A SIMPLIFIED MESSAGE-PASSING LIBRARY
Timothy J. McGuire
Sam Houston State University
Department of Computer Science
Huntsville, TX  77341-2090
(936) 294-1571
mcguire@shsu.edu

ABSTRACT

MPI, the standard library for message-passing in distributed computing, is a complex package containing over 200 library routines. Experience with teaching distributed processing to 1st year students has shown that MPI is too complex for even well-motivated students. Others have observed that only six MPI routines are necessary to solve any message-passing problem. A simplified API is necessary to make message-passing more accessible to the average programmer. Another benefit of a simplified API is the ease of porting it to new platforms and computing environments.

This simplified library is based on a minimalist subset of MPI. The goal is to allow people to run simple message-passing programs on a network of workstations using a small library. Analogues for the six standard point-to-point and environmental functions are included. Barrier, Bcast, Gather, Scatter, and Reduce are also implemented.

The current implementation simply maps the simplified functions onto actual MPI functions. A socket-based implementation is possible and is under current development.

INTRODUCTION

This research grew out of the author’s experience with teaching distributed processing to 1st year students. This first exposure was greeted with enthusiasm; however, the attrition rate was high. The high attrition rate was probably due to the complexity of the API; that is:

- There are more MPI functions than any one person will likely learn and use
- The parameter lists for the functions are typically long and relatively complicated

Thus, there is a substantial learning curve to using MPI, and most users (beginners or not) just want to solve problems. Beginning students must have a clear understanding of the use of arrays, pointers, string construction, buffers, and other advanced concepts before they can begin to use message-passing. We concluded therefore, that a simplified API was necessary to make message-passing more accessible the undergraduate student, and to hide unnecessary details from the user.

The target machine for this research was Beowulf cluster; however, all the techniques we discuss can also be extended to a symmetric multiprocessor or a network of workstations with no
modification whatsoever. The environment has been tested on a variety of Linux-based machines, using both LAM and MPICH versions of MPI.

THE RESEARCH PLAN

The chosen research plan was relatively straightforward:

• Design a simple API to perform message passing using the six essential message-passing functions
• Examine which of the remaining operations would be the most useful, and add those to the API
• Implement the API using MPI routines
• Also implement (for efficiency) the API using lower level routines
• Compare the simplified API with MPI in terms of ease of use and efficiency

THE IMPLEMENTATION

Since our goal was to make the API as simple as possible, it was decided that we would assume the SPMD (Single-Program, Multiple-Data) model for distributed processes [5]. In this model, different processes are merged into one program, and within that program, control statements select different parts for each process to execute. All the executables may be started together at the beginning, saving the complexity of implementing dynamic process creation. In addition, using this (relatively) simple programming model enables the programmer to write virtually any message-passing program using only six essential functions [1]. The MPI versions of these are: MPI_Init(), MPI_Comm_size(), MPI_Comm_rank(), MPI_Send(), MPI_Recv(), and MPI_Finalize(). Previous experience with first-year computer science students shows that if they are sufficiently motivated and mentored, they can readily grasp what these functions do, and solve a wide variety of problems. Although the problems described in this paper may be solved using only these six functions, some solutions can be made more efficient by using other functions such as MPI_Bcast() or MPI_Reduce() [2].

One of the difficulties experienced by the first-year students was the relatively long list of formal parameters required by the MPI functions. Since the goal was simplification, and the use of the SPMD programming model was assumed, a major goal was the simplification of this parameter list. For example, MPI_Send() requires six actual arguments, and MPI_Recv() requires seven. Since we chose not to implement any communicator other than the default (MPI_COMM_WORLD,) nor message tags or status objects, this allowed us to limit the parameter list for these functions to a more acceptable size of four.

We chose a C-based, rather than a C++-based interface to avoid problems when attempting to pass objects in messages. It is our opinion that it would be better to implement class member functions using the application classes and call the MPI C functions inside the class member functions when message-passing is required.
DATA TYPES

The data types chosen were selected from those analogous to the primitive data types of C and C++, as indicated in the table below. The BYTE type (analogous to MPI_BYTE) was also included to enable the passing of non-primitive types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Analogous C data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>None</td>
</tr>
<tr>
<td>CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>INT</td>
<td>signed int</td>
</tr>
<tr>
<td>LONG</td>
<td>signed long</td>
</tr>
<tr>
<td>LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>UNSIGNED_LONG</td>
<td>unsigned long</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
</tbody>
</table>

THE MESSAGE-PASSING FUNCTIONS

In all, 11 functions were eventually settled upon: Init(), Finalize(), GetRank(), GetSize(), Send(), Recv(), Bcast(), Reduce(), Scatter(), Gather(), and Barrier().

```c
void Init(int argc, char ***argv);
• Initializes the simplified message-passing environment. This function must be called by each process before any message-passing functions are called.

void Finalize(void);
• Each process must call Finalize() before it exits.

int GetRank(void);
• Returns the process rank of the calling process.

int GetSize(void);
• Returns the number of processes in the process group.

void Send(Datatype type, void *buffer, int count, int dest);
• Sends a message to another process.

void Recv(Datatype type, void *buffer, int count, int source);
• Receives a message from another process.

void Bcast(Datatype type, void *buffer, int count, int root);
• The contents of the send buffer are copied to all other processes

void Reduce(Datatype type, void *sourcebuffer, void *recvbuffer, int count, Operation op, int root);
• Combines the elements stored in the source buffer of each process in the process group using the operation specified in op, and returns the combined values in the receive buffer of the process with rank root.
```
void Gather(Datatype type, void *sendbuffer, void *recvbuffer, int count, int root);

- Each process, including the root process, sends the data in sendbuffer to the recvbuffer of the process with rank root.

void Scatter(Datatype type, void *sendbuffer, void *recvbuffer, int count, int root);

- The process with rank root sends a different part of sendbuffer to each process (including itself). The received data is stored in the recvbuffer. Process i receives count contiguous elements of data type type starting from the i*count position of root's sendbuffer to the recvbuffer of the process with rank root.

void Barrier(void);

- This function ensures synchronization of all the processes in the process group.

REDUCTION OPERATORS

Reduce() requires that reduction operators be defined. They are:

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXIMUM</td>
<td>Maximum</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>Minimum</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>Product</td>
</tr>
<tr>
<td>SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>LOGICAL_AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>LOGICAL_OR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>LOGICAL_XOR</td>
<td>Logical Exclusive OR</td>
</tr>
<tr>
<td>BITWISE_AND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>BITWISE_OR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>BITWISE_XOR</td>
<td>Bitwise Exclusive OR</td>
</tr>
</tbody>
</table>

A SIMPLE PROGRAM

```c
#include <stdio.h>
#include "simple.h"
int 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 */

    /* Start up SIMPL */
    Init(&argc, &argv);

    /* Find out process rank */
    my_rank = GetRank();

    /* Find out number of processes */
```
p = GetSize();

if (my_rank != 0)
{
    /* Create message */
    sprintf(message, "Greetings from process%d!", my_rank);
    dest = 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    Send(CHAR, message, strlen(message)+1, dest);
}
else /* my_rank == 0 */
    for (source = 1; source < p; source++)
    {
        Recv(CHAR, message, 100, source);
        printf("%s\n", message);
    } /* end for */
/* Shut down SIMPL */
Finalize();
} /* main */

FUTURE WORK

Work has begun on a socket based implementation. The first version does not use secure sockets; thus this software would be potentially subject to malicious attack or perhaps random events connecting to the insecure ports. Attention should be paid to portability across a variety of Unix and Linux dialects, since there are slight variation in their use of sockets over TCP/IP. The benefits of this library should include easy customization, and instructional use for socket programming. This library is currently being used in the author’s operating systems course, and the results will be reported in a future paper.

REFERENCES


