INTRODUCTION TO MPI
Message-passing interface

MPI is an application programming interface (API) for communication between separate processes
  – The most widely used approach for distributed parallel computing

MPI programs are portable and scalable

MPI is flexible and comprehensive
  – Large (hundreds of procedures)
  – Concise (often only 6 procedures are needed)

MPI standardization by MPI Forum
Execution model

- Parallel program is launched as set of independent, identical processes
- The same program code and instructions
- Can reside in different nodes
  - or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, srun, aprun, poe, ...
MPI ranks

MPI runtime assigns each process a rank
  – identification of the processes
  – ranks start from 0 and extent to N-1

Processes can perform different tasks and handle different data basing on their rank

```c
... if ( rank == 0 ) {
    ...
} if ( rank == 1 ) {
    ...
}
...```
Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages

```
a = 1.0
b = 2.0
```

```
a = -1.0
b = -2.0
```

MPI

Process 1 (rank 0)

Process 2 (rank 1)
MPI communicator

- Communicator is an object connecting a group of processes
- Initially, there is always a communicator MPI_COMM_WORLD which contains all the processes
- Most MPI functions require communicator as an argument
- Users can define own communicators
Routines of the MPI library

- Information about the communicator
  - number of processes
  - rank of the process

- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes

- Synchronization between processes

- Advanced features
Programming MPI

MPI standard defines interfaces to C and Fortran programming languages
  – There are unofficial bindings to Python, Perl and Java

C call convention
  \[ rc = \text{MPI}_\text{Xxxx}(\text{parameter},...) \]
  – some arguments have to be passed as pointers

Fortran call convention
  \[ \text{CALL MIPS}_\text{XXXX}(\text{parameter},..., \text{rc}) \]
  – return code in the last argument
First five MPI commands

Set up the MPI environment

MPI_Init()

Information about the communicator

MPI_Comm_size(comm, size)
MPI_Comm_rank(comm, rank)

– Parameters

  comm   communicator
  size   number of processes in the communicator
  rank   rank of this process
First five MPI commands

- Synchronize processes
  `MPI_Barrier(comm)`
- Finalize MPI environment
  `MPI_Finalize()`
Writing an MPI program

- Include MPI header files
  
  **C:**
  ```
  #include <mpi.h>
  ```
  
  **Fortran:**
  ```
  use mpi
  ```

- Call MPI_Init

- Write the actual program

- Call MPI_Finalize before exiting from the main program
Summary

In MPI, a set of independent processes is launched
  – Processes are identified by ranks
  – Data is always local to the process

Processes can exchange data by sending and receiving messages

MPI library contains functions for
  – Communication and synchronization between processes
  – Communicator manipulation
POINT-TO-POINT COMMUNICATION
Introduction

- MPI processes are independent, they communicate to coordinate work
- Point-to-point communication
  - Messages are sent between two processes
- Collective communication
  - Involving a number of processes at the same time
MPI point-to-point operations

One process *sends* a message to another process that *receives* it

Sends and receives in a program should match – one receive per send
MPI point-to-point operations

Each message (envelope) contains

- The actual *data* that is to be sent
- The *datatype* of each element of data.
- The *number of elements* the data consists of
- An identification number for the message (*tag*)
- The ranks of the *source* and *destination* process
Presenting syntax

Send operation

**MPI_Send**(buf, count, datatype, dest, tag, comm)

- **buf**: The data that is sent
- **count**: Number of elements in buffer
- **datatype**: Type of each element in buf (see later slides)
- **dest**: The rank of the receiver
- **tag**: An integer identifying the message
- **comm**: A communicator
- **error**: Error value; in C/C++ it’s the return value of the function, and in Fortran an additional output parameter

Note! Extra error parameter for Fortran

Operations presented in pseudocode, C and Fortran bindings presented in extra material slides.

INPUT arguments in red

OUTPUT arguments in blue

Slide with extra material included in handouts
Send operation

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

buf        The data that is sent
count      Number of elements in buffer
datatype   Type of each element in buf (see later slides)
dest       The rank of the receiver
tag        An integer identifying the message
comm       A communicator
error      Error value; in C/C++ it’s the return value of the function, and in Fortran an additional output parameter
Receive operation

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

<table>
<thead>
<tr>
<th>buf</th>
<th>Buffer for storing received data</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of elements in buffer, not the number of elements that are actually received</td>
</tr>
<tr>
<td>datatype</td>
<td>Type of each element in buf</td>
</tr>
<tr>
<td>source</td>
<td>Sender of the message</td>
</tr>
<tr>
<td>tag</td>
<td>Number identifying the message</td>
</tr>
<tr>
<td>comm</td>
<td>Communicator</td>
</tr>
<tr>
<td>status</td>
<td>Information on the received message</td>
</tr>
<tr>
<td>error</td>
<td>As for send operation</td>
</tr>
</tbody>
</table>
MPI datatypes

MPI has a number of predefined datatypes to represent data

Each C or Fortran datatype has a corresponding MPI datatype

- C examples: MPI_INT for int and MPI_DOUBLE for double
- Fortran example: MPI_INTEGER for integer

One can also define custom datatypes
Case study: parallel sum

- Array originally on process #0 (P0)

Parallel algorithm
- Scatter
  - Half of the array is sent to process 1
- Compute
  - P0 & P1 sum independently their segments
- Reduction
  - Partial sum on P1 sent to P0
  - P0 sums the partial sums
Case study: parallel sum

Step 1.1: Receive operation in scatter

P1 posts a receive to receive half of the array from P0
Case study: parallel sum

Step 1.2: Send operation in scatter

P0 posts a send to send the lower part of the array to P1
Case study: parallel sum

Step 2: Compute the sum in parallel

P0 & P1 computes their parallel sums and store them locally
Case study: parallel sum

Step 3.1: Receive operation in reduction

P0 posts a receive to receive partial sum
Case study: parallel sum

Step 3.2: send operation in reduction

P1 posts a send with partial sum
**Case study: parallel sum**

Step 4: Compute final answer

P0 sums the partial sums
MORE ABOUT POINT-TO-POINT COMMUNICATION
Blocking routines & deadlocks

**Blocking routines**
- Completion depends on other processes
- Risk for deadlocks – the program is stuck forever

**MPI_Send** exits once the send buffer can be safely read and written to

**MPI_Recv** exits once it has received the message in the receive buffer
Point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data
Combined send & receive

\[ \text{MPI}_\text{Sendrecv}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{dest}, \text{sendtag}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{source}, \text{recvtag}, \text{comm}, \text{status}) \]

- Parameters as for \text{MPI}_\text{Send} and \text{MPI}_\text{Recv} combined
- Sends one message and receives another one, with one single command
  - Reduces risk for deadlocks
- Destination rank and source rank can be same or different
Case study 2: Domain decomposition

Computation inside each domain can be carried out independently; hence in parallel

*Ghost layer* at boundary represent the value of the elements of the other process
Case study 2: One iteration step

Have to carefully schedule the order of sends and receives in order to avoid deadlocks.
Case study 2: MPI_Sendrecv

MPI_Sendrecv

- Sends and receives with one command
- No risk of deadlocks
Special parameter values

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>MPI_PROC_NULL</td>
<td>Null destination, no operation takes place</td>
</tr>
<tr>
<td>comm</td>
<td>MPI_COMM_WORLD</td>
<td>Includes all processes</td>
</tr>
<tr>
<td>error</td>
<td>MPI_SUCCESS</td>
<td>Operation successful</td>
</tr>
</tbody>
</table>
Special parameter values

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>MPI_PROC_NULL</td>
<td>No sender, no operation takes place</td>
</tr>
<tr>
<td></td>
<td>MPI_ANY_SOURCE</td>
<td>Receive from any sender</td>
</tr>
<tr>
<td>tag</td>
<td>MPI_ANY_TAG</td>
<td>Receive messages with any tag</td>
</tr>
<tr>
<td>comm</td>
<td>MPI_COMM_WORLD</td>
<td>Includes all processes</td>
</tr>
<tr>
<td>status</td>
<td>MPI_STATUS_IGNORE</td>
<td>Do not store any status data</td>
</tr>
<tr>
<td>error</td>
<td>MPI_SUCCESS</td>
<td>Operation successful</td>
</tr>
</tbody>
</table>
Status parameter

The status parameter in MPI_Recv contains information on how the receive succeeded

- Number and datatype of received elements
- Tag of the received message
- Rank of the sender

In C the status parameter is a struct, in Fortran it is an integer array
Status parameter

Received elements
Use the function
\texttt{MPI\_Get\_count(status, datatype, count)}

Tag of the received message
\begin{itemize}
  \item C: \hspace{0.5cm} \texttt{status.MPI\_TAG}
  \item Fortran: \hspace{0.5cm} \texttt{status(MPI\_TAG)}
\end{itemize}

Rank of the sender
\begin{itemize}
  \item C: \hspace{0.5cm} \texttt{status.MPI\_SOURCE}
  \item Fortran: \hspace{0.5cm} \texttt{status(MPI\_SOURCE)}
\end{itemize}
Summary

- Point-to-point communication
  - Messages are sent between two processes
- We discussed send and receive operations enabling any parallel application
  - MPI_Send & MPI_Recv
  - MPI_Sendrecv
- Special argument values
- Status parameter
COLLECTIVE OPERATIONS
Outline

- Introduction to collective communication
- One-to-many collective operations
- Many-to-one collective operations
- Many-to-many collective operations
- Non-blocking collective operations
- User-defined communicators
Introduction

Collective communication transmits data among all processes in a process group
- These routines must be called by all the processes in the group

Collective communication includes
- data movement
- collective computation
- synchronization

Example

MPI_Barrier makes each task hold until all tasks have called it
int MPI_Barrier(comm)
MPI_BARRIER(comm, rc)
Collective communication outperforms normally point-to-point communication.

Code becomes more compact and easier to read:

```fortran
if (my_id == 0) then
    do i = 1, ntasks-1
        call mpi_send(a, 1048576, &
                     MPI_REAL, i, tag, &
                     MPI_COMM_WORLD, rc)
    end do
else
    call mpi_recv(a, 1048576, &
                   MPI_REAL, 0, tag, &
                   MPI_COMM_WORLD, status, rc)
end if
```

Communicating a vector `a` consisting of 1M float elements from the task 0 to all other tasks.
Introduction

- Amount of sent and received data must match
- Non-blocking routines are available in the MPI 3 standard
  - Older libraries do not support this feature
- No tag arguments
  - Order of execution must coincide across processes
Broadcasting

Send the same data from one process to all the other

This buffer may contain multiple elements of any datatype.
Broadcasting

With MPI_Bcast, the task \textit{root} sends a \textit{buffer} of data to all other tasks

\texttt{MPI\_Bcast(buffer, count, datatype, root, comm)}

- \texttt{buffer} data to be distributed
- \texttt{count} number of entries in buffer
- \texttt{datatype} data type of buffer
- \texttt{root} rank of broadcast root
- \texttt{comm} communicator
Scattering

Send equal amount of data from one process to others

Segments A, B, ... may contain multiple elements
Scattering

MPI_Scatter: Task root sends an equal share of data (sendbuf) to all other processes

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>send buffer (data to be scattered)</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements sent to each process</td>
</tr>
<tr>
<td>sendtype</td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td>recvbuf</td>
<td>receive buffer</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements in receive buffer</td>
</tr>
<tr>
<td>recvtype</td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td>root</td>
<td>rank of sending process</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>
One-to-all example

Assume 4 MPI tasks. What would the (full) program print?

if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if

call mpi_scatter(a,4,MPI_INTEGER, &
  aloc,4,MPI_INTEGER, &
  0,MPI_COMM_WORLD,rc)
if (my_id==3) print *, a(:)

if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if

call mpi_bcast(a,16,MPI_INTEGER,0, &
  MPI_COMM_WORLD,rc)
if (my_id==3) print *, a(:)

A. 1 2 3 4
B. 13 14 15 16
C. 1 2 3 4
   5 6 7 8
   9 10 11 12
   13 14 15 16
Varying-sized scatter

Like MPI_Scatter, but messages can have different sizes and displacements.

MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

- `sendbuf`: send buffer
- `sendcounts`: array (of length ntasks) specifying the number of elements to send to each processor
- `displs`: array (of length ntasks). Entry i specifies the displacement (relative to sendbuf)
- `sendtype`: data type of send buffer elements
- `recvbuf`: receive buffer
- `recvcount`: number of elements in receive buffer
- `recvtype`: data type of receive buffer elements
- `root`: rank of sending process
- `comm`: communicator
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
  sendcnts = (/ 1, 2, 3, 4 /)
  displs = (/ 0, 1, 3, 6 /)
end if

call mpi_scatterv(a, sendcnts, &
    displs, MPI_INTEGER,&
    alloc, 4, MPI_INTEGER, &
    0, MPI_COMM_WORLD, rc)

Assume 4 MPI tasks. What are the values in alloc in the last task (#3)?
Gathering

Collect data from all the process to one process

Segments A, B, ... may contain multiple elements
Gathering

MPI_Gather: Collect equal share of data (in sendbuf) from all processes to root

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

sendbuf: send buffer (data to be gathered)
sendcount: number of elements pulled from each process
sendtype: data type of send buffer elements
recvbuf: receive buffer
recvcount: number of elements in any single receive
recvtype: data type of receive buffer elements
root: rank of receiving process
comm: communicator
Varying-sized gather

Like MPI_Gather, but messages can have different sizes and displacements

```c
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
```

- `sendbuf`: send buffer
- `sendcount`: the number of elements to send
- `sendtype`: data type of send buffer elements
- `recvbuf`: receive buffer
- `recvcounts`: array (of length ntasks). Entry i specifies how many to receive from that process
- `displs`: array relative to recvcounts, displacement in recvbuf
- `recvtype`: data type of receive buffer elements
- `root`: rank of receiving process
- `comm`: communicator
Reduce operation

Applies an operation over set of processes and places result in single process

\[
\begin{array}{cccc}
P_0 & A_0 & B_0 & C_0 & D_0 \\
P_1 & A_1 & B_1 & C_1 & D_1 \\
P_2 & A_2 & B_2 & C_2 & D_2 \\
P_3 & A_3 & B_3 & C_3 & D_3 \\
\end{array}
\]

\[
\begin{array}{cccc}
P_0 & \Sigma A_i & \Sigma B_i & \Sigma C_i & \Sigma D_i \\
P_1 & & & & \\
P_2 & & & & \\
P_3 & & & & \\
\end{array}
\]

REDUCE (SUM)
Reduce operation

Applies a reduction operation \textit{op} to \textit{sendbuf} over the set of tasks and places the result in \textit{recvbuf} on \textit{root}

\texttt{MPI	extunderscore Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)}

- \texttt{sendbuf}: send buffer
- \texttt{recvbuf}: receive buffer
- \texttt{count}: number of elements in send buffer
- \texttt{datatype}: data type of elements in send buffer
- \texttt{op}: operation
- \texttt{root}: rank of root process
- \texttt{comm}: communicator
Global reduce operation

MPI_Allreduce combines values from all processes and distributes the result back to all processes

- Compare: MPI_Reduce + MPI_Bcast

\[
\text{MPI\_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)}
\]

- **sendbuf**: starting address of send buffer
- **recvbuf**: starting address of receive buffer
- **count**: number of elements in send buffer
- **datatype**: data type of elements in send buffer
- **op**: operation
- **comm**: communicator

\[
\begin{align*}
\Sigma A_i & \rightarrow B \quad \Sigma C_i \rightarrow D_i \\
\Sigma B_i & \rightarrow C \\
\Sigma C_i & \rightarrow D_i
\end{align*}
\]
Allreduce example: parallel dot product

```
real :: a(1024), aloc(128)
...
if (my_id==0) then
  call random_number(a)
end if
call mpi_scatter(a, 128, MPI_INTEGER, &
    aloc, 128, MPI_INTEGER, &
    0, MPI_COMM_WORLD, rc)
rloc = dot_product(aloc,aloc)
call mpi_allreduce(rloc, r, 1, MPI_REAL,&
    MPI_SUM, MPI_COMM_WORLD, rc)
```

```bash
> aprun -n 8 ./mpi_pdot
id= 6 local= 39.68326  global= 338.8004
id= 7 local= 39.34439  global= 338.8004
id= 1 local= 42.86630  global= 338.8004
id= 3 local= 44.16300  global= 338.8004
id= 5 local= 39.76367  global= 338.8004
id= 0 local= 42.85532  global= 338.8004
id= 2 local= 40.67361  global= 338.8004
id= 4 local= 49.45086  global= 338.8004
```
MPI_Allgather gathers data from each task and distributes the resulting data to each task

- Compare: MPI_Gather + MPI_Bcast

MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- sendbuf: send buffer
- sendcount: number of elements in send buffer
- sendtype: data type of send buffer elements
- recvbuf: receive buffer
- recvcount: number of elements received from any process
- recvtype: data type of receive buffer
From each to every

Send a distinct message from each task to every task

"Transpose" like operation
From each to every

MPI_Alltoall sends a distinct message from each task to every task

- Compare: “All scatter”

```c
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
```

- sendbuf: send buffer
- sendcount: number of elements to send to each process
- sendtype: data type of send buffer elements
- recvbuf: receive buffer
- recvcount: number of elements received from any process
- recvtype: data type of receive buffer elements
- comm: communicator
Assume 4 MPI tasks. What will be the values of `aloc` in the process #0?

A. 1, 2, 3, 4
B. 1,...,16
C. 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4
Common mistakes with collectives

✘ Using a collective operation within one branch of an if-test of the rank

\[
\text{IF (my_id == 0) CALL MPI_BCAST(...) }
\]

– All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), must call the collective routine!

✘ Assuming that all processes making a collective call would complete at the same time

✘ Using the input buffer as the output buffer

\[
\text{CALL MPI_ALLREDUCE(a, a, n, MPI_REAL, MPI_SUM, ...)
}
Summary

Collective communications involve all the processes within a communicator
  – All processes must call them

Collective operations make code more transparent and compact

Collective routines allow optimizations by MPI library

Performance consideration:
  – Alltoall is expensive operation, avoid it when possible
USER-DEFINED COMMUNICATORS
Communicators

The communicator determines the "communication universe"
- The source and destination of a message is identified by process rank within the communicator

So far: MPI_COMM_WORLD

Processes can be divided into subcommunicators
- Task level parallelism with process groups performing separate tasks
- Parallel I/O
Communicators

Communicators are dynamic

A task can belong simultaneously to several communicators
- In each of them it has a unique ID, however
- Communication is normally within the communicator
Grouping processes in communicators

MPI_COMM_WORLD

Comm 1

Comm 2

Comm 3
Creating a communicator

MPI_Comm_split creates new communicators based on 'colors' and 'keys'

MPI_Comm_split(comm, color, key, newcomm)

comm: communicator handle
color: control of subset assignment, processes with the same color belong to the same new communicator
key: control of rank assignment
newcomm: new communicator handle

If color = MPI_UNDEFINED, a process does not belong to any of the new communicators
Creating a communicator

```c
if (myid%2 == 0) {
    color = 1;
} else {
    color = 2;
}
MPI_Comm_split(MPI_COMM_WORLD, color, myid, &subcomm);
MPI_Comm_rank(subcomm, &mysubid);
printf("I am rank %d in MPI_COMM_WORLD, but %d in
    Comm %d.\n", myid, mysubid, color);
```

I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
Communicator manipulation

**MPI_Comm_size**
Returns number of processes in communicator's group

**MPI_Comm_rank**
Returns rank of calling process in communicator's group

**MPI_Comm_compare**
Compares two communicators

**MPI_Comm_dup**
Duplicates a communicator

**MPI_Comm_free**
Marks a communicator for deallocation
Basic MPI summary

- User-defined communicators
- Point-to-point communication
  - Sendrecv
  - Send &Recv
- Collective communication
  - One-to-all collectives
  - All-to-one collectives
  - All-to-all collectives