1.2345, 3.1257, 9.2134 \\
\text{p1} & : 2.4345, 7.0321, -0.9234 \\
\text{p2} & : -0.2374, 3.2478, 0.2319
\end{align*}
\]

\[
\begin{align*}
\text{MPI_ALLREDUCE} & (\text{sendbuf}, \text{recvbuf}, 3, \text{float}, \text{SUM}, \text{comm}) \\
\text{MPI_Allreduce} & (\text{sendbuf}, \text{recvbuf}, 3, \text{float}, \text{SUM}, \text{comm})
\end{align*}
\]
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
- $Recvbuf$, $recvnt$, $recvtype$ 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)

C:
Fortran:

p0(root) sendbuf

1.2345
3.1257

recvcnts

1.2345
3.1257

2.4345
7.0321
-0.2374

3.2478

P1
sendbuf

P2
sendbuf

3.2478

p0 recvbuf

C: Fortran:
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:
<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Fortran:
<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
</table>

- p0(root) sendbuf
  - 1.2345
  - 3.1257

- P1 sendbuf
  - 2.4345
  - 7.0321
  - -0.2374

- P2 sendbuf
  - 3.2478

- recvbuf
  - 1.2345
  - 3.1257

- displs
  - 2
  - 3
  - 1

- recvtype
  - 0
  - 2
  - 5

- root
  - 1
  - 3
  - 6

- p0 recvbuf
  - 3.2478
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_SCATTER(
    sendbuf, 1.2345, 3.1257, 2.4345, 7.0321, -0.2374, 3.2478,
    sendcnt, p0, 1.2345, 3.1257, 2.4345, 7.0321, -0.2374, 3.2478,
    sendtype, p0, 1.2345, 3.1257, 2.4345, 7.0321, -0.2374, 3.2478,
    recvbuf, p1, 1.2345, 3.1257, 2.4345, 7.0321, -0.2374, 3.2478,
    recvcnt, p2, 1.2345, 3.1257, 2.4345, 7.0321, -0.2374, 3.2478,
    recvtype, root, 1.2345, 3.1257, 2.4345, 7.0321, -0.2374, 3.2478,
    root, comm)
```
MPI_SCATTERV

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

P0

C:
Fortran:

recvcnts

2 3 1

displs

0 2 5
1 3 6

p1

1.2345
3.1257
2.4345
7.0321
-0.2374
3.2478

p0

1.2345
3.1257

p1

2.4345
7.0321
-0.2374

p2

3.2478
MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status);
Summary

Communication: point-to-point, collective, blocking, non-blocking
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>
</tr>
</thead>
<tbody>
<tr>
<td>double t1, t2;</td>
</tr>
<tr>
<td>double elapsed;</td>
</tr>
<tr>
<td>t1 = MPI_Wtime();</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>// code segment to be timed</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>t2 = MPI_Wtime();</td>
</tr>
<tr>
<td>elapsed = t2 - t1;</td>
</tr>
</tbody>
</table>

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

\[ \pi = \int_{0}^{1} \frac{4.0}{1 + x^2} \, dx \]
Break
Outline

- 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** (Task 1)
- **Preparing 12 steaks** (Task 2)
- **Preparing 12 deserts** (Task 3)

Task parallelism

- **4 meals**
  - salad
  - steak
  - desert

Data parallelism
Example 6: Data Distribution

Cyclic Distribution

Process id

| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 |

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

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

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.

### C

```c
block_map(1,N,nprocs,myid,&l1,&l2);
for (i=l1; i<=l2; i++){
    x = h*(i-0.5);
    sum += 4.0/(1.0+x*x);
}
sum = sum*h;
```

### Fortran

```fortran
call block_map(1,N,nprocs,myid,l1,l2)
do i=11, 12
    x = h*(i-0.5d0)
    sum = sum + 4.0d0/(1.0d0+x*x)
enddo
sum = sum*h
```

calc_PI_block.c
Example 6: Data Distribution

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

**Block Distribution**

C

```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

```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.

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
</table>
| 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; | 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

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

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

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

\[ b_i\tilde{\mathbf{a}}_i + \ldots + b_j\tilde{\mathbf{a}}_j \]
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:

\[ u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = h^2f_{i,j} \quad (0 < i, j < n + 1) \]

\[ u_{i,j} = 0.25 \times (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2f_{i,j}) \]

k+1 Jacobi iteration step at \( x_i = ih; \ y_j = jh; \ i, j = 1: n \)

\[ u^{k+1}_{i,j} = \frac{1}{4}(u^k_{i-1,j} + u^k_{i,j+1} + u^k_{i,j-1} + u^k_{i+1,j}) - h^2f_{i,j} \]

Jacobi iteration across all points:

```
do j=1, n
    do i = 1, n
        unew(i, j) = 0.25*(u(i-1,j) + u(i, j+1) + u(i+1,j) + u(i, j-1)) - f(i,j)*h^2
    end do
end do
```
Example 8: Solving the x-y Poisson Equation

Boundary conditions (stay fixed):

\[
\begin{align*}
    u(x_i, 0) &= \frac{\cos(\pi x_i) - \pi^2}{\pi^2} & (0 \leq x \leq 1) \\
    u(x_i, 1) &= \frac{\cos(\pi x_i) - \pi^2 \cos(x_i)}{\pi^2} & (0 \leq x \leq 1) \\
    u(0, y_j) &= \frac{1}{\pi^2} - 1 & (0 \leq y \leq 1) \\
    u(1, y_j) &= -\left(\frac{1}{\pi^2} + \cos(y_j)\right) & (0 \leq y \leq 1)
\end{align*}
\]

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
1d-Domain Decomposition

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.
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**:

<table>
<thead>
<tr>
<th>Support Levels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_THREAD_SINGLE</td>
<td>Only one thread will execute.</td>
</tr>
<tr>
<td>MPI_THREAD_FUNNELED</td>
<td>Process may be multi-threaded, but only the main thread will make MPI calls (calls are “funneled” to main thread). <em>Default</em></td>
</tr>
<tr>
<td>MPI_THREAD_SERIALIZE</td>
<td>Process may be multi-threaded, and any thread can make MPI calls, but threads cannot execute MPI calls concurrently; they must take turns (calls are “serialized”).</td>
</tr>
<tr>
<td>MPI_THREAD_MULTIPLE</td>
<td>Multiple threads may call MPI, with no restriction.</td>
</tr>
</tbody>
</table>
Example 9: Hybrid Programming

cmpiicc  -qopenmp [options] prog.c –o prog.exe
ncmpifort –qopenmp [options] prog.f90 –o prog.exe

ex2_single.c        ex2_serialized.c
ex2_funnel.c        ex2_multiple.c
### MPI+OMP Hybrid Programming

- Not all programs will benefit from hybrid programming.

---

**poisson_1d performance comparison: pure MPI vs hybrid**

<table>
<thead>
<tr>
<th>np</th>
<th>Pure MPI</th>
<th>Hybrid OMP_NUM_THREADS=2</th>
<th>Hybrid OMP_NUM_THREADS=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>22.8697s</td>
<td>25.0954s</td>
<td>49.7846s</td>
</tr>
<tr>
<td>4</td>
<td>11.9441s</td>
<td>13.3360s</td>
<td>25.0660s</td>
</tr>
<tr>
<td>8</td>
<td>6.5824s</td>
<td>7.5272s</td>
<td>13.3054s</td>
</tr>
<tr>
<td>16</td>
<td>3.6141s</td>
<td>4.6635s</td>
<td>7.7067s</td>
</tr>
</tbody>
</table>

**Hybrid is worse**

- example8/poisson_1d.c (.f90)
- example9-hybrid/poisson_1d_hybrid.c
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.
Profiling

- **gradfun** (99.7%)
- **stiff_residl** (70.59%)
- **residl_dual** (%12.35)
- **calcgrad** (%12.32)

- **Forward solver**
- **Initialization**

Called once only

Paralleized with OpenMP

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