Parallel Programming

Tarek El-Ghazawi

Parallel Programming Models

◆ What is a programming model?
  – A view of data and execution
  – Where architecture and applications meet

◆ Best when a “contract”
  – Everyone knows the rules
  – Performance considerations important

◆ Benefits
  – Application - independence from architecture
  – Architecture - independence from applications
The Data Parallel Model

- Easy to write and comprehend, no synchronization required
- No independent branching
- Example: HPF

The Message Passing Model

- Programmers control data and work distribution
- Explicit communication, two-sided
- Library-based
- Excessive buffering
- Significant communication overhead for small transactions
- Example: MPI
The Shared Memory Model

- Simple statements
  - read remote memory via an expression
  - write remote memory through assignment
- Manipulating shared data may require synchronization
- Does not allow locality exploitation
- Example: OpenMP

The Distributed Shared Memory Model

- Similar to the shared memory paradigm
- Memory $M_i$ has affinity to thread $Th_i$
- Helps exploiting locality of references
- Simple statements
- Examples: UPC, CAF, and Titanium
Hybrid Model(s)

Example: Shared + Message Passing

- Example: OpenMP at the node (SMP), and MPI in between

Performance AND Ease of Use

- Why explicit message passing is often bad
- Contributors to performance under DSM
- Some optimizations that are possible
- Some implementation strategies
What is MPI?

- MPI is not a revolutionary new way of programming parallel computers
- It is an attempt to collect the best features of many message-passing systems that have been developed over the years, improve them and standardize them.

The MPI Forum

- At Supercomping'92, a committee was formed to define a message-passing standard with the following goals
  - Define a portable standard for message passing
  - Operate in a completely open way, allowing anyone to join the discussion
  - Finish it in one year
- The MPI standard was completed on May 1994.
- The forum reconvened during 1995-97 to extend MPI to include remote memory operations, parallel I/O and dynamic process management.
MPI is a library, not a language. It specifies:
- The names, calling sequences and results of subroutines to be called from Fortran programs.
- The functions to be called from C programs
- The classes and methods that make up the MPI C++ library.

The programs that users write are compiled with ordinary compilers and linked with the MPI library.

MPI is a specification, not a particular implementation. A correct MPI program should be able to run on all MPI implementations without change.
MPI Fundamentals

- MPI is a standard specification for a library of message passing functions
- MPI was developed by the “MPI Forum”, a broadly based consortium of parallel computer vendors, library writers, and application specialist
- MPI specifies the library in a language independent form, and provides Fortran and C/C++ bindings.

*MPI Forum: voluntary organization representing industry, government labs, academia

MPI History

- MPI-1 finished in May 1994 (1.0)
- Clarifications (1.1) June 1995
- MPI-1 achieves critical mass in 1996
- MPI-2 started in 1995, finished July 1997
  - MPI-1.2 issued at same time: small corrections/clarifications
  - MPI-2 is additions to MPI-1, not a change
MPI Overview

◆ Goals
  – Portability and Standardization
  – Enable high performance
  – Reflect “common practice”
  – allow efficient implementation

◆ Implementations
  – IBM PCs on Linux and Windows
  – All main Unix workstations and all major parallel computers
  – MPICH is most popular implementation, by ANL and MSU

◆ Features
  – Process model
  – Point-to-point communication (message passing)
  – Miscellaneous utilities, 200 more functions than PVM
Is MPI Large or Small?

◆ MPI is Large (more than 200 functions)
  – Many feature requires extensive API
  – Complexity of use not related to number of functions

◆ MPI is small (6 functions)
  – All that’s needed to get started are only 6 functions

◆ MPI is just right!
  – Flexibility available when required
  – Can start with small subset

Programmer View of MPI

◆ Group of processes that are allowed to communicate with each other, through send and receive calls

◆ A communicator argument, most often MPI_COMM_WORLD, defines your group of processes

◆ A process can use a call to find out its rank (id)

◆ A process can use a call to find out the size of its group

◆ The program (process before execution) is made of a typical C (FORTRAN) program along with these calls
# Anatomy of An MPI Programs

```c
#include<mpi.h>

void 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);
    /* … your code here … */
    printf("Hello from node %d of %d\n", rank, size);
    MPI_Finalize();
}
```

- **Header file**
- **Communicator**
- **Initializing MPI**
- **Rank**
- **Size**
- **Exiting MPI**
- **MPI function format**

### “Hello World” output

- **To Compile:**
  ```
  mpcc hello.c -o hello
  ```
- **To run with 4 processes:**
  ```
  mpirun -np 4 hello
  ```
- **Output:**
  ```
  Hello from node 2 of 4
  Hello from node 1 of 4
  Hello from node 3 of 4
  Hello from node 0 of 4
  ```

* Note - Order of output is not specified by MPI
# Basic MPI functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int MPI_Init(void *args, char **argv)</code></td>
<td>Initialize the MPI environment; must be called before calling any other MPI function</td>
</tr>
<tr>
<td><code>int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)</code></td>
<td>Send contents of buf, containing count instances of datatype to process specified by envelope information</td>
</tr>
<tr>
<td><code>int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)</code></td>
<td>Receive count instances of datatype into buf from process given by envelope, and return information on status</td>
</tr>
<tr>
<td><code>int MPI_Comm_rank(MPI_Comm comm, int *rank)</code></td>
<td>Return in <code>rank</code> the process number of the calling process in group of <code>comm</code></td>
</tr>
<tr>
<td><code>int MPI_Comm_size(MPI_Comm comm, int *size)</code></td>
<td>Return in <code>size</code> the number of processes in group associated with <code>comm</code></td>
</tr>
<tr>
<td><code>int MPI_Finalize(void)</code></td>
<td>Conclude operation and clean up MPI; no MPI functions can be called after.</td>
</tr>
</tbody>
</table>

## Features of MPI

- **MPI features include:**
  - A set of functions to achieve:
    - Point-to-Point Communication (Blocking and Non Blocking)
    - Collective communication (One-to-All, All-to-All, …)
  - Strong Typing
  - Tools for definition of Virtual Topologies
  - A set of tools for performance monitoring
Point-to-Point Communication

Point-to-point communication in MPI

Process 1
Memory
Data
MPI_Send(data, ...)

Process 2
Memory
MPI_Receive(data, ...)
Point-to-Point Communication

- Communication messages involve *Typed data* with an associated *tag*
- Typing of the message is necessary for heterogeneous support
- Tag allows selectivity of messages at the receiving end. Message selectivity on the source of the message is also provided

Example: Process 0 sending a message to process 1; Code executes on both process 0 and process 1; Process 0 sends a character string using MPI_Send.

```c
char msg[20]
int myrank, tag = 99;
MPI_status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find my rank */
if (myrank == 0) {
    strcpy(msg, "Hello there");
    MPI_send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```
**Blocking Send**

\[
\text{MPI\_SEND(buf, count, datatype, dest, tag, comm)}
\]

- **IN** buf address of send buffer
- **IN** count number of entries to send (integer)
- **IN** datatype datatype of each entry (handle)
- **IN** dest rank of destination (integer)
- **IN** tag message tag (integer)
- **IN** comm communicator (handle)

**Blocking Receive**

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

- **OUT** buf Initial address of receive buffer (choice)
- **IN** count max number of entries to receive (integer)
- **IN** datatype datatype of each entry (handle)
- **IN** source rank of source
- **IN** tag message tag (integer)
- **IN** comm communicator (handle)
- **OUT** status return status
Potential Deadlock

- The following example shows a deadlock situation when using blocking send and receive:

  MPI_Comm_rank(comm, &rank)
  if (rank == 0) {
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
  }
  elseif (rank == 1) {
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm)
  }

- The receive operation of the first process must complete before its send, and can complete only if the matching send of the second process is executed.

- The receive operation of the second process must complete before its send and can complete only if the send of the first process is executed. This is a deadlock situation.

Avoiding deadlock

- A modification of the previous program (changing the order of the send & receive operations in the first process) avoids the deadlock:

  MPI_Comm_rank(comm, &rank)
  if (rank == 0) {
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);
  }
  elseif (rank == 1) {
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm)
  }
Non-Blocking Communications: Posting operations

- Calls that post send and receive operations have the same name as the corresponding blocking calls, except for an additional prefix I.

  - MPI_ISEND (buf, count, datatype, dest, tag, comm, request)
  - MPI_Irecv(buf, count, datatype, source, tag, comm, request)

Non-blocking Operations: Completion Operations

- MPI_WAIT and MPI_TEST are used to complete nonblocking send and receive request.

  **MPI_WAIT**(request, status)
  - INOUT request request handle
  - OUT status status object (Status)

  **MPI_TEST**(request, flag, status)
  - INOUT request request handle
  - OUT flag true if operation completed
  - OUT status status object
Collective Communication

Collective communications

- Collective communication addresses data transmission among all processes in a group specified by an intercommunicator object.

- MPI provides the following collective communication functions:
  - Barrier synchronization across all group members.
  - Global communication functions
  - Global reduction operations such as sum, max, min, or user-defined functions.
Global communication functions

- Broadcast from one member to all members of a group
- Gather data from all group members to one member
- Scatter data from one member to all members of the other group
- Scatter/Gather data from all members to all members of a group

Barrier Synchronization

- Group synchronization is achieved through MPI_BARRIER(comm)
- MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.
**Broadcast**

MPI_BCAST(buffer, count, datatype, root, comm)

- INOUT buffer starting address of buffer
- IN count number of entries in buffer
- IN datatype data type of buffer
- IN root rank of broadcast root
- IN comm communicator

- **MPI_BCAST** broadcasts a message from the process with rank root to all processes of the group, itself included. Comm must represent the same intragroup communication domain.

![Broadcast Operation Diagram]

**Gather**

- Each process (root included) sends the contents of its send buffer to the root process.
- The root process receives the messages and stores them in rank order.
- The outcome is as if each of the n processes in the group had executed a call to MPI_Send(), and the root had executed n calls to MPI_Recv() where extent() is the type extent obtained from a call to MPI_Type_get_extent().
Gather

MPI_GATHER(sendbuff, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- IN sendbuff starting address of buffer
- IN sendcount number of elements in send buffer
- IN sendtype data type of send buffer elements
- IN recvbuf address of receive buffer
- IN recvcount number of elements for any single receive
- IN recvtype data type of recv buffer elements
- IN root rank of receiving process
- IN comm communicator

Scatter

- MPI_SCATTER is the inverse operation to MPI_GATHER
- The outcome is as if the root sends a message with MPI_SEND(sendbuf, sendcount,n, sendtype, ...). This message is split into n equal segments, the i\textsuperscript{th} segment is sent to the i\textsuperscript{th} process in the group, and each process receives this message as above.
Scatter

MPI_SCATTER(sendbuff, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- IN sendbuff starting address of buffer
- IN sendcount number of elements in send buffer
- IN sendtype data type of send buffer elements
- IN recvbuf address of receive buffer
- IN recvcount number of elements for any single receive
- IN recvtype data type of recv buffer elements
- IN root rank of receiving process
- IN comm communicator

Gather to all

MPI_ALLGATHER(sendbuff, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
- IN sendbuff starting address of buffer
- IN sendcount number of elements in send buffer
- IN sendtype data type of send buffer elements
- IN recvbuf address of receive buffer
- IN recvcount number of elements for any single receive
- IN recvtype data type of recv buffer elements
- IN comm communicator

- MPI_ALLGATHER can be thought of as MPI_GATHER, except all processes receive the result, instead of just the root. The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

- The outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm), for root = 0, …, n-1.
### All to All Scatter/Gather

**MPI_ALLTOALL**

```c
MPI_ALLTOALL(sendbuff, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
```

- **IN** `sendbuff` starting address of buffer
- **IN** `sendcount` number of elements in send buffer
- **IN** `sendtype` data type of send buffer elements
- **IN** `recvbuf` address of receive buffer
- **IN** `recvcount` number of elements for any single receive
- **IN** `recvtype` data type of recv buffer elements
- **IN** `comm` communicator

- **MPI_ALLTOALL** is extension of **MPI_ALLGATHER** to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

### Global Reduction Op.

- **Reduce**
- **Minloc and Maxloc**
- **All reduce**
- **Reduce-Scatter**
Example using collective operations: Numerical Integration

(computation of $\pi$)

Integrate the function $f$ (which equals $\pi$):

$$
\int_0^1 \frac{1}{1+x^2}
$$

$f(x) = \frac{4}{1+x^2}$

- A perfect parallel program
  - it can be expressed with a minimum of communication
  - Load balancing is automatic
  - We can verify the answer
Implementation in C

```c
#include "mpi.h"
#include <math.h>
int main (int argc, char *argv[]) {
    int n, myid, numprocs; i;
    double PI25DT = 3.141592653589793238462643
    double mypi, pi, h, sum, x;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
        else
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double) i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi-PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```