MPI

Overview

MPI is an acronym for Message Passing Interface, which is a standard Application Programming Interface (API) for parallel programming in C, C++ and Fortran on distributed memory machines.

MPI is implemented as a library of functions, a C, C++ or Fortran programmer doesn't have to learn a new language to use MPI; if a sequential program already exists, much of that program can be used for an MPI version. An MPI program complies just like a regular program. At link time, the MPI library must be accessed.

On clusters, each node has a separate memory space, i.e. distributed memory, as a result, there is no way to get at another's except via message passing over the network. Messages may be used for:

- Sending data
- Performing operations on data
- Synchronization between tasks

For efficiency, MPI is usually run on clusters of many computers of the same type, with high-speed connection. MPI could also be run (with less reliable speedup) on collections of several types of computers with relative slow connections. Due to distributed memory between processors, the data for the problem has been divided up among these processors. This means that, from time to time, one processor may need to "ask" another processor for a copy of certain data or result.

Another attractive approach to parallel programming is called OpenMP. This system is used when a shared memorymachine is available, this is when several processors share a single physical memory, or a number of processors can address the same "logical" memory. An existing sequential program can be turned into an OpenMP program simply by inserting special comment statements.

MPI Code Structure

MPI has a wide range of capabilities. There are three subsets of functionality:

- Basic (about 6 functions)
- Intermediate
- Advanced (up to 125 functions)

All processes must initialize and finalize MPI

- `MPI_Init()`: Starts up the MPI runtime environment
- `MPI_Finalize()`: Shuts down the MPI runtime environment

MPI uses `MPI_Comm` objects to define subsets of processors (i.e. communicators) that may communicate with one another. By default `MPI_COMM_WORLD` includes all your processors.

Some examples of simple MPI programs are provided in the Examples section.

MPI Execution

Single Program, Multiple Data (SPMD) means every process gets a copy of the executable. Each process looks at its own rank (unique ID within a communicator) to determine which part of the problem to work on. Except when communicating, each process works completely independently.

Point-to-Point (P2P) Communication

Point-to-Point (P2P) Communication sends data from one point to another, one task sends while another receives.

- **Basic P2P** involves two functions, `MPI_Send()` and `MPI_MPI_Recv()`. An alternative is `MPI_SendRecv` with different tags to indicate send and receive stages. It's especially useful when each node both sends and receives messages (two-way communication).
- **Synchronous Communication**: Messages are sent and received via `MPI_Ssend()` and `MPI_Srecv()`. Handshaking occurs between send and receive tasks to confirm a safe send. As a result, in some case, it blocks send/receive.
- **Buffered Communication**: Messages are sent and received via `MPI_Bsend()` and `MPI_Brecv()`. The content of the message is copied into a system-controlled block of memory (system buffer). The sender continues executing other tasks; when the receiver is ready to receive, the system simply copies the buffered message into the appropriate memory location.

Blocking vs. Non-blocking

MPI communications can be blocking or non-blocking:

- A **Blocking** (`MPI_Send()` and `MPI_MPI_Recv()`) send routing will only return after it is safe to modify the buffer. "Safe" in this case means that modification will not affect the data to be sent. Safe does not imply that the data was actually received.
- **Non-blocking** (`MPI_Isend()` and `MPI_Irecv()`) send/receive routines return immediately. Non-blocking operations request that the MPI library perform the operation "when possible". It is unsafe to modify the buffer until the requested operation has been performed. Non-blocking communications are primarily used to overlap computation with communication, optimizing performance.

Deadlock

Improper use of blocking receive/send will result in deadlock, where two processors cannot progress because each of them is waiting on the other. (This can also happen for many processors.) For example, the following code contains a deadlock:

```
      IF (rank==0) THEN
             CALL
MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)

             CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)

      ELSEIF (rank==1) THEN
             CALL
MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)

             CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)

      ENDIF
```

This can be solved with the following minor modification:

```
      IF (rank==0) THEN

            CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)

            CALL
MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)

      ELSEIF (rank==1) THEN

            CALL
MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)

            CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)

      ENDIF
```

Collective Communication

As the name suggests, collective communication is defined as communication between more than two (usually many) processors. They come in a few forms:

- One-to-many
- Many-to-one
- Many-to-many

A collective operation requires that all processes within the communicator group call the same collective communication function with matching arguments. A few points to note:

- Collective communicates are blocking operations.
- Collective operations on subsets of processes require separate grouping/new communicators.

- The size of data sent must exactly match the size of the data received.

Some of the most important functions in collective communication are listed below. The actions of these functions are summarized in Figure 1.

- `MPI_BCAST()`: Broadcast a message from the root process
- `MPI_SCATTER()`: Each process receive a segment from the root
- `MPI_GATHER()`: Each process sends contents to the root (opposite of `MPI_SCATTER()`)
- `MPI_ALLGATHER()`: An `MPI_GATHER()` whose result ends up on all processors
- `MPI_REDUCE()`: Applies a reduction operation on all tasks and places the result in the receive buffer on the root process
- `MPI_ALLREDUCE()`: Applies a reduction and place the result in all tasks in the group (`MPI_REDUCE()`+`MPI_BCAST()`)

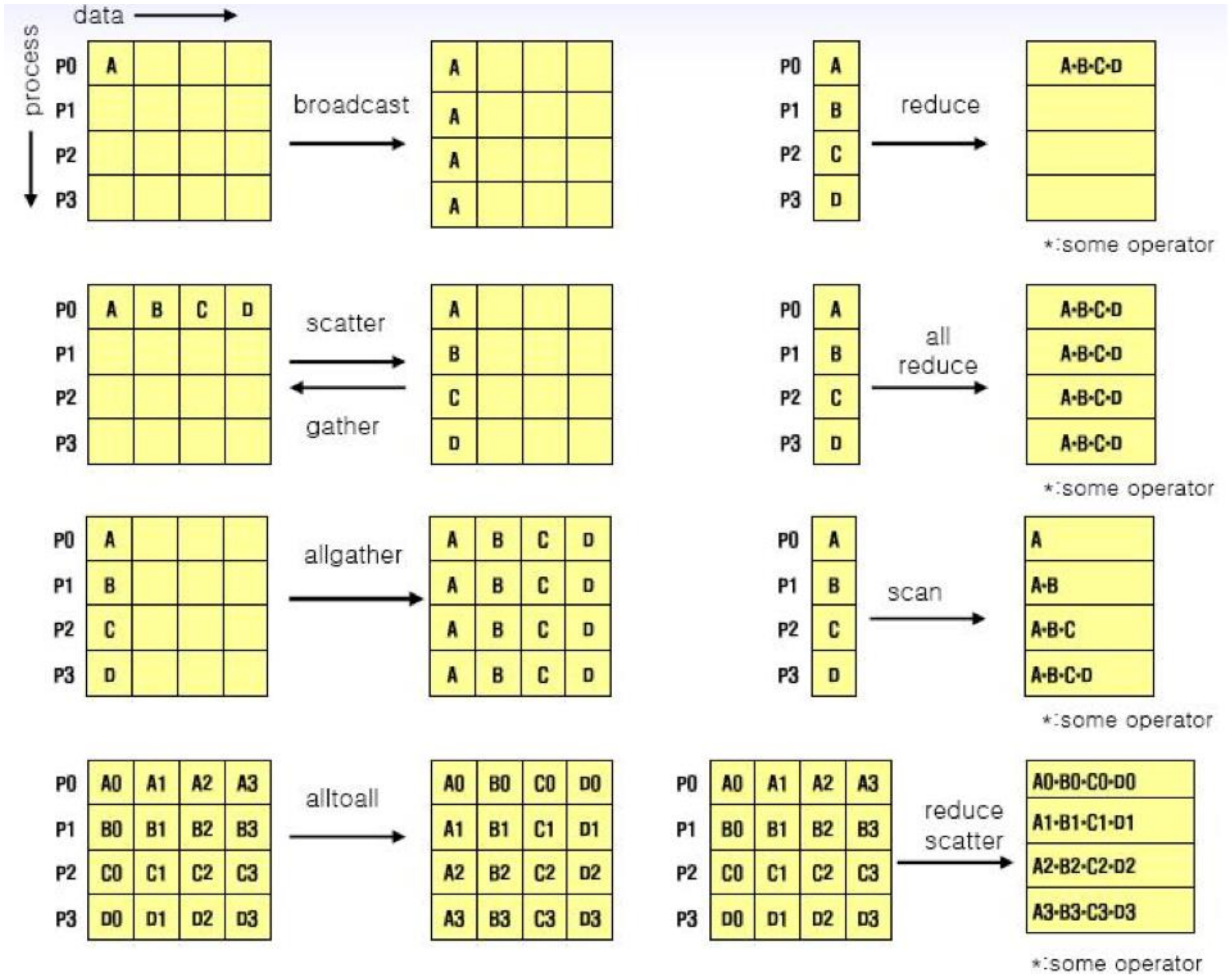


Figure 1: A summary of some common collective communications

Compiling

The following section gives an example of compiling MPI programs using MPICH:

- C: `mpicc -o foo foo.c`
- Fortran: `mpif90 -o foo foo.f`

And the command for running MPI programs using MPICH is: `mpirun -np 2 <progname>`

OpenMPI is an alternative implementation of open source MPI-2 on ARC machines.

//

MPI Parallelizes Work among Multiple Processors or Hosts

Introduction

Message passing is a common technique for performing parallel processing spread among multiple processors. Processes execute tasks on individual processors and communicate with each other by sending messages. In this way, processes can operate in a semi-autonomous manner, performing distinct computations that form part of a larger job, sharing data and synchronizing with each other when required.

Message-passing systems assume a distributed memory model in which each process executes in a different memory space from the other processes. This scheme works equally well when the processors are part of the same computer or spread across a range of heterogeneous machines spanning a network. Message-passing systems can be highly scalable, but to make the best use of message passing, applications must be designed to exploit the parallelism available.

Many academic institutions and commercial organizations have developed their own proprietary models for message passing. In the 1990s, the Message Passing Interface Forum (MPIF) defined a standard application programming interface (API) for message passing, known as the Message Passing Interface (MPI). The work of the MPIF was strongly influenced by work performed by IBM research engineers and by the Intel® NX/2 operating system, among others.

MPI is a specification. MPI libraries are available from a number of sources that implement MPI on particular hardware or that map through to vendor-specific message-passing implementations. Applications that use MPI should be portable across all platforms that have an MPI library available.

This paper provides an introduction to the concepts that underpin MPI and highlights some examples of MPI using MPICH. MPICH is a freely downloadable implementation of MPI that is available for Windows* and Linux*.

Overview of the Message Passing Interface

Message passing provides a means for complex tasks and computations to be broken down into discrete pieces, each of which can be performed in parallel on different processors. Messages containing processing requests and data can be passed from one processor to another in a controlled, synchronized manner.

Message passing is not a new concept, and many organizations have used this approach to make use of spare capacity in underused machines, or to take advantage of specialized hardware (high-precision numeric processors, for example) at the appropriate points in applications. The computers involved may be local to each other, or they may be spread over wide distances.

In the past, organizations that used message passing have often designed and implemented their own schemes. While this is an interesting exercise, the results tended to be highly specific to a particular set of computers, topology, operating system, and even the application being built. MPI was designed as a portable specification to support message passing across a range of environments. MPI supports many different platforms, including highly parallel multi-processor computers, tightly connected clusters, and loosely connected heterogeneous networks.

It is important to realize that all MPI defines is a model for passing me ssages and the definitions of a number of methods and data types that support that model. MPI is language-independent, but it defines language binding for common programming languages such as C, C++, and Fortran. MPI is not a product; different implementations of MPI from different vendors should interoperate as long as they adhere closely to the specifications.

The original MPI specification (MPI-1) provided co-operative, point-to-point send and receive operations; one process would send data while the other would explicitly receive it. This mechanism is synchronous; a sender will block if the receiving process is not listening, and a receive process will block if the sending process has not yet sent anything. This model helps to prevent inadvertent loss of messages.

MPI also provides non-blocking send and receive operations, which can increase the degree of parallelism at the cost of the additional complexity required to perform synchronization when required. One point to bear in mind is that MPI does not guarantee the sequence in which messages will be received. For example, if process A and process B send a message to process C, process C can receive the messages from processes A and B in any order.

The MPI-2 specification extended the model to permit one-sided communications. In this scheme, a process can simply transfer information from one address space to another. If the transfer is made from the process's own address space to the memory of another process, the operation is referred to as a *put*. If the transfer is made from another process's address space to that of the current process, the operation is called a *get*.

Either way, only one process actively participates in the operation, and blocking will not occur. Synchronization, however, is an important issue. MPI-1 was restricted to using a fixed set of processes, but MPI-2 supports the creation of dynamic processes.

MPI also provides collective communications. One process can broadcast the same message to many others for processing. Data can also be received from multiple source processes and combined together (this is called reduction). Note that, strictly speaking, send and receive operations are simply special cases of broadcast and reduce.

Processes communicating using MPI are collected into groups, often to allow the subdivision of work into independent chunks. Processes can only communicate with other processes that are in the same group, although processes can be members of multiple groups. Processes are identified by their positions (*rank*) within the groups that they belong to.

When a process sends a message, it adds a user-defined tag that helps the receiving process to identify the message. The receiving process can elect to screen messages based on this tag information; for example, messages with a particular tag value can be ignored. Messages also have a context, which can be used by the MPI implementation to help identify the set of messages passing between processes.

The process group and the context together form the *communicator*. MPI applications can create their own communicators or use the default communicator that spans all the initial processes, called *MPI_COMM_WORLD*. When a process sends or receives a message, it must specify which communicator to use.

MPI Provides Architecture-Neutral Datatypes and Extensive Functions

A key feature of MPI is its portability. A process executing on one computer can send a message to another process running on a different processor architecture. The data transmitted must be received in an unambiguous manner. Any differences in data formats, such as big-endian versus little-endian issues or data lengths, must be transparent.

MPI provides a number of basic, architecture-neutral datatypes for supported language binding (such as *MPI_INT*, *MPI_FLOAT*, and *MPI_DOUBLE* for C programs). MPI also contains functions that allow the developer to define custom datatypes such as arrays, structures, and vectors.

Although MPI includes in excess of one hundred functions, it is possible to write functional MPI programs using a core set of six functions:

- **MPI_INIT**. This function is used to initialize the MPI environment and must be called before performing any other MPI functions.
- **MPI_FINALIZE**. This function terminates an MPI computation and should be called as part of the general cleaning up before an application finishes.
- **MPI_COMM_SIZE**. This function ascertains the number of processes being used.
- **MPI_COMM_RANK**. This function can be used to determine the identity of the current process.

- **MPI_SEND**. This function is used to send a message to a process using the basic point-to-point mechanism.
- **MPI_RECEIVE**. This function receives a message using the basic point-to-point mechanism.

Two other functions are useful when using collective communications:

- **MPI_BCAST**. This function sends a message from one process to all others in the same communicator.
- **MPI_REDUCE**. This function combines data from all processes in the same communicator and returns it to one process.

The MPMD Model: Different Tasks over Different Data

MPI is geared toward systems performing computations using a number of cooperating processes. MPI is suitable for two particular programming models:

- Multiple Program Multiple Data (MPMD), in which processes perform different tasks over different data.
- Single Program Multiple Data (SPMD), in which processes perform the same task over different data.

The MPMD model is sometimes referred to as *Master and Slave*. A master process is responsible for subdividing the work to be performed into discrete tasks and then allocating each task to a slave process to perform. The slave processes can operate autonomously and can communicate with the master process by sending and receiving messages. Slave processes can also communicate with other slave processes if necessary.

The program below shows an example written in C implementing the MPMD model using MPI:

```
#include "mpi.h"

#include "stdio.h"

#define MULT_MSG 1
```

```
#define ADD_MSG 2

int myid, namelen;
char processorname[MPI_MAX_PROCESSOR_NAME];

void adder()
{
int data[4] = {0, 0, 0, 0}, sum = 0, i;
MPI_Status status;
MPI_Recv(data, 4, MPI_INT, 0, ADD_MSG, MPI_COMM_WORLD, &status);
for (i = 0; i < 4; i++)
{
sum += data[i];
}
printf("Result generated by process %d on %s is %d",
myid, processorname, sum);
}

void multiplier()
{
int data[4] = {0, 0, 0, 0}, product = 1, i;
MPI_Status status;

MPI_Recv(data, 4, MPI_INT, 0, MULT_MSG,
MPI_COMM_WORLD, &status);
for (i = 0; i < 4; i++)
{
product *= data[i];
}
printf("Result generated by process %d on %s is %d",
myid, processorname, product);
}

void master()
{
int data[4] = {1, 3, 5, 7};

MPI_Send(data, 4, MPI_INT, 1, ADD_MSG, MPI_COMM_WORLD);
MPI_Send(data, 4, MPI_INT, 2, MULT_MSG, MPI_COMM_WORLD);
printf("Data sent from process %d on %s",
myid, processorname);
}
```

```


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

{

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myid);

MPI_Get_processor_name(processorname, &namelen);


switch (myid)

{

case 0: master();

break;

case 1: adder();

break;

case 2: multiplier();

break;

}

MPI_Finalize();

}
```

The program is intended to span three processes:

- A process that runs the **adder** function that computes the sum of a set of numbers.

- A process that runs the **multiplier** function that calculates the product of a set of numbers.
- A process that runs the **master** function that supplies the data to be summed and multiplied.

Although MPI specifies the functions and data structures required to support message passing, it does not indicate how they should be implemented nor how a compiler should handle them. The code examples in this paper have been developed using the MPICH library* (a portable MPI implementation) with the Microsoft Visual C++* Compiler.

The application must be executed using a runtime environment that supports MPI and will create the required number of processes. Most MPI implementations provide such a tool, and MPICH supplies the MPIRun utility for this purpose. This utility is available in two versions; one for use from the command line and another that provides a GUI interface (this paper uses the GUI version of the tool).

When the application starts, the **MPI_Init** function is called to initialize the MPI environment (line 49). The function s **MPI_Comm_rank** and **MPI_Get_processor_name** (lines 50 – 51) retrieve the ID of the current process, and the name of the host computer running the process. (MPICH can be configured to launch processes on different networked hosts, as long as each computer has a copy of the MPICH runtime installed.) The process ID is used to determine which function to call (**master**, **adder**, or **multiplier**). The program finishes by calling **MPI_Finalize** (line 62) to tidy up the environment.

The master function (lines 37-45) creates and populates an array of data to be sent to the other two processes. The first **MPI_Send** function call (line 41) transmits this data to the process running the **adder** function. The parameters are:

- The address of the data being sent.
- The number of elements (4).
- The data type of the elements (**MPI_INT**).
- The ID of the process the message should be sent to (1).
- A user-defined tag identifying the message to the receiver (**ADD_MSG**).
- The communicator to use (this example uses the global communicator **MPI_COMM_WORLD**).

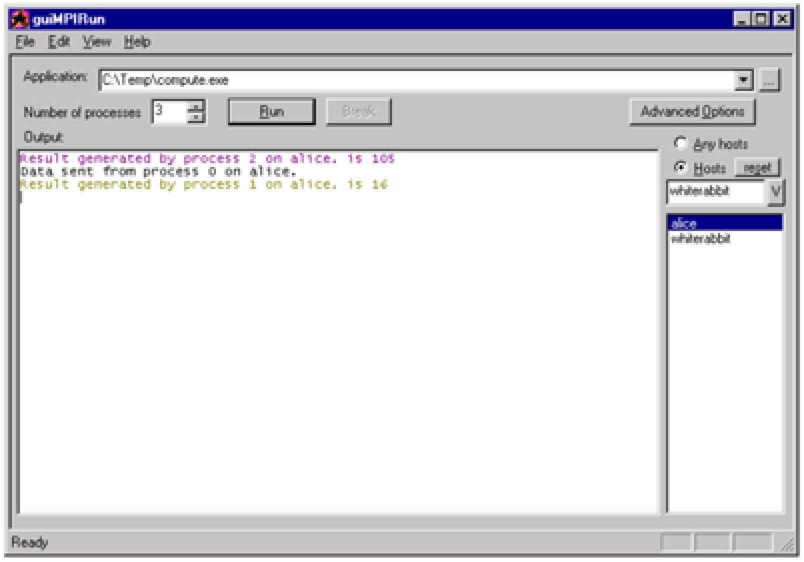
The same data is sent to the process running the **multiplier** method (process 2) with the tag **MULT_MSG**. Note that both send operations are synchronous and will block until the data has been received.

The **adder** function (lines 9-20) creates an array that will act as a buffer to receive data into. The **MPI_Recv** function call (line 13) synchronously receives the data sent by the process identified by the fourth parameter (process 0) with the tag **ADD_MSG**. The **status** variable can be used to obtain further information about the message, as well as any error conditions that have occurred.

Error handling is an important part of any application, especially in a distributed processing environment. MPI provides a number of features for detecting and handling errors that are not discussed in this paper. For more information, see the MPICH Web site*.

Once the information has been received, the code at lines 14-17 sums the values, and the result is then printed out. The **multiplier** method (lines 22-35) is similar, except that it receives a message with the tag **MULT_MSG** and computes the product of the data items.

Figure 1 shows the application running using the GUI version of the MPIRun utility. Note that the number of processes has been set to 3. You should also notice that, for convenience when debugging applications that run as parallel processors, the output of each process is color-coded. The order of the output is not guaranteed, however, as the way in which the processes are scheduled will depend on many factors.



Application: C:\Temp\compute.exe

Number of processes: 3

○ Any hosts
● Hosts: whiterabbit

alice
whiterabbit

```
Result generated by process 2 on alice. is 105
Data sent from process 0 on alice.
Result generated by process 1 on alice. is 14
```

Figure 1. The application executing using MPIRun.

The SPMD Model: Identical Tasks over Different Data

In the Single Program Multiple Data (SPMD) model, all processes execute the same code but use different data. This model is useful for performing operations that naturally partition themselves into small, independent pieces that can be performed autonomously. The MPIRun utility provides an option that allows multiple processes to be started over the same application code. Each process will be placed in the same group and can send or receive messages from other processes in the group.

The following code shows a simple application that exploits this model. The program computes the sum of a series of numbers, breaking the problem down into several parts. Each part is executed by a separate process started by the MPI runtime.

```
#include "mpi.h"

#include "stdio.h"


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

{

int myid, numprocs, datastart, dataend, i, namelen;

double localsum = 0, totalsum;

char processorname[MPI_MAX_PROCESSOR_NAME];


MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myid);

MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

MPI_Get_processor_name(processorname, &namelen);


datastart = myid * numprocs;

dataend = myid * numprocs + numprocs - 1;

for (i = datastart; i <= dataend; i++)

{

localsum += i;
35
36 }
37
38 printf("Process %d on %sData is %f",
39
40 myid, processorname, localsum);
41
42
43 MPI_Reduce(&localsum, &totalsum, 1, MPI_DOUBLE,
44
45 MPI_SUM, 0, MPI_COMM_WORLD);
46
47 if (myid == 0)
48
49 {
50
51 printf("Total sum is %f", totalsum);
52
53 }
54
55
56 MPI_Finalize();
57
58 }
```

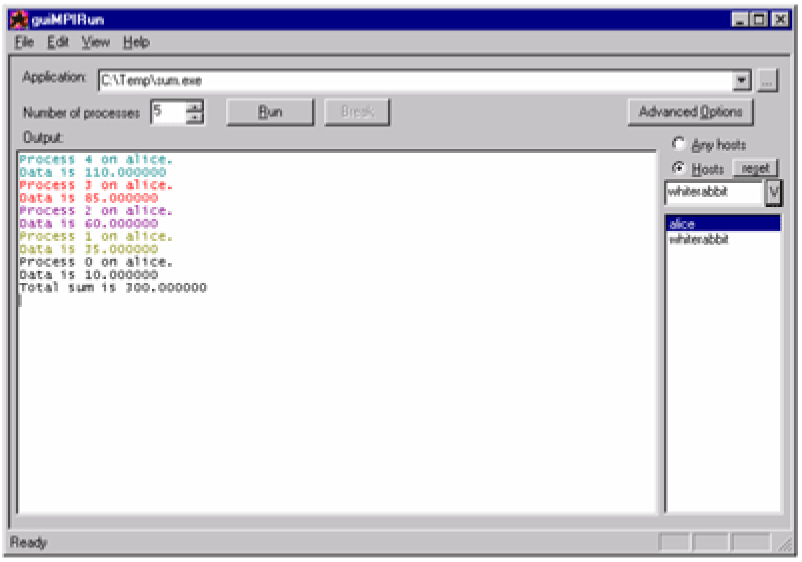
The statements between lines 15 and 20 compute the sum of a set series of values based on the process ID and the number of processes. Process 0 will compute the sum of the values 0 to n-1 (where n is the number of processes), process 1 will compute the sum of the values n to 2n-1, and so on. The result is stored in the variable **localsum** in each process. The value computed by each process is then displayed (lines 21 – 22).

The key part of this application is the call to **MPI_Reduce** in lines 24 and 25. When each process invokes **MPI_Reduce**, it supplies a buffer containing its own computed sum (**&localsum**) as the first parameter. Parameters 3 and 4 indicate the number of values being returned (1), and their data types (**MPI_DOUBLE**) respectively. Parameter 6 (0) specifies which process receives the reduced result, parameter 5 (**MPI_SUM**) indicates the type of reduction to perform, and parameter 2 (**&totalsum**) identifies an output buffer for storing t he result.

When this function is called, all processes (including process 0) will supply their locally computed sums, which will be aggregated together and the result placed in the **totalsum** variable of process 0. This value is then displayed.

MPI defines a number of predefined operations that can be used as the reduction function besides **MPI_SUM**, including **MPI_MIN**, **MPI_MAX**, **MPI_PROD**, as well as a variety of logical reduction operations. You can also create user-defined operations as long as they are commutative (the order in which the operands are generated is not guaranteed).

Figure 2 shows the output of the application from the Sum application when run using five processes. Notice the partial sums computed by each process, as well as the total sum output by process 0.



Application: C:\Temp\sum.exe
Number of processes: 5
Any hosts [✓ Hosts]
whiterabbit
alice
whiterabbit

```
Process 4 on alice.
Data is 110.000000
Process 3 on alice.
Data is 85.000000
Process 2 on alice.
Data is 60.000000
Process 1 on alice.
Data is 35.000000
Process 0 on alice.
Data is 10.000000
Total sum is 300.000000
```

Figure 2. The Sum application executing using MPIRun.

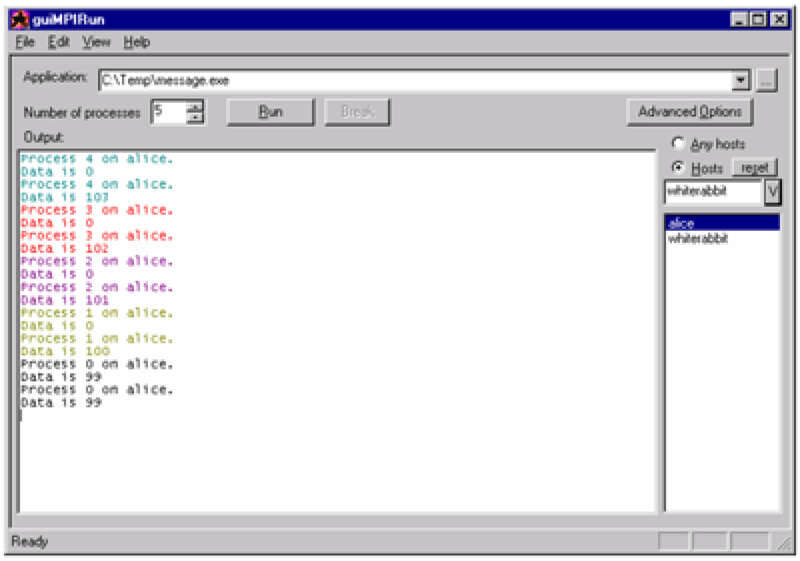
As a second example of the SMPD model, shown below, shows another simple program that uses **MPI_Bcast** to broadcast some data to each process that it uses to compute a value, which is then displayed.

```
#include "mpi.h"

#include "stdio.h"

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

{

int myid, numprocs, data = 0, namelen;

char processorname[MPI_MAX_PROCESSOR_NAME];


MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myid);

MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

MPI_Get_processor_name(processorname, &namelen);


if (myid == 0)

{

data = 99;

}

printf("Process %d on %sData is %d",

myid, processorname, data);


MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

data += myid;

printf("Process %d on %sData is %d",

myid, processorname, data);


MPI_Finalize();

}
```

The **data** variable is initially set to zero, but lines 13-16 change its value to 99 in process 0. Lines 17-18 display the current value of this variable in all processes. The **MPI_Bcast** function call at line 20 sends a message to all processes. The way in which this function works differs depending on whether it is invoked by the process doing the broadcasting or the receiving processes.

The fourth parameter (0) indicates which process is broadcasting the message; all other processes act as receivers. The first parameter (**&data**) specifies a buffer indicating the data being broadcast when process 0 executes this statement. The same parameter specifies a buffer that will be populated with the broadcast data for all other processes. The second (1) and third (**MPI_INT**) parameters indicate the number and data type of the values being sent.

Line 21 adds the rank of the executing process to the data received, and the result is printed out in lines 22-23. Figure 3 shows the output of this application (the Message application) when run using MPIRun with five processes:



Application: C:\Temp\message.exe

Number of processes: 5

Hosts: whiterabbit, alice, whiterabbit

Output:

```
Process 4 on alice.
Data is 0
Process 4 on alice.
Data is 103
Process 3 on alice.
Data is 0
Process 3 on alice.
Data is 102
Process 2 on alice.
Data is 0
Process 2 on alice.
Data is 101
Process 1 on alice.
Data is 0
Process 1 on alice.
Data is 100
Process 0 on alice.
Data is 99
Process 0 on alice.
Data is 99
```

Figure 3. The Message application executing using MPIRun.

Notice the values of the **data** variable in each process before and after the broadcast operation.

Conclusion

The MPI specification allows vendors to create portable libraries that can take advantage of multiple processors to provide high performance. An increasing number of implementations covering most common platforms are becoming available.

MPI is well-suited to a range of environments and can function using multiple processors in a single high-performance computer as well as across a network of lower-powered machines. MPI comprises a large number of functions, although it is possible to write complete working applications using a small subset of them.

MPI is not intended to be a replacement for distributed computing technologies such as CORBA, DCOM, or Web services. Furthermore, despite the fact that MPI will operate in a clustered environment, its primary purpose is to deliver throughput, rather than fault-tolerance.