

Introduction to MPI

9/10/07

MPI

- Stands for Message Passing Interface
- A method to implement *practical* parallel programs that use the message passing model
- Is a library of functions that are :
 - Usable on many clusters
 - Optimized for some specific machines
 - “portable” at least in theory
 - Usable from C or Fortran

Overview

- Each application consists of multiple processes running on multiple processors/machines
- All processes run the same program
- The number of processes may be specified prior to running the parallel program
 - Usually on the command line or in a script

Example

- From the programming assignment, one model could be:
 - One processor is the “master” and distributes work to be processed
 - The remaining processors are “workers” and perform the work needed
- In general branching in a parallel program is used to define roles of processors rather than write independent programs

Details

- Processors are identified with numbers from 0, ..., $p - 1$
- Using MPI functions, each processor can determine its identifier as well as the total number of processes.
- In addition, processors can be organized into higher order structures using other MPI functions

Communication

- There are two basic types of communication:
 - Point to point communication
 - MPI_Send and MPI_Recv
 - Collective communication
 - Broadcast, reduce, all-to-all, etc.

Communicators

- In MPI, all processes are contained in “MPI_COMM_WORLD” which is predefined
- Alternative communicators can be formed using library functions as desired

“Hello world”, MPI style

- Parallel program generates p processes
- Processor 0 waits for a message from the other processors
- All other $p - 1$ processors send a message to processor 0, which prints it out

An example from Pacheco

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

main(int argc, char* argv[]) {

    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for */
    /* receive */
```

The basics

```
/* Start up MPI */
```

```
MPI_Init(&argc, &argv);
```

```
/* Find out process rank*/
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
/* Find out number of processes*/
```

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

The “guts”

```
if (my_rank!= 0) {
    /* Create message */
    sprintf(message, "Greetings from process %d!", my_rank);
    dest= 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
} else { /* my_rank== 0 */

    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
        printf("%s\n", message);
    }
}
```

The end

```
/* Shut down MPI */
```

```
MPI_Finalize();
```

```
} /* main */
```

To review

- This is a minimal MPI program

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

Main (int argc, char* argv[]) {

    /* Start up MPI */
    MPI_Init (&argc, &argv);

    printf ("Hello!!!\n");

    /* Shut down MPI */
    MPI_Finalize ();

} /* main */
```

MPI datatypes

- There are many predefined datatypes such as:
 - MPI_INT
 - MPI_FLOAT
 - MPI_DOUBLE
 - MPI_CHAR
 - etc.

Message passing analogy

- Amy wants to send a message to Bob
- To do so, she composes a letter, puts it in an envelope, addresses and stamps it and puts it in the mailbox
- Bob will get the message when he checks his mail

A summary

- Composing a letter -> filling a message buffer
- Sending a letter -> MPI_send
- Receiving a letter -> MPI_receive
- Sorting letters -> attached tags

MPI_send

```
Int MPI_Send(  
  
    void * message,  
    int count,  
    MPI_Datatype Dt,  
    int dest,  
    Int tag,  
    MPI_Comm comm  
  
);
```

MPI_receive

```
Int MPI_Recv(  
  
    void * message,  
    int count,  
    MPI_Datatype dt,  
    int source,  
    int tag,  
    MPI_Comm comm,  
    MPI_Status* status  
  
)
```

Blocking vs. non-blocking

- Both `MPI_send` and `MPI_receive` are “blocking” operations. In other words, they will not finish until the message is received
- Substantial improvements can be gained from using `MPI_Irecv` instead
- Further, if needed synchronization can be used to reduce overflow errors.

Details for master-worker

- Each worker processor can send a message to processor 0 with its success or failure indicated by the tag field
- How does the master know what to do?
 - Use `MPI_ANY_SOURCE` to receive from any worker
 - Look at the returned status for `MPI_SOURCE`

Compiling on ND system

- /opt/und/mpich/x86/mpich-1.2.7/bin/{mpiCC,mpicc, mpif90, etc.}
- Examples

Cluster status

- <http://iss.cse.nd.edu/ganglia/>

OpenMPI

- <http://www.open-mpi.org/>

Disadvantages of parallel programming

- Almost no good, free parallel debugger
 - Only instance where using printf is a good idea
- Because of issues of running multiple processes, it is important to keep track of the role a given processor is playing