Outlines

- Basics of MPI
- How to compile and execute MPI programs?
- MPI library calls used in Example program
- MPI point-to-point communication
- MPI advanced point-to-point communication
- MPI Collective Communication and Computations
- MPI Datatypes
- MPI Communication Modes
- MPI special features
Is MPI Large or Small?

The MPI Message Passing Interface Small or Large

MPI can be small

One can begin programming with 6 MPI function calls

- MPI_INIT: Initializes MPI
- MPI_COMM_SIZE: Determines number of processors
- MPI_COMM_RANK: Determines the label of the calling process
- MPI_SEND: Sends a message
- MPI_RECV: Receives a message
- MPI_FINALIZE: Terminates MPI

MPI can be large

One can utilize any of 125 functions in MPI.

MPI Blocking Send and Receive

Blocking Send

A typical blocking send looks like

\[ \text{send (dest, type, address, length)} \]

Where

- **dest** is an integer identifier representing the process to receive the message
- **type** is nonnegative integer that the destination can use to selectively screen messages
- **(address, length)** describes a contiguous area in memory containing the message to be sent
MPI Blocking Send and Receive

Point-to-Point Communications

The sending and receiving of messages between pairs of processors.

- **BLOCKING SEND**: returns only after the corresponding RECEIVE operation has been issued and the message has been transferred.

  \[\text{MPI\_Send}\]

- **BLOCKING RECEIVE**: returns only after the corresponding SEND has been issued and the message has been received.

  \[\text{MPI\_Recv}\]

If we are sending a large message, most implementations of blocking send and receive use the following procedure.

\[
\begin{align*}
S &= \text{Sender} \\
R &= \text{Receiver}
\end{align*}
\]

\[
\begin{align*}
\text{MPI\_SEND} & & \text{(blocking standard send)} \\
\text{size} & > \text{threshold} & \text{data transfer from source complete} \\
\text{task waits} & & \text{wait} \\
S & & R
\end{align*}
\]

Transfer doesn't begin until word has arrived that corresponding \text{MPI\_RECV} has been posted.

\[
\begin{align*}
\text{MPI\_RECV} & & \text{task continues when data transfer to user's buffer is complete}
\end{align*}
\]
**MPI Non-Blocking Send and Receive**

**Non-blocking Receive:** does not wait for the message transfer to complete, but immediate returns control back to the calling processor.

```
MPI_IRecv
```

**C**

```
MPI_Irecv (buf, count, dtype, dest, tag, comm, request);
```

**Fortran**

```
MPI_Irecv (buf, count, dtype, dest, tag, comm, request, ierror);
```

**Non-Blocking Communications**

- Separate communication into three phases:
  - Initiate non-blocking communication.
  - Do some work (perhaps involving other communications?)
  - Wait for non-blocking communication to complete.
If we are sending a small message, most implementations of non-blocking sends and receive use the following procedure. The message can be sent immediately and stored in a buffer on the receiving side.

S = Sender  R = Receiver

An MPI-Wait checks to see if a non-blocking operation has completed. In this case, the MPI_Wait on the sending side believes the message has already been received.

MPI Non-Blocking Send and Receive

MPI_ISEND (non-blocking standard send)
size ≤ threshold  MPI_WAIT

Transfer to buffer on receiving node can be avoided if
MPI_RECV posted early enough

MPI_RECV
no delay if MPI_WAIT is late enough

MPI_WAIT
If we are sending a large message, most implementations of non-blocking sends and receive use the following procedure. The send is issued, but the data is not immediately sent. Computation is resumed after the send, but later halted by an MPI_Wait.

S = Sender    R = Receiver

An MPI_Wait checks to see if a non-blocking operation has completed. In this case, the MPI_Wait on the sending side sees that the message has not been sent yet.

MPI_Communications Modes

- **Synchronous mode**
  - The same as standard mode, except the send will not complete until message delivery is guaranteed

- **Buffered mode**
  - Similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this
If we the programmer allocate some memory (buffer space) for temporary storage on the sending processor, we can perform a type of non-blocking send.

\[ S = \text{Sender} \quad R = \text{Receiver} \]

MPI Buffered Send and Receive

Sender mode | Notes
---|---
Synchronous send | Only completes when the receive has completed
Buffered send | Always completes (unless an error occurs), irrespective of receiver.
Standard send | Either synchronous or buffered.
Ready send | Always completes (unless an error occurs), irrespective of whether the receive has completed.
Receive | Completes when a message has arrived.

MPI Communication Modes

MPI_BSEND (buffered send)

S

MPI_RECV

R

data transfer to user-supplied buffer complete

copy data to buffer

task waits
**MPI Communication Modes**

**MPI Sender Modes**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>MPI CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI_SEND</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI_SSEND</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI_BSEND</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI_RSEND</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_RECV</td>
</tr>
</tbody>
</table>

**MPI Datatypes**

**Message type**

- A message contains a number of elements of some particular datatype
- MPI datatypes:
  - Basic types
  - Derived types - Vector, Struct, Others
- Derived types can be built up from basic types
- C types are different from Fortran types
Contiguous Data

- The simplest derived datatype consists of a number of contiguous items of the same datatype

- C:
  ```c
  int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
  ```

- Fortran:
  ```fortran
  MPI_Type_contiguous (count, oldtype, newtype)
  integer count, oldtype, newtype
  ```
### Constructing a Vector Datatype

- **C**
  ```c
  int MPI_Type_vector (int count, int blocklength, int stride,
                      MPI_Datatype oldtype, MPI_Datatype *newtype);
  ```

- **Fortran**
  ```fortran
  MPI_Type_vector (count, blocklength, stride, oldtype, newtype, ierror)
  ```

### Extent of a Datatype

- **C**
  ```c
  int MPI_Type_extent (MPI_Datatype datatype, int *extent);
  ```

- **Fortran**
  ```fortran
  MPI_Type_extent (datatype, extent, ierror)
  ```

  integer datatype, extent, ierror

---

### Struct Datatype Example

- `count = 2`
- `array_of_blocklengths[0] = 1`
- `array_of_types[0] = MPI_INT`
- `array_of_blocklengths[1] = 3`
- `array_of_types[1] = MPI_DOUBLE`
Constructing a Struct Datatype

- C:
  ```c
  int MPI_Type_Struct (int count, int array_of_blocklengths,
                      MPI_Aint *array_of_displacements,
                      MPI_Datatype *array_of_types,
                      MPI_Datatype *newtype);
  ```

- Fortran:
  ```fortran
  MPI_Type_Struct (count, array_of_blocklengths,
                   array_of_displacements, array_of_types, newtype, ierror)
  ```

Committing a datatype

- Once a datatype has been constructed, it needs to be committed before it is used.

- This is done using MPI_TYPE_COMMIT

- C
  ```c
  int MPI_Type_Commit (MPI_Datatype *datatype);
  ```

- Fortran
  ```fortran
  MPI_Type_Commit (datatype, ierror)
  ```
MPI - Using topology

**MPI : Support for Regular Decompositions**

- Using topology routines
  - "MPI_Cart_Create"
  - User can define virtual topology
- Why you use the topology routines
  - "Simple to use (why not?)"
  - "Allow MPI implementation to provide low expected contention layout of processes (contention can matter)"
  - "Remember, contention still matters; a good mapping can reduce contention effects"

---

MPI Persistent Communication

**MPI : Nonblocking operations, overlap effective**

- Isend, Irecv, Waitall

**MPI : Persistent Operations**

- Potential saving
  - "Allocation of MPI_Request"
- Variation of example
  - "sendinit, recvinit, startall, waitall"
  - "startall(recevs), sendrecv/barrier, startall(redits), waitall"
- Vendor implementations are buggy
Collective Communications

The sending and/or receiving of messages to/from groups of processors. A collective communication implies that all processors need participate in the communication.

- Involves coordinated communication within a group of processes
- No message tags used
- All collective routines block until they are locally complete
- Two broad classes:
  - Data movement routines
  - Global computation routines

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.
Characteristics of Collective Communication

- Collective action over a communicator
- All processes must communicate
- Synchronization may or may not occur
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size

Communication is coordinated among a group of processes

- Group can be constructed “by hand” with MPI group-manipulation routines or by using MPI topology-definition routines
- Different communicators are used instead
- No non-blocking collective operations

Collective Communication routines - Three classes

- Synchronization
- Data movement
- Collective computation
# MPI Collective Communications

## Barrier

A barrier insures that all processors reach a specified location within the code before continuing.

- C:
  
  ```c
  int MPI_Barrier (MPI_Comm comm);
  ```

- Fortran:
  
  ```fortran
  MPI_barrier (comm, ierror)
  integer comm, ierror
  ```

## Broadcast

A broadcast sends data from one processor to all other processors.

- C:
  
  ```c
  int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
  ```

- Fortran:
  
  ```fortran
  MPI_bcast (buffer, count, datatype, root, comm, ierror)
  <type> buffer(*)
  integer count, datatype, root, comm, ierror
  ```
Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.

- Examples:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

MPI Collective Computations

Fortran

    MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
    <type> sendbuf (*), recvbuf (*)
    integer count, datatype, op, root, comm,
    integer ierror

C

    int MPI_Reduce (void *sendbuf, void *recvbuf, int
    count, MPI_Datatype datatype, MPI_Op
    op, int root, MPI_Comm comm) ;
## MPI Collective Computations

### Collective Computation Operations

<table>
<thead>
<tr>
<th>MPI Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or (xor)</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
</tbody>
</table>

### MPI Collective Computations

#### Collective Computation Operation

<table>
<thead>
<tr>
<th>MPI Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
</tbody>
</table>
**MPI Collective Computations**

**Reduction**

A reduction compares or computes using a set of data stored on all processors and saves the final result on one specified processor.

Global Reduction (sum) of an integer array of size 4 on each processor and accumulate the same on processor P1

**MPI Collective Communication**

**Gather**

Accumulate onto a single processor, the data that resides on all processors

Gather an integer array of size of 4 from each processor
MPI Collective Communication

**Scatter**

Distribute a set of data from one processor to all other processors.

Scatter an integer array of size 16 on 4 processors

![Scatter Diagram]

**Broadcast**

P0 A B C D
P1
P2
P3

P0 A
P1 A
P2 A
P3 A

**Scatter**

P0 A
P1 B
P2 C
P3 D

**Gather**

P0 A
P1 B
P2 C
P3 D

Representation of collective data movement in MPI
MPI Collective Communication

P0
A0

P1
A1
Reduce (A,B,P1,SUM)

P2
A2

P3
A3

P0
A0

P1
A1

P2
A2

P3
A3

Representation of collective data movement in MPI

MPI Collective Communications & Computations

<table>
<thead>
<tr>
<th>Allgather</th>
<th>Allgatherv</th>
<th>Allreduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alltoall</td>
<td>Alltoallv</td>
<td>Bcast</td>
</tr>
<tr>
<td>Gather</td>
<td>Gatherv</td>
<td>Reduce</td>
</tr>
<tr>
<td>Reduce Scatter</td>
<td>Scatterv</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- All versions deliver results to all participating processes
- V-version allows the chunks to have different non-uniform data sizes (Scatterv, Allgatherv, Gatherv)
- All reduce, Reduce, ReduceScatter, and Scan take both built-in and user-defined combination functions
**MPI Collective Communication**

All-to-All

Performs a scatter and gather from all four processors to all other four processors. Every processor accumulates the final values.

All-to-All operation for an integer array of size 8 on 4 processors.
Features of MPI

- **Profiling** - Hooks allow users to intercept MPI calls to install their own tools

- **Environmental**
  - Inquiry
  - Error control

- **Collective**
  - Both built-in and user-defined collective operations
  - Large number of data movements routines
  - Subgroups defined directly or by topology

- **Application-oriented process topologies**
  - Built-in support for grids and graphs (uses groups)

---

Features of MPI

- **General**
  - Communicators combine context and group for message security
  - Thread safety

- **Point-to-Point communication**
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
**Features of MPI**

- **Non-message-passing concepts included:**
  - Active messages
  - Threads

- **Non-message-passing concepts not included:**
  - Process management
  - Remote memory transfers
  - Virtual shared memory

---

**Features of MPI**

- **Positives**
  - MPI is De-facto standard for message-passing in a box
  - Performance was a high-priority in the design
  - Simplified sending message
  - Rich set of collective functions
  - Do not require any daemon to start application
  - No language binding issues
Features of MPI

Pros

- Best scaling seen in practice
- Simple memory model
- Simple to understand conceptually
- Can send messages using any kind of data
- Not limited to "shared -data"

Cons

- Debugging is not easy
- Development of production codes is much difficult and time consuming
- Codes may be indeterministic in nature, using asynchronous communication
- Non-contiguous data handling either use derived data types which are error prone or use lots of messages, which is expensive
Features of MPI-2

MPI-2 Techniques – Positives

- Non-blocking collective communications
- One-sided communication
- “put/get/barrier to reduce synchronization points
- Generalized requests (interrupt receive)
- Dynamic Process spawning/deletion operations
- MPI-I/O
- Thread Safety is ensured
- Additional language bindings for Fortran90 /95 and C++

MPI - Performance

Tuning Performance (General techniques)

- Aggregation
- Decomposition
- Load Balancing
- Changing the Algorithm

Tuning Performance

- Performance Techniques
- MPI-Specific tuning
- Pitfalls
References


3. William Gropp, Rusty Lusk, Tuning MPI Applications for Peak Performance, Pittsburgh (1996)