Introduction to Code Parallelization Using MPI (Part I)

Ping Luo
TAMU HPRC

September 24, 2017
HPRC Short Course – Fall 2017
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 will use the term *processor* to refer to the smallest physical processing unit where a program is executed. It is synonymous to CPU *core* in this sense.

- 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]

- Processors (P1, P2, P3, P4) with their own local memory (Mem).
- Interconnect network connecting all processors.
Parallel Computing Systems

- A Hybrid System

![Diagram of a hybrid system with nodes and interconnect network](image-url)
### Parallel Programming Models

- Mapping from the parallel programming models to the computing systems

<table>
<thead>
<tr>
<th>Model</th>
<th>System Type</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
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
```
**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

call MPI_INIT(ierr)
print *, "Hello, world"
call MPI_Finalize(ierr)
end program hello
```
## Layout of an MPI Program

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;mpi.h&gt;</code></td>
<td><code>PROGRAM SAMPLE1</code></td>
</tr>
<tr>
<td><code>int main()</code></td>
<td><code>USE MPI !F90</code></td>
</tr>
<tr>
<td>```</td>
<td><code>!f77: include &quot;mpif.h&quot;</code></td>
</tr>
<tr>
<td><code>    int argc, char **argv)</code></td>
<td><code>integer ierr</code></td>
</tr>
<tr>
<td>```</td>
<td>```</td>
</tr>
<tr>
<td><code>    {</code></td>
<td><code>CALL MPI_INIT(ierr)</code></td>
</tr>
<tr>
<td>```</td>
<td>```</td>
</tr>
<tr>
<td><code>        no mpi calls</code></td>
<td><code>CALL MPI_FINALIZE(ierr)</code></td>
</tr>
<tr>
<td>```</td>
<td>```</td>
</tr>
<tr>
<td><code>        MPI_Init(&amp;argc, &amp;argv);</code></td>
<td><code>... no mpi calls</code></td>
</tr>
<tr>
<td>```</td>
<td>```</td>
</tr>
<tr>
<td><code>    MPI_Finalize();</code></td>
<td><code>END PROGRAM SAMPLE1</code></td>
</tr>
<tr>
<td>```</td>
<td>```</td>
</tr>
<tr>
<td><code>    ... no mpi calls</code></td>
<td>```</td>
</tr>
<tr>
<td>```</td>
<td>```</td>
</tr>
<tr>
<td><code>}</code></td>
<td>```</td>
</tr>
</tbody>
</table>

*mpi calls happen here*

multiple concurrent *processes* execute at their own pace unless synchronization is applied.

*mpi calls happen here*
Compiling and Linking MPI Programs

module load intel/2017A
mpiicc  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)

mpiicc  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
  
  \texttt{module load intel/2017A}

- Run the \texttt{mpi} program interactively
  
  \texttt{mpirun -np n [options] prog.exe [prog_args]}

- Useful options
  
  \texttt{-ppn/-perhost}
  \texttt{-hosts}
  \texttt{-hostfile}
  \texttt{-h}
## Running MPI Programs

### Batch Examples

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

module load intel/2017A
mpirun prog.exe

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 mostly 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 communications must specify a communicator.
Basic MPI Concepts

message, point-to-point communication, collective communication
<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;stdlib.h&gt;</code></td>
<td>program simple</td>
</tr>
<tr>
<td><code>#include &lt;mpi.h&gt;</code></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>if (rank == 0) {</td>
<td>number = 1234</td>
</tr>
<tr>
<td>number = 1234;</td>
<td>call MPI_SEND(number, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)</td>
</tr>
<tr>
<td>MPI_Send(&amp;number, 1, MPI_INT, 0, MPI_COMM_WORLD);</td>
<td>printf(&quot;Process %d sends out %d to process 1\n&quot;, rank, number);</td>
</tr>
<tr>
<td>printf(&quot;Process %d sends out %d to process 1\n&quot;, rank, number);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>else if (rank == 1) then</td>
</tr>
<tr>
<td>else if (rank == 1) {</td>
<td>call MPI_RECV(number, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, status, ierr)</td>
</tr>
<tr>
<td>MPI_Recv(&amp;number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &amp;status);</td>
<td>printf(&quot;Process %d receives %d from process 0\n&quot;, rank, number);</td>
</tr>
<tr>
<td>printf(&quot;Process %d receives %d from process 0\n&quot;, rank, number);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>endif</td>
</tr>
<tr>
<td>MPI_Finalize();</td>
<td>call MPI_Finalize(ierr)</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>
MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
Send and Receive a Message

- The structure of a message and how the data will be sent and received determine the interface of send and receive operations are similar.

C

```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

```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_Sendrecv
## Blocking Send

### C

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

### Fortran

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

- **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)
<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.
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

\[
\text{MPI_RECV}(\text{buf},\text{count},\text{datatype},\text{source},\text{tag},\text{comm},\text{status})
\]

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

- The argument \texttt{status} in MPI\_Recv provides a way of retrieving message source, message tag, and message error from the message.

- \texttt{status} is useful when MPI wildcards (MPI\_ANY\_SOURCE, MPI\_ANY\_TAG) are used in MPI\_Recv.

- \texttt{status} can be ignored with \texttt{MPI\_STATUS\_IGNORE}

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{MPI_Status status...}</td>
<td>\texttt{integer status(MPI_STATUS_SIZE)...}</td>
</tr>
<tr>
<td>MPI_Recv(...,&amp;status)</td>
<td>CALL MPI_RECV(...,status,ierr)</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();
}
```

```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)

| IN  | buf       | initial address of send buffer |
| IN  | count     | number of elements in send buffer |
|     |           | (non-negative integer)          |
| IN  | datatype  | datatype of each send buffer element |
| IN  | dest      | rank of destination             |
| IN  | tag       | message tag                     |
| IN  | comm      | communicator                    |
| OUT | request   | communication request           |
|     |           | (a handle that can be used later to refer the outstanding receive) |
Non-blocking Receive

`MPI_IRECV(buf, count, datatype, source, tag, comm, request)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>initial address of send buffer</td>
</tr>
<tr>
<td>count</td>
<td>number of elements in receive buffer (non-negative integer)</td>
</tr>
<tr>
<td>datatype</td>
<td>datatype of each receive buffer element</td>
</tr>
<tr>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
</tr>
<tr>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
<tr>
<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>request</th>
<th>request (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>status object (Status)</td>
</tr>
</tbody>
</table>

<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,...,</td>
<td>call MPI_Irecv(recv_buf,count,...&amp;</td>
</tr>
<tr>
<td>comm,&amp;request);</td>
<td>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)

---

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

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

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Request *requests;</code></td>
<td><code>integer, allocatable::requests(:)</code></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 input: &quot;);</td>
<td>print *, &quot;Type an integer from the 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>
<tr>
<td>for (i=1; i&lt;np; i++)</td>
<td>do i=1, np-1</td>
</tr>
<tr>
<td>MPI_Isend(&amp;number, 1, MPI_INT, 0, MPI_COMM_WORLD,</td>
<td>call MPI_ISEND(number, 1, MPI_INTEGER, i, 0, &amp;</td>
</tr>
<tr>
<td>&amp;requests[i-1]);</td>
<td>MPI_COMM_WORLD, ierr)</td>
</tr>
<tr>
<td>MPI_Waitall(np-1, requests, MPI_STATUSES_IGNORE);</td>
<td>enddo</td>
</tr>
<tr>
<td>free(requests);</td>
<td>call MPI_WAITALL(np-1, requests, MPI_STATUSES_IGNORE, ierr)</td>
</tr>
<tr>
<td>}else{</td>
<td>deallocate(requests)</td>
</tr>
<tr>
<td>MPI_Recv(&amp;number, 1, MPI_INT, 0, MPI_COMM_WORLD,</td>
<td>else</td>
</tr>
<tr>
<td>MPI_STATUS_IGNORE);</td>
<td>call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, &amp;</td>
</tr>
<tr>
<td>printf(&quot;My id is %d. I received %d\n&quot;, rank, number);</td>
<td>MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)</td>
</tr>
<tr>
<td>}</td>
<td>print &quot;(2(A,I6))&quot;,&quot;Process &quot;,rank,&quot; received &quot;, number</td>
</tr>
</tbody>
</table>
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>
<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 \texttt{comm}.
- Non-root processes: receives a message from the specified \texttt{root}.
- Each receiving process blocks until the message has arrived its \texttt{buffer}.
- All processes in \texttt{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

\[
\begin{array}{c}
\text{p0} \\
1.2345 \\
3.1257 \\
9.2134 \\
\text{p1 (root)} \\
2.4345 \\
7.0321 \\
-0.9234 \\
\text{p2} \\
-0.2374 \\
3.2478 \\
0.2319 \\
\end{array}
\]

\[
\begin{array}{c}
\text{MPI_REDUCE} \\
\text{MPI_SUM} \\
3.4306 \\
13.4056 \\
8.5219 \\
\end{array}
\]
## Predefined Reduction Operations

<table>
<thead>
<tr>
<th>PREDEFINDED 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,</td>
</tr>
<tr>
<td></td>
<td>MPI_INT, MPI_SHORT, MPI_LONG,</td>
</tr>
<tr>
<td></td>
<td>MPI_LONG, MPI_DOUBLE</td>
</tr>
<tr>
<td>MPI_BAND, MPI_BOR,</td>
<td>MPI_INTEGER, MPI_INT,</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>MPI_SHORT, MPI_LONG</td>
</tr>
<tr>
<td></td>
<td>MPI_LONG</td>
</tr>
</tbody>
</table>
MPI_ALLREDUCE

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.
- recvctn is number of elements received per process.
- Recvbuf, recvctn, 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:

p0 recvbuf

P1
sendbuf
2.4345
7.0321
-0.2374

P2
sendbuf
3.2478

p0(root)
sendbuf
1.2345
3.1257

recvtype, root, comm

C:
Fortran:

recv cnts
2 3 1

displs
MPI_GATHERV (cont.)

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

C:
Fortran:

C:    0  2  5
Fortran:  1  3  6

sendbuf

recvbuf

p0(root) 1.2345 3.1257
P1 2.4345 7.0321 -0.2374
P2 3.2478

 recvcnts
2 3 1

 displs

MPI_GATHERV (cont.)
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 recvbuf.
MPI_SCATTER

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

MPI_SCATTER

\[
\begin{aligned}
&\text{sendbuf} = \begin{bmatrix}
1.2345 \\
3.1257 \\
2.4345 \\
7.0321 \\
-0.2374 \\
3.2478
\end{bmatrix} \\
&\text{recvbuf} = \begin{bmatrix}
p0 \\
p1 \\
p2
\end{bmatrix}
\end{aligned}
\]
MPI_SCATTERV

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

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

- C:
  ```
  recvcnts
  2 3 1
  displs
  0 2 5
  1 3 6
  ```

- Fortran:
  ```
  recvcnts
  2 3 1
  displs
  0 2 5
  1 3 6
  ```

- Diagram:
  - p0
    - recvctns: 1.2345, 3.1257
    - displs: 2
  - p1
    - recvctns: 2.4345, 7.0321, -0.2374
    - displs: 3
  - p2
    - recvctns: 3.2478
    - displs: 4
Timing Routine

**MPI_WTIME()**

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

### C

double t1, t2;
double 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
Important Note On Using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs
Example 5: Calculate PI

\[ \pi = \int_0^1 \frac{4.0}{1 + x^2} \, dx \]
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