Introduction to Code Parallelization Using MPI

A three-session short course by High Performance Research Computing Texas A&M University
Instructor: Ping Luo
Agenda

- **Day 1**
  - Basic concepts
  - Compiling and running MPI programs
  - Point-to-point communication: blocking send/receive

- **Day 2**
  - Point-to-point communication: non-blocking send/receive, MPI_Sendrecv
  - Scheduling methods
  - Collective communication

- **Day 3**
  - More on collective communication
  - Solving xy-possion equations
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*
Experiment Environment

- **Connect to Eos/Ada**
  - For windows users
    - Mobaxterm + xming
    - Putty + xming
  - For Linux/Mac Os users
    - $ ssh -X eos.tamu.edu or $ ssh -X ada.tamu.edu

- **Copy the course material**
  - $ setup-mpi-training.sh
  - $ cd $SCRATCH/mpi-training

- **Select an editor to view and edit the source code**
  - gedit (with GUI) $ gedit &
  - vi
  - emacs
Distributed Memory System

Interconnect Network
Shared Memory System

processor processor ... processor

Memory
Eos and Ada are hybrid clusters.
Process and Thread

Process (or task): an instance of a program execution

Thread: a light weight process
Viewing Processes

Windows task (process) manager

Linux

$ top
Viewing Threads

top –u username, then press ‘H’
Example 1: Hello World!

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

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

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

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

- `$ module load intel/2015B`
- `$ cd demos/example1`
- `$ mpiicc hello.c -o hello.exe`
- `$ mpiifort hello.f90 -o hello.exe`
- `$ mpirun -np 4 ./hello.exe`

Hello, world!
Hello, world!
Hello, world!
Hello, world!
What is MPI

- **Message Passing Interface**: a specification for the library interface that implements message passing in parallel programming.
- Is standardized by MPI forum for implementing portable, flexible, and reliable codes for **distributed memory systems**, regardless of the underneath architectures.
  - Evolving over time: MPI-2(1998), MPI-3(2012),
- Has C/C++/Fortran bindings.
  - C++ binding deprecated since MPI-2.2
- Different implementations (libraries): Intel MPI, MPICH, OpenMPI, etc.
- It is the mostly widely used parallel programming system for large scale scientific computing.
MPI: Key Aspects

- MPI is designed specifically for distributed memory systems.
- A pure MPI program is executed by processes (or tasks), not threads.
- Each MPI process has its own address space (AS)
  - Each process can only directly access its own AS, but cannot directly access another process' AS.
- All processes communicate by explicitly passing messages.
  - Point-to-point communication
  - Collective communication
  - One-sided communication (will not be covered)
- MPI is to distributed memory systems, as OpenMP is to shared memory systems.
- MPI can also be used for shared memory and hybrid systems (like Eos and Ada).
Question: how many copies of \textit{num} when the program is running?
Example 1: Longer Version

C

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

int main(int argc, char **argv){
  int np, rank;
  int name_len;

  char proc_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc, &argv);
  MPI_Get_processor_name(proc_name, &name_len);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello, world\n" from process %d on node %s, total %d processes\n", rank, proc_name, np);
  MPI_Finalize();
}
```

Fortran

```fortran
program hello
  use mpi
  implicit none
  integer ierr, name_len, np, rank;
  character (len=MPI_MAX_PROCESSOR_NAME) proc_name

  call MPI_INIT(ierr)
  call MPI_GET_PROCESSOR_NAME(proc_name, name_len,&
               ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  print "'(A, I4, 3(A), I4, A)'", "'Hello, world!' from process ", &
  rank, " on node ", trim(proc_name), " total ", np, " processes."
  call MPI_Finalize(ierr)

end program hello
```
Compiling and Linking (Intel MPI)

- **Eos**
  
  Module load `intel/2015B`
  
  ```
  mpiicc  prog.c  [options]  -o prog.exe  (C)
  mpiicpc  prog.cpp  [options]  -o prog.exe  (C++)
  mpiifort  prog.f  [options]  -o prog.exe  (Fortran)
  ```

- **Ada**
  
  module load `intel/2015B`
  
  ```
  mpiicc  prog.c  [options]  -o prog.exe  (C)
  mpicpc  prog.cpp  [options]  -o prog.exe  (C++)
  mpiifort  prog.f  [options]  -o prog.exe  (Fortran)
  ```
Running MPI Programs Interactively – Intel MPI

- For testing and developing MPI code only
- Total cores cannot exceed 4 on Eos and 8 on Ada, cumulatively.
- To run on multiple login nodes, passwordless ssh must be in place
  - Ada: set by default (don’t delete your $HOME/.ssh directory)
  - Eos: manual setup by users using /g/software/bin/init-ssh
Running MPI Programs Interactively - Intel MPI (cont.)

- Load the modules first
  
  \texttt{module load intel/2015B}

- Launch the mpi program with \textit{mpirun}
  
  \texttt{mpirun -np n [options] prog.exe [prog args]}

- Useful options
  
  \texttt{-ppn/-perhost}
  \texttt{-hosts}
  \texttt{-hostfile}
  \texttt{-h}
Demo Example 1

[pingluo@ada2 example1]$ mpirun -np 2 ./hello.exe
"Hello, world" from process 1 on node login2, total 2 processes
"Hello, world" from process 0 on node login2, total 2 processes

[pingluo@ada2 example1]$ mpirun -np 4 -hosts login1,login2 -ppn 2 ./hello.exe
"Hello, world" from process 0 on node login1, total 4 processes
"Hello, world" from process 1 on node login1, total 4 processes
"Hello, world" from process 2 on node login2, total 4 processes
"Hello, world" from process 3 on node login2, total 4 processes

[pingluo@ada2 example1]$ mpirun -hostfile ./host.list -np 4 ./a.out
"Hello, world" from process 1 on node login1, total 4 processes
"Hello, world" from process 2 on node login1, total 4 processes
"Hello, world" from process 3 on node login1, total 4 processes
"Hello, world" from process 0 on node login1, total 4 processes

[pingluo@ada2 example1]$ mpirun -np 4 -hostfile ./host.list -ppn 1 ./a.out
"Hello, world" from process 1 on node login2, total 4 processes
"Hello, world" from process 3 on node login4, total 4 processes
"Hello, world" from process 2 on node login3, total 4 processes
"Hello, world" from process 0 on node login1, total 4 processes
Running on Eos: Batch

- Sample batch script

```bash
#PBS -S /bin/bash
#PBS -l nodes=2:ppn=8,walltime=10:00:00,mem=1gb
#PBS -N hello
#PBS -j oe

module load intel/2015B
cd $PBS_O_WORKDIR
mpirun ./hello2.exe
#mpirun -np 16 -ppn 8 hello2.exe  # same as last line
#pbs.mpiexec ./hello2.exe  # This is the preferred way of
# launching Intel MPI tasks on Eos

- Submit a batch job

qsub hello.job
```
Running on Ada: Batch

- **Sample batch script**

  ```
  #BSUB -L /bin/bash
  #BSUB -n 20
  #BSUB -M 100 -W 20
  #BSUB -R 'rusage[mem=100] span[ptile=10]'
  #BSUB -J hello
  #BSUB -o hello.%J

  module load intel/2015B
  mpiexec.hydra ./hello2.exe
  ```

- **Submit a batch job**

  ```
  bsub < hello.job
  ```
MPI Process vs Processor

- A processor (or processor core, CPU core, core) is the smallest physical unit in a computer where a process runs.
- Usually, one MPI process will run on a single core.
  - Eos node: 8 or 12
  - Ada node: 20 or 40
- In some cases, less processes than number of cores may improve performance (undersubscription).
  - Because of better utilization of the memory subsystem
  - Test to find out
- For most applications, more processes than total number of cores is usually bad.
Viewing Processes On Processors

Run “top –u username”, press “f”, press “j”
Undersubscribe a Node on Eos

- Sample batch script

    #PBS -S /bin/bash
    #PBS -l nodes=2:ppn=8,walltime=10:00:00,mem=1gb
    #PBS -N hello
    #PBS -j oe

    module load intel/2015B
    cd $PBS_O_WORKDIR

    pbs.mpiexec -np 14 -ppn 7 ./test-mpi.exe
Undersubscribe a Node on Ada

Sample batch script

```bash
#BSUB -L /bin/bash
#BSUB -n 40
#BSUB -M 100 -W 20
#BSUB -R 'rusage[mem=100] span[ptile=20]'
#BSUB -J test-mpi
#BSUB -o test-mpi.%J

module load intel/2015B

export I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=0 # Needed to respect perhost request
mpiexec.hydra -np 36 -perhost 18 ./test-mpi.exe
```
Example 2

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

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

    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, MPI_STATUS_IGNORE);
        printf("Process %d receives %d from process 0\n", rank, number);
    }

    MPI_Finalize();
}
```

---

program simple
use mpi
implicit none
integer ierr, np, rank, inum;

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

call MPI_Finalize(ierr)

end program simple
Demo Example 2

Question: what’s the value of num before process 1 calls MPI_Recv?
what will happen if np > 2?
MPI processes communicate by passing messages.

Question: what’s the value of source and destination in a message?
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) constitutes an instance of a communicator that includes the whole universe of 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 & 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) 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) integer comm, rank, ierr</td>
</tr>
</tbody>
</table>
Exercise 1

- Modify hello.c/hello.f90 to print out the total number of processes activated in the program and the rank of each process.
- Compile and run the code.
MPI processes communicate by passing messages.

### Envelope
- Source
- Destination
- Tag
- Communicator

### Data Information
- Buffer Initial Address
- Count
- Datatype
MPI Tag

- Tag, an integer, is a message parameter that is used to differentiate a message.
- For any MPI implementation, tag can assume a specific range of integer values,

\[ 0 \leq \text{tag} \leq \text{MPI\_TAG\_UB} \]

The value of MPI\_TAG\_UB is implementation dependent.
MPI processes communicate by passing messages.

<table>
<thead>
<tr>
<th>Envelope</th>
<th>Data Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Buffer Initial Address</td>
</tr>
<tr>
<td>Destination</td>
<td>Count</td>
</tr>
<tr>
<td>Tag</td>
<td>Datatype</td>
</tr>
<tr>
<td>Communicator</td>
<td></td>
</tr>
</tbody>
</table>
**MPI Basic Datatypes**

- MPI basic datatypes are predefined, each corresponding to a data type in the host language.

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER (1 byte)</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL*4</td>
</tr>
<tr>
<td>MPI_REAL8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL (8 bytes)</td>
</tr>
</tbody>
</table>

Tables list most commonly used, but not all the basic MPI types in C/Fortran
Point-to-Point Communication

- **Blocking**: MPI_Send, MPI_Recv
- **Non-blocking**: MPI_Isend, MPI_Irecv
- **Send-Receive**: MPI_Sendrecv
**Blocking Send (Generic Interface)**

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

**IN buf**  initial address of send buffer
**IN count** number of elements in send buffer (nonnegative integer)
**IN datatype** datatype of each send buffer element
**IN dest** rank of destination
**IN tag** message tag
**IN comm** communicator
**Blocking Send (C/Fortran Binding)**

**Fortran Interface**

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

**C Interface**

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

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

- The calling process causes count 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 use the send buffer.
  - Safe means the message has been copied either into a system buffer, or directly into the receiver’s buffer.
Blocking Receive (Generic Interface)

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

<table>
<thead>
<tr>
<th>OUT</th>
<th>buf</th>
<th>initial address of receive buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in receive buer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each receive buer 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>status</td>
<td>status object</td>
</tr>
</tbody>
</table>

MPI_ANY_SOURCE and MPI_ANY_TAG are MPI defined (in include file) wildcards.
Blocking Receive (C/Fortran Binding)

Fortran Interface

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

C Interface

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *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 count many datatype elements.
  - An error returns if buf is smaller than data received.
Comments on Blocking Receive (cont.)

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

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

- The argument **status** in MPI receive provides a way of retrieving **message source**, **message tag**, and **message error**.

In Fortran, status is an array of integers

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

In C, status is typically an object of the structure type MPI_Status. It must be passed by reference.

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

- **status** is Useful when MPI wildcards are used.
- **status** can be **MPI_STATUS_IGNORED**: MPI_RECV(…, MPI_STATUS_IGNORED, …)
Example 3

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

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

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

Program ex3

```fortran
program ex3
use mpi
implicit none
integer rank, np, ierr, num, 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 *, num
    do i=1, np-1
        call MPI_SEND(num, 1, MPI_INTEGER, i, 0, &
                      MPI_COMM_WORLD, ierr)
    enddo
else
    call MPI_RECV(num, 1, MPI_INTEGER, 0, 0,
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
    print "(2(A,I6))","Process ",rank, " received ", num
endif
call MPI_FINALIZE(ierr)
end program ex3
```
Demo Example 3
Exercise 2

- Modify ex2.c/ex2.f90 to do the following:
  - Process 0 reads an integer from the user input
  - Process 0 sends the integer to process 1. Process 1 receives the integer and sends it to process 2, process 2 to process 3, and so on, until process n-1 also receives the integer.
Day 2
Point-to-Point Communication

- **Blocking**
  - MPI_Send, MPI_Recv
- **Non-blocking**
  - MPI_Isend, MPI_Irecv
- **Send-Receive**
  - MPI_Sendrecv
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
Non-Blocking Send (Generic Interface)

**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

- request is a handle associated with the outstanding send.
- It can be used later to query the status of the communication or wait for its completion.
Non-Blocking Send (C/Fortran Binding)

Fortran Interface

MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierr)
<type> buf(*)
integer count, datatype, dest, tag, comm, request, ierr

C Interface

int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Non-blocking Receive (Generic Interface)

MPI_Irecv(buf, count, datatype, source, 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 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>

Question: why there is no status compared with MPI_RECV
Non-blocking Receive (C/Fortran Binding)

Fortran Interface

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

C Interface

```c
int MPI_Irecv(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```
Auxiliary Routines for Isend/Ircev

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Wait</td>
<td>The calling process waits for the completion of a non-blocking send/receive identified by request.</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>The calling process waits for all pending operations in a list of requests.</td>
</tr>
<tr>
<td>MPI_Test</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>

Table lists some commonly used but not all completion routines for non-blocking communication.
**MPI_WAIT**

**MPI_WAIT(request, status)**

<table>
<thead>
<tr>
<th>INOUT</th>
<th>request</th>
<th>request (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>status</td>
<td>status object (Status)</td>
</tr>
</tbody>
</table>

Fortran Example:

```fortran
integer request
integer status(MPI_STATUS_SIZE)
...
call MPI_Irecv(recv_buf,count,...,comm,request,ierr)
... do some computations ...
call MPI_WAIT(request,status,ierr)
```

C Example:

```c
MPI_Request request;
MPI_Status status;
...
MPI_Irecv(recv_buf,count,...,comm,&request);
...do some computations ...
MPI_Wait(&request, &status);
```
MPI_WAITALL

MPI_WAITALL(count, requests, statuses)

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

Fortran Example:
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_ISEND(..., reqs(4), ierr)
...
... do some computations ...
...
call MPI_WAITALL(4, reqs, statuses, ierr)

C Example:
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 ...
...
call MPI_Waitall(4, reqs, statuses);
**MPI_TEST**

**MPI_TEST(request, flag, status)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INOUT</td>
<td>request communication request (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>flag true if operation completed (logical)</td>
</tr>
<tr>
<td>OUT</td>
<td>status status object (Status)</td>
</tr>
</tbody>
</table>

**Fortran example:**

```fortran
integer request
integer status(MPI_STATUS_SIZE)
logical flag

... call MPI_Irecv(recv_buf,count,...&
    comm,request,ierr)

do
    call MPI_TEST(request,flag,&
    status,ierr)
    if (flag == .FALSE.) then
        ... some computations ...
    else
        exit
    endif
endo
```

**C example:**

```c
MPI_Request request;
MPI_Status status;
_Bool flag;

... MPI_Irecv(recv_buf,count,...,
    comm,&request);

flag = 0;
for (;;){
    MPI_Test(&request,&flag,&status)
    if (flag == 0){
        do some computations ...
    }else
        break;
}
```
More On Return Status

- The auxiliary routines all return a status object or array of status objects.
- The status object is the same as the one returned by `MPI_RECV`.
  - You can extract the source and the tag of the message, or check the error code.
- Status objects can be ignored if not needed.
  - `MPI_STATUS_IGNORE`
  - `MPI_STATUSES_IGNORE`

EX: call `MPI_WAITALL(4, reqs, MPI_STATUSES_IGNORE, ierr)`
Timing Routine

**MPI_WTIME()**

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

**Fortran Interface**

```
DOUBLE PRECISION MPI_WTIME()

real*8 t1 t2
real*8 elapsed
t1 = MPI_WTIME()
...
! Stuff to be timed
...
t2 = MPI_WTIME()
elapsed = t2 - t1
```

**C Interface**

```
double MPI_Wtime(void)

double t1, t2;
double elapsed;
t1 = MPI_Wtime();
...
// stuff to be timed
...
t2 = MPI_Wtime();
elapsed = t2 - t1;
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv){
    int number, np, rank, i;
    MPI_Request *requests;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (np < 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);
        requests = (MPI_Request *)(malloc(sizeof(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);
    }
    MPI_Finalize();
}
program example4
use mpi
implicit none

integer rank, np, ierr, num, i
integer, allocatable:: requests(:)

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 *, num
   allocate(requests(np-1))
   do i=1, np-1
      call MPI_ISEND(num, 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(num, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
   print "(2(A,I6))", "Process ", rank, " received ", num
endif

call MPI_FINALIZE(ierr)
end program example4
Demo Example 4
Exercise 3

- Modify ex3.c/ex3.f90 to use non-blocking send/receive.
Point-to-Point Communication

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

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

- Combines send and receive operations in one call
  - Sends a message to one process.
  - Receives of another message from another process.
- The source and destination can be the same.
- The message sent out by Send-receive can be received by MPI_RECV/MPI_IRecv; Likewise, it can receive a message sent by MPI_SEND/MPI_Isend.
- 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.
Exercise 4

- Modify ex4.c/ex4.f90 to use MPI_sendrecv.
Important Note on Using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
Scheduling Methods

- Scheduling methods deal with task assignment.

- Types of scheduling:
  - Block distribution
  - Cyclic distribution
  - Block-cyclic distribution
  - Self-scheduling (used in the matvec_mult example)

Note: scheduling methods are not part of MPI specification.
Block Distribution

- Distribute \( n \) iterations to \( p \) processes. Each process takes a block of consecutive iterations.

<table>
<thead>
<tr>
<th>Iter id</th>
<th>Process id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

BLOCK_MAP(N1,N2,NPROCS,MYID,L1,L2)

IN N1 starting index of the loop
IN N2 ending index of the loop
IN NPROCS number of processes
IN MYID process rank
OUT L1 starting index of assigned loop block
OUT L2 ending index of assigned loop block
Block Distribution: block_map

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
```

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;
}
```
Cyclic Distribution

- Assign $n$ iterations to $p$ processes in a round-robin manner.

<table>
<thead>
<tr>
<th>Iter id</th>
<th>Process id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>1 2 3 1 2 3 1 2</td>
</tr>
</tbody>
</table>

```plaintext
do i = n1+myid, n2, nprocs
    ... do some computations...
    ... by process whose rank is myid...
end do
```

```plaintext
for (i=n1+myid; i<n2; i+= nprocs)
    ... do some computations...
    ... by process whose rank is myid...
}
```
**Block-Cyclic Distribution**

- Partitions $n$ iterations into chunks of size $m$ and assigns the $\left\lfloor \frac{n}{m} \right\rfloor$ chunks in a round-robin manner to $p$ processes.

<table>
<thead>
<tr>
<th>Iter id</th>
<th>Process id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

**Fortran**
```fortran
do i = n1+myid*BLK, n2, nprocs*BLK
   do j=i, min(n2,i+BLK-1)
      ... do some computations...
      ... by process whose rank is myid...
   enddo
end do
```

**C**
```c
for (i=n1+myid*BLK;i<=n2;i+=nprocs*BLK)
{
   for (j=i; j<min(n2,i+BLK-1);j++)
   {
      ... do some computations....
      ... by process whose rank is myid...
   }
}
```
Example 5: Calculate PI

\[ \pi = \int_0^1 \frac{4.0}{1 + x^2} \, dx \]
Demo Calc_PI
Exercise 5

- Modify ex5.c/ex5.f90 to use block distribution or cyclic distribution.
Collective Communication

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

<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></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></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></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>

Table lists most but not all of the collective communication operations.
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(p0)

p1

p2

p3
**Comments on MPI_BCAST**

- Root process: sends a message to all processes (including root) in `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.
Exercise 6

- Modify ex6.c/ex6.f90 to broadcast num from process 0 to all processes in MPI_COMM_WORLD.
Example 6: matvec_mult

\[ A \times b = \begin{pmatrix} \vdots & a_1 & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & a_n & \vdots \\ b_1 \\ \vdots \\ b_n \end{pmatrix} \times \begin{pmatrix} b_1 \\ \vdots \\ a_1 \times b_1 \\ \vdots \\ a_n \times b_n \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} = c \]

Each row \( a_i \) of \( A \) is a row vector. The dot product \( a_i \times b \) is the \( i \)th element of \( c \).

The algorithm distributes the computation of the dot product to any process. Each dot product is independent of the other.

Master process

“owns” \( A \) and \( b \), and collects \( c = A \times b \)

1. Broadcast \( b \) to each worker
2. Use tag to signify row number; send a row to each worker.
3. Loop:
   - receive a dot_product entry from worker_i
   - send a new row to worker_i from whom we just received the new dot_product;
   Until all dot_products are received.

Worker process

1. Receive \( b \)
2. Loop:
   - receive a new row from master
   - compute dot product
   - send dot product to master
   Until master sends termination notice.

Self scheduling: also call work load distribution scheme whereby a “master” process hands work to an idle worker process.
Demo matvec-mult

This program is used to demonstrate self-scheduling and MPI broadcast. It doesn’t take performance into consideration. In fact, it is very inefficient since for every data transfer (a send and a receive) between the master and a worker, very little computation is done (only one dot product). Better codes (still not very efficient) can be found in the demos/extra/MATVEC-multi-rows directory.
Day 3
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.
## 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_LONG</td>
</tr>
<tr>
<td>MPI_MIN, MPI_MAX</td>
<td>MPI_INTEGER, MPI_REAL8, MPI_INT, MPI_SHORT, MPI_LONG, MPI_DOUBLE</td>
</tr>
<tr>
<td>MPI_LAND, MPI_LOR, MPI_LXOR</td>
<td>MPI_LOGICAL, MPI_INTEGER, 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>

The table lists most but not all predefined operations. For each category of reduction operations, the table lists only a small subset of datatypes that the operations can apply to.
Exercise 7

- Modify ex7.c/ex7.f90 to use MPI_REDUCE to generate PI.
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.
- recvcnt is number of elements received per process
-Recvbuf, recvcnt, recvtype are significant only at root.
- On root, sendbuf and recvbuf cannot be the same.
- On root, sendbuf can be MPI_IN_PLACE: contribution of root to the gathered vector is already in the right place in recvbuf.
**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:
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:

<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>

<table>
<thead>
<tr>
<th>sendbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2345</td>
</tr>
<tr>
<td>3.1257</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>2.4345</td>
</tr>
<tr>
<td>7.0321</td>
</tr>
<tr>
<td>-0.2374</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>3.2478</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>recvbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
</tr>
<tr>
<td>1.2345</td>
</tr>
<tr>
<td>3.1257</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>recvtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
</tr>
<tr>
<td>P0</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0 sendbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2345</td>
</tr>
<tr>
<td>3.1257</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1 sendbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4345</td>
</tr>
<tr>
<td>7.0321</td>
</tr>
<tr>
<td>-0.2374</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2 sendbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2478</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>displs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
MPI_ALLGATHER/MPI_ALLGATHERV

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

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

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

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

Question: what's the difference between MPI_Scatter and MPI_Bcase?
MPI_SCATTERV

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

C:
Fortran:

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

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

displs

recv cnts

<table>
<thead>
<tr>
<th></th>
<th>1.2345</th>
<th>3.1257</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.4345</td>
<td>7.0321</td>
</tr>
<tr>
<td>3</td>
<td>7.0321</td>
<td>-0.2374</td>
</tr>
<tr>
<td>1</td>
<td>1.2345</td>
<td>3.1257</td>
</tr>
</tbody>
</table>

p0

p1

p2
Example 7: matvec-scatterv

\[ A \times b = \begin{pmatrix} a_1 & \cdots & a_n \end{pmatrix} \times \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = b_1 a_1 + b_2 a_2 + \cdots + b_n a_n = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} = c \]

\( a_i \) is a column vector.

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

\[ b_i a_i + \cdots + b_j a_j \]
Demo matvec-scatterv

Compare its running time with matvev_multi
MPI_ALLTOALL

**MPI_ALLTOALL**

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

- Each process in the communicator sends distinct data to all processes. The **jth block** sent from process **i** is received by process **j** and is placed in the **ith block** of **recvbuf**.
- Sendcnt and recvcnt refer to the block size.
## Example 8: Matrix Transpose

### C: row major

**4 x 4 matrix**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

### Fortran: column major

**1 x 4 matrix**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

### Transpose

**p0**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

**p1**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

**p0**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

**p1**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Example 8: Matrix Transpose

C

<table>
<thead>
<tr>
<th>C</th>
<th>p0 sendbuf</th>
<th>p1 sendbuf</th>
<th>p0 recvbuf</th>
<th>p1 recvbuf</th>
<th>p0 recvbuf</th>
<th>p1 recvbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2</td>
<td>3 4</td>
<td>1 2</td>
<td>9 10</td>
<td>1 5</td>
<td>9 10</td>
</tr>
<tr>
<td></td>
<td>5 6</td>
<td>7 8</td>
<td>5 6</td>
<td>13 14</td>
<td>2 6</td>
<td>10 14</td>
</tr>
<tr>
<td></td>
<td>9 10</td>
<td>11 12</td>
<td>3 4</td>
<td>11 12</td>
<td>3 7</td>
<td>11 15</td>
</tr>
<tr>
<td></td>
<td>13 14</td>
<td>15 16</td>
<td>7 8</td>
<td>15 16</td>
<td>4 8</td>
<td>12 16</td>
</tr>
</tbody>
</table>

Fortran

<table>
<thead>
<tr>
<th>Fortran</th>
<th>p0 sendbuf</th>
<th>p0 recvbuf</th>
<th>p0 recvbuf</th>
<th>p1 recvbuf</th>
<th>p1 recvbuf</th>
<th>p1 recvbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>1 5</td>
<td>9 13</td>
<td>1 2</td>
<td>3 4</td>
<td>1 5</td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>2 6</td>
<td>10 14</td>
<td>3 4</td>
<td>8</td>
<td>5 6</td>
<td>7 8</td>
</tr>
<tr>
<td></td>
<td>3 7</td>
<td>11 15</td>
<td>9 13</td>
<td>11 15</td>
<td>9 10</td>
<td>11 12</td>
</tr>
<tr>
<td></td>
<td>4 8</td>
<td>12 16</td>
<td>10 14</td>
<td>12 16</td>
<td>13 14</td>
<td>15 16</td>
</tr>
</tbody>
</table>

Each block transpose locally

MPI_Alltoall
Demo transpose
Example 9: Solving 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.

Decompose the domain along x and y using n internal points in each direction.

The increment

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

\[ x_i = ih, y_j = jh \]

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

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

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

5-point finite difference stencil approximation:

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

\[ u_{i,j} = 0.25 \times (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}) \]
Example 9: Solving x-y Poisson Equation

5-point finite difference stencil approximation

\[ u_{i,j}^{k+1} = \frac{1}{4} \left( u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k + u_{i+1,j}^k - h^2 f_{i,j} \right) \]

**Boundary Conditions (Stay fixed):**

- \( u(x_i,0) = \frac{\cos(\pi x_i) - \pi^2}{\pi^2} \) for \( 0 \leq x \leq 1 \)
- \( u(x_i,1) = \frac{\cos(\pi x_i) - (\pi^2) \cos(x_i)}{\pi^2} \) for \( 0 \leq x \leq 1 \)
- \( u(0,y_j) = \frac{1}{\pi^2} - 1 \) for \( 0 \leq y \leq 1 \)
- \( u(1,y_j) = -\frac{1}{\pi^2 + \cos(y_j)} \) for \( 0 \leq y \leq 1 \)

**RHS:** \( f(x_i,y_j) = (x_i^2 + y_j^2) \cos(x_i y_j) - \cos(\pi x_i) \)

**Jacobi iteration across all points:**

```plaintext
do j=1, n
   do i=1, n
      unew(i,j) = 0.25* ( u(i-1,j) + u(i,j+1) + u(i,j-1) + u(i+1,j) ) - f(i,j)*h^2
   end do
end do
```
Showing border columns that will have to be copied into the address space of a neighbor process during a (5-point) finite-difference iteration scheme (jacobi) used in the p_jacobild program to solve the x-y Poisson equation.
COLUMN-WISE BLOCK DISTRIBUTION: POISSON PROBLEM
Column Exchanges Needed for Parallel 5-point FDM Jacobi Iteration

\[ u(i,j) \]

Showing exchanges of border columns per process. A border column from one process is copied into a "ghost" column of a neighbor's address space.
Demo poisson_1d
#pragma omp parallel for private(x) reduction(+:sum)
for (i=l1; i<=l2; i++){
    x = h*(i-0.5);
    sum += 4.0/(1.0+x*x);
}

$ mpiicc -openmp -xhost -O3 calc_PI.c -o calc_PI.exe
$ export OMP_NUM_THREADS=2
$ mpirun -np 2 ./calc_PI.exe  (run with two MPI processes, each with 2 threads)

 !$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:s)
do i=l1, l2
    x = h*(i-0.5d0)
    s = s + 4.0d0/(1.0d0+x*x)
enddo

$ mpiifort -openmp -xhost -O3 calc_PI.f90 -o calc_PI.exe
$ export OMP_NUM_THREADS=2
$ mpirun -np 2 ./calc_PI.exe

You can also set affinity to control thread placement. See https://sc.tamu.edu/wiki/index.php/Ada:Compile:MPI#Hybrid_MPI.2FOpenMP_Code

Source code: /g/public/training/mpi/Fall2015/demos/extra/hybrid_cal_PI
Course Materials

- All files are located on Ada at 
  /general/public/training/mpi/Fall2015
  - ./demos: source codes for demos
  - ./exercises: skeleton codes for exercises
  - ./slides: presentation slides and supplementary materials
  - ./solutions: solutions to the exercises

- Also in $SCRATCH/mpi-training
  - After running setup-mpi-training.sh
References

- https://computing.llnl.gov/tutorials/mpi/