Introduction to Code Parallelization Using MPI

Marinus Pennings
November 20, 2020

Original slides created by Ping Luo
Outline

- Parallel programming models
- Layout of an MPI program
- Compiling and running an MPI program
- Basic MPI concepts
- Point-to-point communication
- Collective communication

To setup examples, type: `/scratch/training/mpi/setup.sh` (terra and ada)

tamu github repo:  git@github.tamu.edu:pennings/hprc_shortcourse_mpi.git
HPRC Systems Summary

- **terra**: 304 compute nodes, 28 cores per node, 64GB per node
- **ada**: 852 compute nodes, 20 cores per node. 64GB per node
- **grace**: 917 compute nodes, 48 cores per node, 384GB per node
Parallel Programming Models

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

Examples:
- single terra/ada node
- your desktop/laptop

NOTE: every node on terra has 28 cores (48 on grace and 20 on ada). Average desktop/laptop has 4 - 8 cores

Programming model for shared memory Systems: OpenMP
(OpenMP is most popular programming model for shared memory parallelism)
Parallel Programming Models

- **Distributed Memory System** – an abstraction of a parallel system where each processor has its own local memory.

Programming model for distributed memory Systems: MPI

**NOTE:** processes can be on a single node or distributed over multiple nodes.
Parallel Programming models

- Hybrid model

Hybrid Programming model: MPI + OpenMP
(for example: one MPI task per node with 4 OpenMP threads per MPI task)
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 underneath architecture.
  - MPI-4.0 is under discussion (last draft specification November 2020)

- 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 1: Hello World (C/C++)

<table>
<thead>
<tr>
<th>Serial C Code</th>
<th>Parallel C Code Using MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;stdlib.h&gt;</code></td>
<td><code>#include &lt;stdlib.h&gt;</code></td>
</tr>
<tr>
<td><code>int main(int argc, char **argv){</code></td>
<td><code>#include &lt;mpi.h&gt;</code></td>
</tr>
<tr>
<td><code> }</code></td>
<td><code>int main(int argc, char **argv){</code></td>
</tr>
<tr>
<td><code> printf(&quot;Hello, world\n&quot;);</code></td>
<td><code>MPI_Init(&amp;argc, &amp;argv);</code></td>
</tr>
<tr>
<td><code> }</code></td>
<td><code> printf(&quot;Hello, world\n&quot;);</code></td>
</tr>
<tr>
<td><code> </code></td>
<td><code>MPI_Finalize();</code></td>
</tr>
</tbody>
</table>
### Example 1: Hello World (Fortran)

<table>
<thead>
<tr>
<th>Serial Fortran Code</th>
<th>Parallel Fortran Code using MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program hello</code></td>
<td><code>program hello</code></td>
</tr>
<tr>
<td><code>implicit none</code></td>
<td><code>use mpi</code></td>
</tr>
<tr>
<td><code>print *, “Hello, world”</code></td>
<td><code>implicit none</code></td>
</tr>
<tr>
<td><code>end program hello</code></td>
<td><code>call MPI_INIT(ierr)</code></td>
</tr>
<tr>
<td></td>
<td><code>print *, “Hello, world”</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>

```fortran
program hello
use mpi
implicit none
print *, "Hello, world"
call MPI_INIT(ierr)
print *, "Hello, world"
call MPI_Finalize(ierr)
end program hello
```

demo example1-hello_world. How to compile?
Compiling and Linking MPI Programs

We will use Intel MPI implementation in the examples

(Using Intel compiler underneath)

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

(Using GNU compilers underneath)

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

don’t forget to load an mpi toolchain. We will use intel/2019b here

demo how to compile? Try to run
Running an MPI Program

- Load the module
  ```
  module load intel/2019b
  ```

- Use `mpirun` to run your program
  ```
  mpirun -np n [options] prog.exe [prog_args]
  (n is number of “tasks” that will be started)
  ```

- Useful MPI options
  ```
  -ppn/-perhost, -hosts, -hostfile
  example: mpirun -np 4 -hosts login1,login2 -ppn 1 mympi.exe
  ```

**NOTE:** When testing always try on single node before running on multiple nodes

https://hprc.tamu.edu/wiki/Ada:Compile:All#MPI_Programs

run example, single/multiple node, also use hostname
## Running MPI Program in Batch

<table>
<thead>
<tr>
<th>ada</th>
<th>terra/grace</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>
</tbody>
</table>

module load intel/2019b
mpirun prog.exe

module load intel/2019b
mpirun prog.exe

---

No need to specify number of tasks when running MPI jobs using the batch system?
### Layout of an MPI Program

#### C

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

int main(int argc, char **argv) {
    int ierr;
    ... no mpi calls
    MPI_Init(&argc, &argv);
    ... 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
MPI_Finalize(ierr)
... no mpi calls
CALL MPI_Finalize(ierr)
... no mpi calls
END PROGRAM SAMPLE1
```

- **mpi calls** happen here
- multiple concurrent processes execute at their own pace unless synchronization is applied.
- **mpi calls** happen here

---

**Notes:**

- The layout of an MPI program shows the initialization (`MPI_Init`) and finalization (`MPI_Finalize`) of the MPI library.
- The source code examples demonstrate how MPI calls are made within the program.
- In Fortran, the `USE MPI` statement is used to import MPI functions.
- In C, the `#include <mpi.h>` directive is used to include the MPI header file.
- The program structure in C and Fortran is similar, with initialization and finalization blocks.
- The comments indicate where MPI calls are made and where synchronization is applied.

---

**Texas A&M University     High Performance Research Computing  –  http://hprc.tamu.edu**

---
MPI Communicator

- A communicator is a software structure that specifies and maintains a group of processes.
- Each process (also named task) inside 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 the main communicator that includes all the MPI tasks spawned by mpirun.
- Communicators are especially useful for characterizing the tasks that different groups of processes carry out.
Every MPI communication must specify a communicator.

- **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
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 (id) 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>

exercise ex1_hello_tasks
### Example: Hello world (part 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);

    printf("Hello from task %d out of %d tasks",rank,np);
    fprintf("Hello from task %d out of %d tasks",rank,np);
    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)
    print *, "hello from task ", rank, "out of ", np, " tasks"
    call MPI_Finalize(ierr)
end program simple
```

Exercise ex1
Point-to-Point Communication

- **Blocking**
  - MPI_Send, MPI_Recv
- **Non-blocking**
  - MPI_Isend, MPI_Irecv
- **Send-Receive**
  - MPI_Sendrecv
Blocking Send

**C**

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

**Fortran**

```fortran
MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierr
```

- **buf**: starting 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
# 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)
<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

MPI\_ANY\_SOURCE and MPI\_ANY\_TAG are MPI defined wildcards.
### Example 2 – One Sender and One Receiver

**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 %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
```
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, 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
end program simpleC
Tag is an integer used in a message to differentiate one message from other messages.

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

Where to fetch/store the data
Comments on Blocking Send

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

- The calling process sends a contiguous block of data (count elements, type datatype), starting at 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 (see https://www.codingame.com/playgrounds/349/introduction-to-mpi/communication-modes for the various modes)
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, starting at the address 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 amount of 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`.

Return Status

**C**

```c
MPI_Status status
...
MPI_Recv(...,&status)
Source_id = status.MPI_SOURCE
tag = status.MPI_TAG
```

**Fortran**

```fortran
integer status(MPI_STATUS_SIZE)
...
CALL MPI_RECV(...,status,ierr)
source_id = status(MPI_SOURCE)
tag = status(MPI_TAG)
```
Timing

MPI_WTIME()

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

C/C++

double t1, t2, elapsed;
t1 = MPI_Wtime();
...  
// code segment to be timed
...
t2 = MPI_Wtime();
elapsed = t2 - t1;

Fortran

real*8 t1, t2
real*8 elapsed
t1 = MPI_WTIME()
...
! Code segment to be timed
...
t2 = MPI_WTIME()
elapsed = t2 - t1
Case Study: Computing Pi (1)

Monte Carlo method to compute Pi:

- assume a circle with radius 1, the enclosing box will be 2 by 2
- surface of circle = \( \pi r^2 = \pi \times 1^2 = \pi \), surface enclosing box = 2\(\times\)2 = 4
- if you pick a very large number of random points (x and y between 0 and 1) the fraction of the number of points inside the circle will be \(\pi/4\). To compute \(\pi\):
  - Generate a large number of random points (x, y)
  - Count #points inside the circle (inside if \(\sqrt{x^2+y^2} \leq 1\))
  - #inside / #total == \(\pi/4\) → \(\pi = (4\times\text{#inside})/\text{#total}\)

First (very naive) Attempt

1. Root distributes the large array of random points among all the tasks
2. every task locally computes #points inside the circle
3. every task sends local #points back to root
4. root adds up all the local #points and computes pi using the formula
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

```c
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>
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>Function</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>
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
### MPI_WAIT

**MPI_WAIT(request, status)**

- **request**: request (handle)
- **status**: status object (Status)

---

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>```c</td>
<td>```c</td>
</tr>
<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>

**status** can be ignored with MPI_STATUS_IGNORE
MPI_WAITALL

MPI_WAITALL(count, requests, statuses)

| count       | lists length (non-negative integer) |
| requests    | array of requests (array of handles) |
| statuses    | array of status objects (array of Status) |

C

```c
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 computations ...
...
MPI_Waitall(4,reqs,statuses);
```

Fortran

```fortran
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)
```

status can be ignored with MPI_STATUSES_IGNORE

dexample4-non_blocking_send_receive
Example 4 (non blocking send)

### 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
```
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
```
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.
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>
</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_SCATTER

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

- Scatters data from root to all tasks in comm
- Data Sent by root is assigned by rank order.
- sendcnt is number of elements send to each task
- sendbuf, sendcnt, sendtype are significant only at root.
MPI_SCATTERV

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

sendcnts

C:
Fortran:

displs

p0
recvbuf

1.2345
3.1257
2.4345
7.0321
-0.2374
3.2478

p1
sendbuf

p1
recvbuf

p2
recvbuf
MPI_SCATTERV

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

sendcnts
2 3 1

displs
C:
0 2 5
Fortran:
1 3 6

1.2345
3.1257
2.4345
7.0321
-0.2374
3.2478

p0 recvbuf

p1 recvbuf

p2 recvbuf
MPI_SCATTERV

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

sendcnts

<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>

displs

C:  

<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>

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>

p0  
recvbuf

1.2345  
3.1257

p1  
recvbuf

1.2345  
3.1257
2.4345  
7.0321  
-0.2374
3.2478

p2  
recvbuf

3.2478

sendbuf

1.2345  
3.1257
Case Study: Computing Pi (2)

In the first version, the root distributed all the random points one by one among all the tasks. In this exercise, instead of sending all the random points, we will use the MPI_SCATTER function to distribute the input.
MPI_BCAST

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

- **root** sends **count** elements of type **datatype**, starting at **buf** to all tasks (including **root**) in communicator **comm**.
- Non-root tasks receive data from **root**.
- Each receiving task blocks until the message has arrived in its **buffer**.
- All tasks in **comm** must call this routine.
Case Study: Computing Pi (3)

Let's adjust the compute_pi program. Instead of the root sending (or scattering) all the random points, root will only broadcast the total number of points. Every task will generate the random points themselves.

(shows overhead of excessive communication)
MPI_GATHER

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

- Gathers data from all tasks in comm and stores them in task root.
- Data received by root is 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)

C:
0  2  5
1  3  6

Fortran:
0  2  5
1  3  6

p0 recvbuf

C:
1  3  6

p0 sendbuf

1.2345  3.1257
2.4345  7.0321  -0.2374
3.2478

P1

sendbuf

1.2345  3.1257
2.4345  7.0321  -0.2374

P1

sendbuf

P2

sendbuf

1.2345  3.1257
2.4345  7.0321  -0.2374
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.
Case Study: Computing Pi (4)

In the previous version, all the tasks sent their results back to the root and the root received them one by one and combined (i.e. reduced) the results. In this exercise we will use MPI_GATHER to collect the results.
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 tasks 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>1.2345</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.1257</td>
</tr>
<tr>
<td></td>
<td>9.2134</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p1 (root)</th>
<th>2.4345</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7.0321</td>
</tr>
<tr>
<td></td>
<td>-0.9234</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p2</th>
<th>-0.2374</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.2478</td>
</tr>
<tr>
<td></td>
<td>0.2319</td>
</tr>
</tbody>
</table>

+ MPI_SUM

<table>
<thead>
<tr>
<th>p1</th>
<th>3.4306</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13.4056</td>
</tr>
<tr>
<td></td>
<td>8.5219</td>
</tr>
</tbody>
</table>
# 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, MPI_INT, MPI_SHORT, MPI_LONG</td>
</tr>
<tr>
<td>MPI_MIN, MPI_MAX</td>
<td>MPI_INTEGER, MPI_REAL8, MPI_INT, MPI_SHORT, MPI_LONG</td>
</tr>
<tr>
<td>MPI_LAND, MPI_LOR, MPI_LXOR</td>
<td>MPI_LOGICAL, MPI_INT, MPI_SHORT, MPI_LONG</td>
</tr>
<tr>
<td>MPI_BAND, MPI_BOR, MPI_BXOR</td>
<td>MPI_INTEGER, MPI_INT, MPI_SHORT, MPI_LONG</td>
</tr>
</tbody>
</table>
MPI_ALLREDUCE

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

<table>
<thead>
<tr>
<th>p0</th>
<th>p1</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>

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

```
+  
```

```
p0
1.2345
3.1257
9.2134
```

```
p1
2.4345
7.0321
-0.9234
```

```
p2
-0.2374
3.2478
0.2319
```

```
p0
3.4306
13.4056
8.5219
```

```
p1
3.4306
13.4056
8.5219
```

```
p2
3.4306
13.4056
8.5219
```
In the previous version, root gathered the partial results and combined (i.e. reduced) the results manually. Let’s use MPI_REDUCE to do the last step.
DEM0: matvec

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

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

Use the following steps to calculate the matrix vector multiplication:
1. Distribute the columns of matrix \( \mathbf{A} \) among all the tasks
2. Distribute vector \( \mathbf{b} \) among all the tasks
3. Each task computes partial mat-vec multiplication \( (b_i \mathbf{a}_i + \cdots + b_j \mathbf{a}_j) \)
4. Root will collect the partial results and combine them to complete the multiplication

What MPI functions do we use to distribute the data and collect the results?
Some final Considerations

- Exploring parallelism
  - Task-parallelism
  - Data-parallelism
- MPI/OMP hybrid programming
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

- 12 salads
- 12 steaks
- 12 deserts

**Task parallelism**

1. **Preparing 12 salads**
2. **Preparing 12 steaks**
3. **Preparing 12 deserts**

**Data parallelism**

- 4 meals:
  - Task 1: salad, steak, desert
  - Task 2: salad, steak, desert
  - Task 3: salad, steak, desert
  - Task 4: salad, steak, desert
Data Distribution: cyclic

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

C

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

Fortran

```fortran
do i=myid+1, N, nprocs
    x = h*(i-0.5d0)
    sum = sum+4.0d0/(1.0d0+x*x)
enddo
sum = sum*h;
```
Data Distribution: block

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);</td>
<td></td>
</tr>
<tr>
<td>for (i=l1; i&lt;=l2; i++){</td>
<td></td>
</tr>
<tr>
<td>x = h*(i-0.5);</td>
<td></td>
</tr>
<tr>
<td>sum += 4.0/(1.0+x*x);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>sum = sum*h;</td>
<td></td>
</tr>
<tr>
<td>call block_map(1, N, nprocs, myid, l1, l2)</td>
<td></td>
</tr>
<tr>
<td>do i=l1, l2</td>
<td></td>
</tr>
<tr>
<td>x = h*(i-0.5d0)</td>
<td></td>
</tr>
<tr>
<td>sum = sum + 4.0d0/(1.0d0+x*x)</td>
<td></td>
</tr>
<tr>
<td>enddo</td>
<td></td>
</tr>
<tr>
<td>sum = sum*h;</td>
<td></td>
</tr>
</tbody>
</table>
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
```
**Data Distribution: block cyclic**

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

<table>
<thead>
<tr>
<th>Process id</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</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>

### C Code

```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 Code

```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
```

**example6-data_distribution/calc_PI_bc**
MPI/OpenMP Hybrid Programming

- Simplest and intuitive form: master-only: only master thread can execute MPI calls
  
  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

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_FUNNELD
  - MPI_THREAD_SERIALIZED
  - MPI_THREAD_MULTIPLE

  ex2_single.c
  ex2_funnel.c
  ex2_serialized.c
  ex2_multiple.c
Acknowledgement

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

You can always reach us at help@hprc.tamu.edu