

MPI Programming Quick Start, v. 0.1b

Author: Dimitri Kosturos

dkostur@calstatela.edu

Draft: April 3, 2003

Modified: May 6, 2003

Acknowledgements

First and foremost, thank you to Dr. Raj Pamula for allowing us to continue investigating the Message Passing Interface. I would like to thank my lab partners for their dedication to our task: John SangHi Lee, Weiti Chen, and Tarik Booker. I hope we get the opportunity to work together again.

Introduction

The Message Passing Interface (MPI) was first published in the early 1990's. Since its initial release, it has become the de facto standard in Message Passing parallel programming libraries. MPI allows multiple node systems to communicate via message passing. At CSU, Los Angeles we use the LAM implementation of the MPI library.

Setting Up Your MPI Environment

To enable the MPI environment, you must make sure you have the appropriate **bhost.def** file in your home directory on the head node of the cluster. The bhost.def file should look something like the following:

bhost.def

```
oscar2
oscar2node1
oscar2node2
oscar2node3
oscar2node4
oscar2node5
```

Note that the head node must be listed first. All the other nodes can be listed in any order. The next step is to have **lamboot** initialize the MPI environment for us. Run the following command:

```
[dkostur@oscar2 mpi]$ lamboot bhost.def
```

Be sure to run this command every time you log into the head node to ensure that the lamd (Local Area Multicomputer Daemon) is running. If you are curious about the mapping from nodes to physical machines via **lamboot**, issue the following command:

```
[dkostur@oscar2 mpi]$ lamnodes
```

n0	oscar2
n1	oscar2node1
n2	oscar2node2
n3	oscar2node3
n4	oscar2node4
n5	oscar2node5

Hello World

In the following program, all of the nodes print “Hello from node X” where X is the node rank.

Hello.c

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

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

int size, node;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &node);
MPI_Comm_size(MPI_COMM_WORLD, &size);

printf(“Hello from node %d / %d nodes\n”, node, (size-1));

MPI_Finalize();
return 0;
}
```

In this small and relatively trivial program we have used four of the six fundamental MPI function calls.

MPI_Init: This function initializes the MPI environment.

MPI_Comm_rank: This function is responsible for obtaining the node rank.

MPI_Comm_size: This function obtains the number of nodes in the environment.

MPI_Finalize: Using this function will clean up the MPI environment before exiting.

Compiling an MPI Program

Instead of using the regular gcc or g++ compiler, you will need to use the LAM/MPI provided mpicc for C, mpiCC for C++, and mpif77 for FORTRAN. To compile a C program, for example, issue the following command at the command prompt:

```
[dkostur@oscar2 mpi]$ mpicc source.c -o executable
```

Running an MPI Program

There are a number of ways in which you can run your MPI programs. As of this writing, OSCAR2 supports up to 6 nodes for computation. To start the hello world program above on a specific node, you would instantiate it with the following command:

```
[dkostur@oscar2 mpi]$ mpirun n3 hello
```

The above command will start the program 'hello' on node 3. It is important to note that the head node typically has rank 0. To start an MPI application on all available nodes, use the C option as follows:

```
[dkostur@oscar2 mpi]$ mpirun C hello
```

If there is a particular range of nodes that are needed, you could instantiate your MPI program with the following:

```
[dkostur@oscar2 mpi]$ mpirun n2-5 hello
```

This should effectively run the program on nodes 2 through 5, inclusive. If you need to start multiple instances of the same program on a single machine, use the following:

```
[dkostur@oscar2 mpi]$ mpirun n3,3,3 hello
```

The above command should start three instances of the hello program on node 3.

Using MPI_Bcast

All effective MPI programs Send and Receive messages, however, there are various ways in which these messages can be sent. In this section, we explore the MPI_Bcast call that will broadcast data to all other nodes from a specified node in the group.

Figure 1. Dissection of the MPI_Bcast function call.

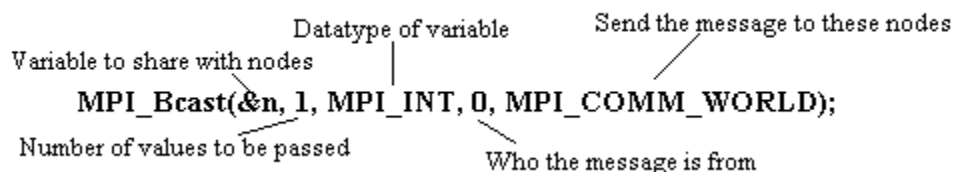


Figure 1 above shows the various components of MPI_Bcast. First specify the name of the variable to be shared. In this case where the C programming language was used the address operator (&) is required. Next specify the number of values to be passed. In this figure only a single variable is being passed. This number must be greater than or equal to 0. After specifying the variable to be shared and the number of values that are being passed, a data type must be assigned. Figure 1 uses the MPI_INT data type. MPI does not know which node should send this data, since all have a copy of the variable as prescribed by the Distributed Memory Architecture. This is why a source must be given. In this case, node 0 is the originator of the broadcast. Finally after knowing all sorts of information about the data to be sent, as well as who is sending it, a destination must be added. MPI_COMM_WORLD is a group communicator for all the nodes specified when starting this program.

The next example is how to use MPI_Bcast to share data among all processes.

bcast.c

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

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

int node, size;
int value;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &node);

if(node == 0) {
    printf("Enter a value: ");
    scanf("%d", &value);
}

MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("Node(%d) has variable: value set to: %d\n", node, value);

MPI_Finalize();
return 0;
}
```

Notice that the MPI_Bcast call does not need to be placed inside of the if statement block. This is because a source for the variable has been specified as the process with rank 0.

Using MPI_Send and MPI_Receive

Sending and receiving messages are the basis of any message passing system. The next example shows how to use MPI_Send and MPI_Recv to send messages to the next node until all nodes hold the same value.

Send and Receive

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

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

int size, node;
int value;

MPI_Status status; MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &node);

if (node == 0) {
    printf("Value: ");
    scanf("%d", &value);
    MPI_Send(&value, 1, MPI_INT, node+1, 0, MPI_COMM_WORLD);
} else {
    MPI_Recv(&value, 1, MPI_INT, node - 1, 0, MPI_COMM_WORLD, &status);
    if(node < size - 1)
        MPI_Send(&value, 1, MPI_INT, node+1, 0, MPI_COMM_WORLD);
}
printf("Node %d has %d in value.\n", node, value);

MPI_Finalize();
return 0;
}
```

Output from the above program under various runs:

```
[dkostur@oscar2 dkostur]$ mpirun C ring
Value: 23
Node 0 has 23 in value.
Node 1 has 23 in value.
Node 5 has 23 in value.
Node 2 has 23 in value.
Node 3 has 23 in value.
```

```
Node 4 has 23 in value.
[dkostur@oscar2 dkostur]$ mpirun C ring
Value: 12
Node 0 has 12 in value.
Node 1 has 12 in value.
Node 2 has 12 in value.
Node 5 has 12 in value.
Node 3 has 12 in value.
Node 4 has 12 in value.
[dkostur@oscar2 dkostur]$
```

The output above indicates that all of the nodes received the value, however, node ranks are not necessarily assigned in the order we might initially think. For example, logical node 2 could possibly be physical node 4. Consider the following output using the above program:

```
[dkostur@oscar2 dkostur]$ mpirun n0,2,2,4 ring
Value: 23
Node 0 has 23 in value.
Node 1 has 23 in value.
Node 2 has 23 in value.
Node 3 has 23 in value.
```

In this example, we have specified which nodes to use; however the node rank is independent of the node number specified on the command line, with the exception of node 0.

MPI_Send parameters are as follows:

MPI_Send(variable, number of variables to send, MPI Data type, who will receive this message, an MPI tag, and the group communicator)

Example Usage:

```
MPI_Send(&value, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
```

MPI_Recv parameters are as follows:

MPI_Recv(variable, number of variables to send, MPI Data type, node sending message, message tag, group communicator, and an MPI status indicator)

Example Usage:

```
MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

Collecting Data from Processes (MPI Reduce)

MPI_Reduce allows programmers to collect data at one node and reduce it by means of a reduction operation. The valid reduction operations are:

Operation	What it does
MPI_SUM	Sums all data received
MPI_PROD	Multiplies all data received
MPI_BAND	Logical and
MPI_BAND	Bit-wise and
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR
MPI_MIN	Minimum
MPI_MAX	Maximum

Programmers can create custom operations via MPI_OP_CREATE, more on this later. When nodes finish processing, a call to MPI_Reduce can bring all the data together. For example, if we calculate the value of π on several nodes we must bring the results together and sum them. An example MPI_Reduce call might look like the following:

```
MPI_Reduce(&nodePi, &pi, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

In this function call we specify the variable to send (nodePi), the variable that receives the data (pi), the number of values to receive (1), an MPI Datatype (MPI_FLOAT), the predefined reduce operation to be used (MPI_SUM), the node which will be receiving the data (0), and which communicator to use (MPI_COMM_WORLD).

Obtaining the physical machine name in an MPI Program

Sometimes it may be helpful to obtain the physical machine name that is associated with a particular node. This may aid in performance tuning and administration. The function to use is called **MPI_Get_processor_name**.

Simple Usage of MPI_Get_processor_name.

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

int main(int argc, char** argv) {
    char name[80];
    int strLen;
    int node, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
MPI_Get_processor_name(name, &strLen);
printf("node(%d) is physically named: %s\n", node, name);
MPI_Finalize();
return 0;
}
```

Barriers in MPI

Barriers are quite useful in distributed and parallel computing. With this in mind the MPI Forum included an `MPI_Barrier` entry in the specification. Using the `MPI_Barrier` function is quite easy:

```
MPI_Barrier(MPI_COMM comm);
```

This function will block processes until all processes in **comm** have reached the barrier.

Building Simple Data Types

Building up simple data types allows MPI code to be easier to read as well as easier to write and maintain. Suppose we have the following structure:

```
struct {  
int x;  
char y;  
} myStruct;
```

The goal is to make it possible to send **myStruct** objects as a single data type instead of sending each component separately. There are a few different ways to accomplish this in MPI. The technique shown in this section is the most general.

Structure Broadcast

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

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

    struct {
        int x;
        char y;
    } myStruct;

    int rank, size, i;

    /* Set up the 2 blocks */
    int blockCounts[2] = {1,1};
    MPI_Datatype types[2];
    MPI_Aint displacements[2];
```

```

MPI_Datatype myStructType;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

/* init types and displace with addresses of items. */
MPI_Address(&myStruct.x, &displace[0]);
MPI_Address(&myStruct.y, &displace[1]);

types[0] = MPI_INT;
types[1] = MPI_CHAR;

for(i = 1; i >= 0; i--) {
    displace[i] -= displace[0];
}

MPI_Type_struct(2, blockCounts, displace, types, &myStructType);
MPI_Type_commit(&myStructType);

if(rank == 0) {
    myStruct.x = 18;
    myStruct.y = 'D';
}

MPI_Bcast(&myStruct, 1, myStructType, 0, MPI_COMM_WORLD);
printf("Rank %d has: %d, %c\n", rank, myStruct.x, myStruct.y);

MPI_Finalize();
return 0;
}

```

The important MPI functions introduced here are **MPI_Address**, **MPI_Type_struct**, and **MPI_Type_commit**.

int MPI_Address(void* location, MPI_Aint *address);

The first parameter is an item from the structure; the second parameter is the returned address of that structural component.

int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);

The first parameter is the number of blocks in the structure. The second parameter is the array holding the length of the blocks. The third parameter is the array of displacements (the adjusted results from MPI_Address). The fourth parameter is

the array of types, and finally the fifth parameter is the address of the structure type.

MPI_Type_commit(MPI_Datatype *datatype);

The only parameter passed to this function is the address of the structure type. After this line of code has been executed, all of the communications functions can utilize this derived type.

Conclusion

This document is in no way complete as indicated by the version number above. It is a work in progress. However, this manual contains enough information to write many useful Message Passing programs. Setting up the programming environment, sending and receiving, broadcasting and reducing, barriers, and using derived data types have been covered.