Simple Parallel Data Structures-1
by
William Gropp and Ewing Lusk

1. Getting Started with “Hello World” – A Worked Example

Write a program that uses MPI and has each MPI process print

Hello world from process i of n

using the rank in MPI_COMM_WORLD for i and the size of MPI_COMM_WORLD for n. You can assume that all processes support output for this example.

Note the order that the output appears in. Depending on your MPI implementation, characters from different lines may be intermixed. A subsequent exercise (I/O master/slaves) will show how to order the output.

You may want to use these MPI routines in your solution:
MPI_Init MPI_Comm_size MPI_Comm_rank MPI_Finalize

A Solution

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

int main( argc, argv )
int  argc;
char **argv;
{
   int rank, size;
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   printf( "Hello world from process %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

Makefile

```
# Generated automatically from Makefile.in by configure.
ALL: helloworld

helloworld: helloworld.c
          mpicc -o helloworld helloworld.c

clean:
          /bin/rm -f helloworld *.o
```

Output

```
% mpicc -o helloworld helloworld.c
% mpirun -np 4 helloworld
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
%
```

2. Exercise: Shared Data

A common need is for one process to get data from the user, either by reading from the terminal or command line arguments, and then to distribute this information to all other processors.

Write a program that reads an integer value from the terminal and distributes the value to all of the MPI processes. Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.

You may find it helpful to include a fflush(stdout); after the printf calls in your program. Without this, output may not appear when you expect it.

You may want to use these MPI routines in your solution:
MPI_Init MPI_Comm_rank MPI_Bcast MPI_Finalize

3. Exercise: Using MPI datatypes to share data

In this assignment, you will modify your argument broadcast routine to communicate different datatypes with a single MPI broadcast (MPI_Bcast) call. Have your program read an integer and a double-precision value from standard input (from process 0, as before), and communicate this to all of the other processes with an MPI_Bcast call. Use MPI datatypes.

Have all processes exit when a negative integer is read.

You may want to use these MPI routines in your solution:
MPI_Address MPI_Type_struct MPI_Type_commit MPI_Type_free MPI_Bcast

4. Exercise: Using MPI_Pack to share data

In this assignment, you will modify your argument broadcast routine to communicate different datatypes by using MPI_Pack and MPI_Unpack.

Have your program read an integer and a double-precision value from standard input (from process 0, as before), and communicate this to all of the other processes with an MPI_Bcast call. Use MPI_Pack to pack the data into a buffer (for simplicity, you can use char packbuf[100]; but consider how to use MPI_Pack_size instead).

Note that MPI_Bcast, unlike MPI_Send/MPI_Recv operations, requires that exactly the same amount of data be sent and received. Thus, you will need to make sure that all processes have the same value for the count argument to MPI_Bcast.

Have all processes exit when a negative integer is read.

You may want to use these MPI routines in your solution:
MPI_Pack MPI_Unpack MPI_Bcast