Introduction to MPI

Victor Getmanskiy

*middle software engineer*

*Singlaris Lab, LLC*
Background on MPI

• MPI - Message Passing Interface
  – Library standard defined by a committee of vendors, implementers, and parallel programmers
  – Used to create parallel programs based on message passing
• 100% portable: one standard, many implementations
• Available on almost all parallel machines in C and Fortran
• Over 100 advanced routines but 6 basic
Key Concepts of MPI

- Used to create parallel programs based on message passing
  - Normally the same program is running on several different processors
  - Processors communicate using message passing
- Typical methodology:

  ```
  start job on n processors
  do i=1 to j
     each processor does some calculation
     pass messages between processor
  end do
  end job
  ```
Messages

• Simplest message: an array of data of one type.

• Predefined types correspond to commonly used types in a given language
  – MPI_FLOAT
  – MPI_DOUBLE
  – MPI_INT

• User can define more complex types and send packages.
Communicators

• Communicator
  – A collection of processors working on some part of a parallel job
  – Used as a parameter for most MPI calls
  – `MPI_COMM_WORLD` includes all of the processors in your job
  – Processors within a communicator are assigned numbers (ranks) 0 to n-1
  – Can create subsets of `MPI_COMM_WORLD`
Include files

- The MPI include file
  
  ```
  #include<mpi.h>
  ```

- Defines many constants used within MPI programs
- In C defines the interfaces for the functions
- Compilers know where to find the include files
Every MPI program needs these...

```c
/* the mpi include file */
#include <mpi.h>

int nPEs,ierr,iam;

/* Initialize MPI */
ierr=MPI_Init(&argc, &argv);

/* How many processors (nPEs) are there?*/
ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPEs);

/* What processor am I (what is my rank)? */
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &iam);
```

...  
ierr=MPI_Finalize();

In C MPI routines are functions and return an error value
Exercise 1 : Hello World

• Write a parallel “hello world” program
  – Initialize MPI
  – Have each processor print out “Hello, World” and its processor number (rank)
  – Quit MPI
Solution 1: Hello World

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```
MPI Basic Send/Receive

- We need to fill in the details in

  Process 0
  \[\text{Send(data)}\]

  Process 1
  \[\text{Receive(data)}\]

- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?
What is message passing?

- Data transfer plus synchronization

- Requires cooperation of sender and receiver
- Cooperation not always apparent in code
Some Basic Concepts

• Processes can be collected into *groups*.
• Each message is sent in a *context*, and must be received in the same context.
• A group and context together form a *communicator*.
• A process is identified by its *rank* in the group associated with a communicator.
• There is a default communicator whose group contains all initial processes, called *MPI_COMM_WORLD*.
Basic Communication

• Data values are transferred from one processor to another
  – One processor sends the data
  – Another receives the data
• Synchronous
  – Call does not return until the message is sent or received
• Asynchronous
  – Call indicates a start of send or receive, and another call is made to determine if finished
Synchronous Send

- MPI_Send(&buffer, count, datatype, destination, tag, communicator);

• Call blocks until message on the way
Synchronous Send

• **Buffer**: The data array to be sent
• **Count**: Length of data array (in elements, 1 for scalars)
• **Datatype**: Type of data, for example: MPI_DOUBLE_PRECISION, MPI_INT, etc
• **Destination**: Destination processor number (within given communicator)
• **Tag**: Message type (arbitrary integer)
• **Communicator**: Your set of processors
• **Ierr**: Error return (Fortran only)
Synchronous Receive

- C
  - MPI_Recv(&buffer, count, datatype, source, tag, communicator, &status);

Call blocks the program until message is in buffer

- Status - contains information about incoming message
  - C
    - MPI_Status status;
Synchronous Receive

• **Buffer**: The data array to be received
• **Count**: Maximum length of data array (in elements, 1 for scalars)
• **Datatype**: Type of data, for example: MPI_DOUBLE_PRECISION, MPI_INT, etc
• **Source**: Source processor number (within given communicator)
• **Tag**: Message type (arbitrary integer)
• **Communicator**: Your set of processors
• **Status**: Information about message
• **Ierr**: Error return (Fortran only)
Exercise 2 : Basic Send and Receive

• Write a parallel program to send & receive data
  – Initialize MPI
  – Have processor 0 send an integer to processor 1
  – Have processor 1 receive an integer from processor 0
  – Both processors print the data
  – Quit MPI
Summary

• MPI is used to create parallel programs based on message passing
• Usually the same program is run on multiple processors
• The 6 basic calls in MPI are:
  – MPI_INIT( ierr )
  – MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
  – MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
  – MPI_Send(buffer, count, MPI_INTEGER, destination, tag, MPI_COMM_WORLD, ierr)
  – MPI_Recv(buffer, count, MPI_INTEGER, source, tag, MPI_COMM_WORLD, status, ierr)
  – call MPI_FINALIZE(ierr)
MPI is Simple

• Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  – MPI_INIT
  – MPI_FINALIZE
  – MPI_COMM_SIZE
  – MPI_COMM_RANK
  – MPI_SEND
  – MPI_RECV

• Point-to-point (send/recv) isn’t the only way...
Introduction to Collective Operations in MPI

• Collective operations are called by all processes in a communicator.

• MPI_BCAST distributes data from one process (the root) to all others in a communicator.

• MPI_REDUCE combines data from all processes in communicator and returns it to one process.

• In many numerical algorithms, SEND/RECEIVE can be replaced by BCAST/REDUCE, improving both simplicity and efficiency.
Collective Operations in MPI syntax

- int MPI_Reduce(
  const void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op op, // op = MPI_MIN, MPI_MAX, MPI_SUM, ...
  int root, // rank to receive reduction in recvbuf
  MPI_Comm comm)

- int MPI_Bcast(
  void *buffer,
  int count,
  MPI_Datatype datatype,
  int root, // rank sending data, others will receive
  MPI_Comm comm)
Example: PI

`mpi.cpp`

`MPI_Bcast(…)`

`MPI_Reduce(…)`
Alternative set of 6 Functions for Simplified MPI

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_BCAST
- MPI_REDUCE

• What else is needed (and why)?
MPI_Scatter

```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```
MPI_Gather

MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    Int root,
    MPI_Comm communicator)
AVG Example

mpiavg.cpp
MPI_Scatter(…)
MPI_Gather(…)
Sources of Deadlocks

• Send a large message from process 0 to process 1
  – If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What happens with

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Send (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Recv (0)</td>
</tr>
</tbody>
</table>

• This is called “unsafe” because it depends on the availability of system buffers
Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
When to use MPI

• Portability and Performance
• Irregular Data Structures
• Building Tools for Others
  – Libraries
• Need to Manage memory on a per processor basis
Summary

• MPI is required to scale parallel app among cluster nodes but also works on local multicore machine.
• The basic paradigm is to send and receive data.
• There are many implementations, on nearly all platforms.
• MPI subsets are easy to learn and use.
• Lots of MPI material is available.
Victor Getmanskiy
middle software engineer
Singularis Lab, LLC
victor.getmanskiy@singularis-lab.com

Dmitry Kryzhanovskiy
CEO at Singularis Lab, LLC
dmitry.kryzhanovskiy@singularis-lab.com