Cluster Programming with MPI

Number Crunching

Linux clusters are all the rage. As this article shows, programming a cluster with MPI (Message Passing Interface) need not be too difficult. Of course, we realize that most readers will not have a cluster of Linux machines under their desks, so the sample programs will run on any normal PC. BY HEIKO BAUKE

MPI is the indisputed leader when it comes to programming Linux clusters and other massively parallel computers.

The development of parallelized programs leads to a number of issues. The load needs to be evenly spread and the work performed by these multiple simultaneous processes needs coordinating.

Message Exchanges

Message exchanges are the central concept at the heart of MPI. Many processes combine to solve a large problem and talk to each other to coordinate their work. The concept is highly generic and can be implemented on a variety of computer architectures. This makes it more or less irrelevant to the programmer whether the program will run on a large-scale SMP machine or throughout the night on a collection of networked, and otherwise idle, office machines.

MPI programs are usually designed on the basis of the SPMD paradigm (Single Program, Multiple Data). Multiple processes run identical program code, but each process handles different data. Each process is assigned a unique rank that influences its execution.

Implementations

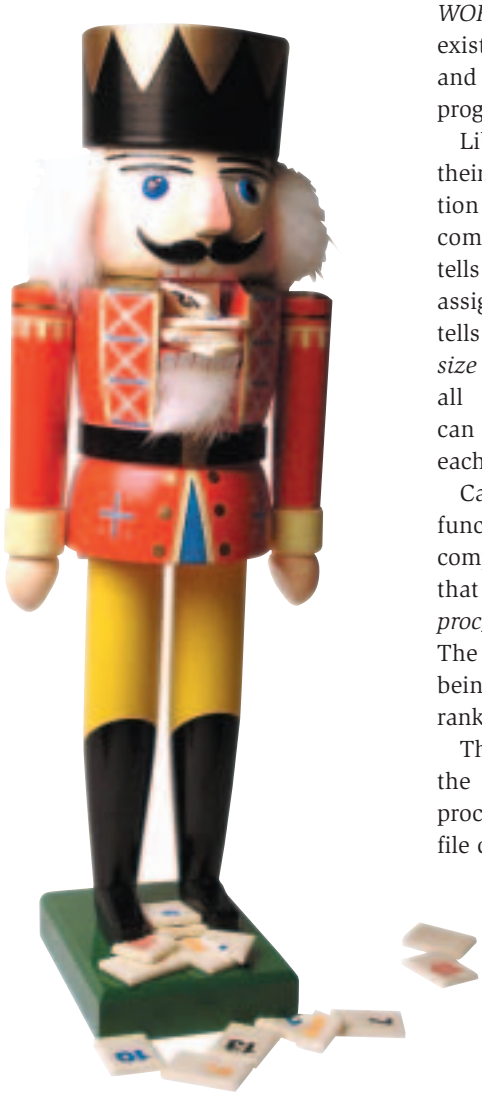
The MPI standard defines only one API (or three to be more precise, one each for Fortran, C, and C++). Every super-computer manufacturer offers its own implementation, optimized for its own hardware. Besides these there are also free implementations such as LAM-MPI [3] and MPICH [2].

How an MPI program is compiled, debugged and launched will depend on the implementation. The information in this article is based on LAM-MPI and the C++ API. The installation steps for LAM are well documented, and in case of installation issues, competent help is always at hand via the LAM mailing list [4].

Hello World!

Listing 1 shows a simple MPI program that outputs the names of the computers running the processes.

The header file included here, *mpi.h*, provides MPI specific prototypes. All the MPI classes and functions reside within the *MPI* namespace. Each MPI program is surrounded by the *MPI::Init* and *MPI::Finalize* tags. MPI calls are illegal if they occur before *MPI::Init* has initialized the MPI environment, or after *MPI::Finalize* has closed the MPI environment.



The communicator is one of MPI's fundamental concepts. It groups processes that can exchange messages. Communicators are implemented by the *MPI::COMM* class. The *MPI::COMM_WORLD* communicator instance always exists; it contains all the MPI processes and is quite sufficient for simple programs.

Libraries who need to encapsulate their communication before the application functions use their own communicators. The *Get_size* method tells you how many processes have been assigned to a communicator, *Get_rank* tells you a process's rank. The *rank* and *size* size variables are local, just like all variables in MPI programs, and can thus assume different values for each process.

Calling the *Get_processor_name* function places the name of the computer running the process in a buffer that the *proc_name* pointer references. *proc_name_length* contains the length. The string is null terminated before being sent to the process with the rank zero.

This process uses a for loop to receive the processor names of the other processes and store them to a file. If the file cannot be opened, the *Abort* method can be used to kill all the processes.

If you are wondering why the process ranked zero (and not 42, for example) collects the data, this is because there will always be a zero rank process. This ensures that the

data will be sent to an existing process, and that the program will run independently of the number of processes. The number of processes is stipulated when launching the program.

All non-zero processes use the *Send* method to send their strings. The prototype of the send method is as follows:

```
void Comm::Send(const void*
buf, int count, const Datatype&
datatype, int dest, int tag)
const
```

It expects a pointer to a contiguous memory area first. *count* stipulates the number of elements of the *datatype* type should be sent to the process ranked *dest*.

Table 1 shows C++ datatypes. *tag* is used to separate the messages, which are uniquely identified by their communicator, a datatype and a tag. The receiving process must possess a receiving method that matches the sending method. So the communicator, datatype, and tag of the sending and receiving methods must match. The prototype of the receiving method is called *Recv*:

```
void Comm::Recv(void* buf, int
count, const  Datatype&
datatype, int source, int tag)
const
```

The arguments mean the same as in the sending method, but in this case they refer to the source of the message, instead of the target, and specify a receive buffer rather than a send buffer. Programmers must ensure that the receive buffer is large enough.

Our Hello World program outputs the processor name to a file and not to standard output. This makes sense because MPI programs are not typically bound to a terminal. Additionally, the MPI standard does not actually specify what happens when data is output to *STDOUT* or *STDERR*. Programmers wanting to create portable code are well advised to avoid output of this kind. The LAM implementation passes any standard output to *STDOUT* by the processes to *mpirun*, other implementations may standard output to */dev/null*.

The program can be compiled with any C++ compiler, although using the

Table 1: An MPI datatype is available for each standard C++ datatype

MPI Datatype	C++ Datatype
MPI::CHAR	char
MPI::WCHAR	wchar_t
MPI::SHORT	signed short
MPI::INT	signed int
MPI::LONG	signed long
MPI::SIGNED_CHAR	signed char
MPI::UNSIGNED_CHAR	unsigned char
MPI::UNSIGNED_SHORT	unsigned short
MPI::UNSIGNED	unsigned int
MPI::UNSIGNED_LONG	unsigned long int
MPI::FLOAT	float
MPI::DOUBLE	double
MPI::LONG_DOUBLE	long double
MPI::BOOL	bool
MPI::COMPLEX	Complex<float>
MPI::DOUBLE_COMPLEX	Complex<double>
MPI::LONG_DOUBLE_COMPLEX	Complex<long double>

Listing 1: Hello World Program with MPI

```
// hello_world.cc
//
// Hello World Program

#include <cstdlib>
#include <iostream>
#include <fstream>
#include "mpi.h"

using namespace std;

int main(void) {

  MPI::Init();  // initialisiere
MPI

  int
rank=MPI::COMM_WORLD.Get_rank();
// ascertain own rank
  int
size=MPI::COMM_WORLD.Get_size();
// and number of processes

  const int
max_proc_name_length=MPI::MAX_PRO
CESSOR_NAME+1;
  int proc_name_length;
  char *proc_name;
  try {
    proc_name=new
char[max_proc_name_length];
  }
  catch (bad_alloc &e) {  // not
enough memory

MPI::COMM_WORLD.Abort(EXIT_FAILUR
E);  // terminate all processes
  }

MPI::Get_processor_name(proc_name
, proc_name_length);  // which
hostname?

proc_name[proc_name_length]='\0';

  if (rank==0) {  // rank 0 is
indicated
    ofstream
out("hello_world.out");  //
create output file
    if (!out)  // error =>
terminate processs

MPI::COMM_WORLD.Abort(EXIT_FAILUR
E);
    // Receive status reports of
other ranks and write to file
    for (int i=0; i<size; ++i) {
      // Receive string and write
to file
      if (i>0)

MPI::COMM_WORLD.Recv(proc_name,
max_proc_name_length, MPI::CHAR,
i, 0);
      out << "Hello World! My
rank is " << i << " of " << size
<< ". "
          << "I am running on "
<< proc_name << "." << endl;
    }
  } else
    // Send string

MPI::COMM_WORLD.Send(proc_name,
proc_name_length+1, MPI::CHAR, 0,
0);

  MPI::Finalize();  // terminate
MPI

  return EXIT_SUCCESS;
}
```

Wrapper compiler, which is normally supplied with the MPI implementation will ensure that the required libraries and header files are found. The Wrapper compiler for LAM-MPI is called *mpiCC*.

```
mpiCC -o hello_world➥
hello_world.cc
```

After compling, simpy enter:

```
mpirun -np 4 hello_world
```

to launch four instances of the hello_world program. The *-np* parameter defines the number of processes. Before doing so, ensure that the LAM daemons have been launched on all the machines that will be running MPI programs later. To do so, you will need to create a *nodes* file that contains a hostname in each line (the simplest case would be just *localhost*) and launch the daemons by typing:

```
lamboot -v nodes
```

You only need to launch the daemons once. If you enter *lamhalt* to terminate the daemons, you will need to re-launch them explicitly.

Ping Pong

The few MPI functions introduced in the previous section allow you to author fairly useful programs. Network bandwidth plays an important role in the case of Beowulf clusters. A small ping pong program can be used to measure the bandwidth (see Listing 2), where a fixed length message is first passed from process 0 to process 1 and then back from process 1 to process 0. The elapsed time is measured and a mean value for multiple attempts ascertained.

You should be familiar with the MPI initialization phase from the Hello World program. The actual measurement is performed in two loops (starting in line 37 and 41 respectively). The *Barrier* is new; it ensures that a communicator's processes are synchronized. Each pro-

Listing 2: Measuring bandwidth with Ping Pong

```
// ping_pong.cc
//
//
//
// Ascertain bandwidth in
relation to packet size


#include <cstdlib>
#include <iostream>
#include <fstream>
#include "mpi.h"

using namespace std;

const int
max_packet_size=0x1000000;  //
maximum size of a message
const int count=250;  // number
of messages per measurement
char buff[max_packet_size];  //
Send and receive buffer

int main(void) {

  MPI::Init();  // initialize MPI

  int
rank=MPI::COMM_WORLD.Get_rank();
// ascertain own rank
  int
size=MPI::COMM_WORLD.Get_size();
// and number of

  if (size==2) {  // exactly two
processes needed for this process
    ofstream out;
    if (rank==0) {  // Open
output file
      out.open("ping_pong.dat");
      if (!out)

MPI::COMM_WORLD.Abort(EXIT_FAILUR
E);
      out << "# Data throughput
relative to packet size" << endl
          << "# Time resolution"
<< MPI::Wtick() << " s" << endl
          << "# Packet size\tmean
time\tmaximum time" << endl;
    }

    // Loop through various
packet sizes
    int packet_size=1;
    while
(packet_size<=max_packet_size) {
      double t_av=0.0;
      double t_max=0.0;
      // Loop through multiple
messages
      for (int i=0; i<count; ++i)
{

MPI::COMM_WORLD.Barrier();   //
Sync processes
        // Send and/or receive
messages
        if (rank==0) {
          double t=MPI::Wtime();
// Starting time

MPI::COMM_WORLD.Send(buff,
packet_size, MPI::CHAR, 1, 0);

MPI::COMM_WORLD.Recv(buff,
packet_size, MPI::CHAR, 1, 0);
          t=(MPI::Wtime()-t)/2.0;
// Time vector
          t_av+=t;
          if (t>t_max)
            t_max=t;
        } else {

MPI::COMM_WORLD.Recv(buff,
packet_size, MPI::CHAR, 0, 0);

MPI::COMM_WORLD.Send(buff,
packet_size, MPI::CHAR, 0, 0);
        }
      }
      if (rank==0) {  // Output
results
        t_av/=count;
        out << packet_size <<
"\t\t" << t_av << "\t" << t_max
<< endl;
      }
      packet_size*=2;  // Double
packet size
    }

    if (rank==0)  // Close output
file
      out.close();
  }

  MPI::Finalize();  // Terminate
MPI

  return EXIT_SUCCESS;
}
```

cess interrupts the running program while the communicator's processes call *Barrier*. Synchronization is important to avoid misleading results. The *MPI::Wtime* function is used to measure the time in seconds since starting an internal timer. The *MPI::Wtick* specifies the granularity of the timer.

Figure 1 shows the results. The throughput is seen to improve proportionally to the message size for Ethernet and Myrinet until the maximum bandwidth of 64 kilobytes is reached. The transfer time for the message roughly comprises two parts.

One of them is the constant delay (approx. 9 microseconds for Myrinet, and approx. 75 microseconds for Ethernet) and the actual transfer time, which is proportional to the message length. The delay is significant for smaller messages.

When messages are exchanged via shared memory, a peak in the transfer rate distribution can be observed. This depends on the hard and software used. Packets of more than 64 Kbytes will not fit in fast access cache memory, and the transfer rate drops drastically. In the case of very large messages, the bandwidth is restricted by the slower main memory.

Collective Communication

Send and *Recv* are used to exchange messages between two processes, however, MPI provides communication methods that involve all the processes handled by a communicator.

A process, needing to communicate to the other processes, could use a loop to call *Send* and thus transfer these results to every other process. The other processes could then call *Recv* to receive the results. The overhead for this method increases relative to the total number of processes.

The *Bcast* provides a simpler and more efficient method. *Bcast* distributes a message to all processes in a time proportional to $\log_2$ *no. of processes*. The prototype for *Bcast* is as follows:

```
void Comm::Bcast(void *buffer,
int count, const Datatype&
datatype, int root) const
```

The first argument is a pointer to a buffer that stores the data to be sent or received. The buffer contains *count* elements of the datatype specified by the third argument. The last argument specifies the process whose data are to be distributed to the other processes.

Bcast communications use an extremely effective tree. Process 0 sends the data to process 4 in step 1. Now two processes have the data (0 and 4). Both of these send the data simultaneously to processes 2 and 6. Finally, the four processes with the data, 0, 2, 4, and 6, send the data to processes 1, 3, 5, and 7.

Reduce is another useful function that does the exactly the opposite of *Bcast*. The prototype for *Reduce* is as follows:

```
void Comm::Reduce(const void
*sendbuf, void *recvbuf, int
count, const Datatype &datatype
,const Op &op, int root) const
```

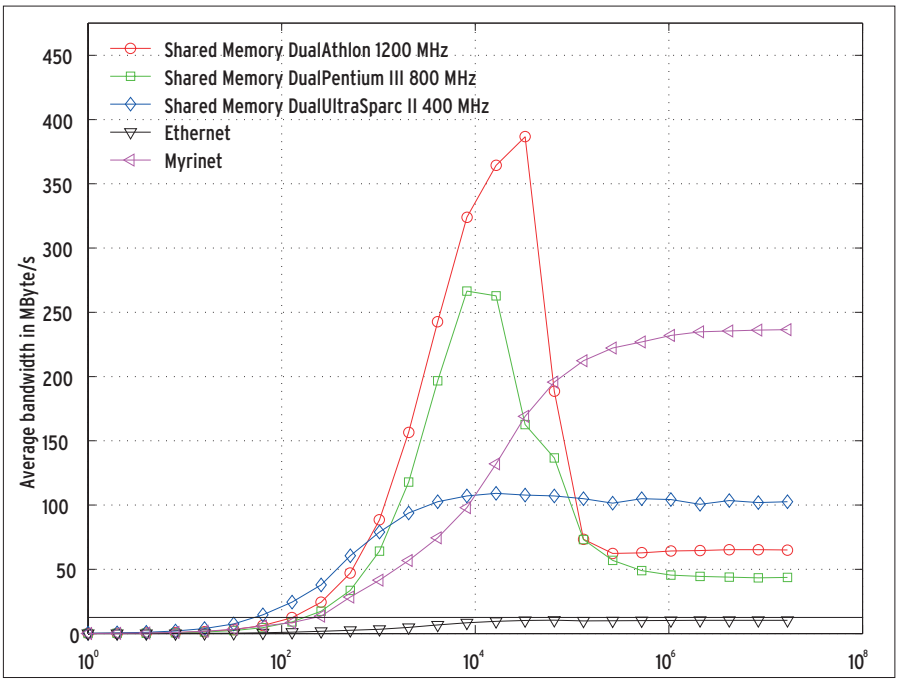


Figure 1: Results of bandwidth test using shared memory, Ethernet, and Myrinet. Myrinet data courtesy of Tobias Czauderna and Andreas Herzog (Univ. Magdeburg)

INFO

[1] MPI Forum: *http://www.mpi-forum.org*

[2] MPICH: *http://www-unix.mcs.anl.gov/mpi/mpich*

[3] LAM-MPI: *http://www.lam-mpi.org*

[4] LAM Mailing list: *http://www.lam-mpi.org/MailArchives*

[5] P. Pacheco; Parallel Programming with MPI; Morgan Kaufmann Publishers; 1996

[6] Brian Hayes; Digital Diffraction; American Scientist; Nr. 5 1996: *http://www.americanscientist.org/issues/comscig96/compscig96-05.html*

[7] Implementations: *http://www.lam-mpi.org/mpi/implementations/*

[8] *http://www.lam-mpi.org/tutorials/*

[9] Ian Foster; Designing and Building Parallel Programs; Addison-Wesley; 1995: *http://www-unix.mcs.anl.gov/dbpp/*

[10] Tina: *http://tina.nat.uni-magdeburg.de*

[11] Author's homepage: *http://tina.nat.uni-magdeburg.de/heiko/*

[12] Additional information: *ftp://www.linux-magazin.de/pub/listings/magazin/2003/05/MPI*

Table 2: MPI Operators for *Reduce*

Operand	Meaning
MPI::MAX	Maximum
MPI::MIN	Minimum
MPI::SUM	Sum
MPI::PROD	Product
MPI::LAND	logical AND
MPI::BAND	binary AND
MPI::LOR	logical OR
MPI::BAND	binary OR
MPI::LXOR	logical exclusive OR
MPI::BXOR	binary exclusive OR
MPI::MAXLOC	Maximum and occurrence of maximum
MPI::MINLOC	Minimum and occurrence of minimum

Reduce uses the operation specified by the *op* operator to collect the data that all processes have placed in the buffer pointed to by *sendbuf*. The results are placed in the *recvbuf* of the process referenced by *root*. The send and receive buffers contain *count* elements. The *count*, *datatype*, *op*, and *root* arguments must be identical for all processes.

The case where local data belonging to one process needs to be distributed across all processes is handled by the *Scatter* function . In contrast to this, *Gather* collates distributed data within a single process. The prototype for *Scatter* is as follows:

```
void Comm::Scatter(const void
*sendbuf,  int sendcount, const
Datatype &sendtype, void
*recvbuf, int recvcount, const
Datatype &recvtype, int root)
const
```

and *Gather* as follows:

```
void Comm::Gather(const void
*sendbuf, int sendcount, const
Datatype &sendtype, void
*recvbuf, int recvcount, const
Datatype &recvtype, int root)
§§ const
```

root specifies the process by or to which *sendtype* type data are sent. *sendcount* specifies the amount of data each processor will send or receive. The *sendcount* and *recvcount*, and *sendtype* and *recvtype* arguments are typically identical.

In listing 3. each process generates a pseudo-random number, and then the maximum, and minimum values, and the sum of the pseudo-random numbers is calculated. Information is collected by process zero and written to a file.

Digital Diffraction

The program on the web site [12] creates parallel diffraction images. It calculates the diffraction between spherical waves originating at various points on the screen. The article at [6] provides additional details.

After initializing MPI, the process ranked zero reads a configuration file that contains geometrical data. A call to *Bcast* distributes the geometrical data to all other processes.

The program uses geometrical distribution to parallelize the calculations. The screen area for which the intensity distribution will be calculated is divided into narrow horizontal bands. After this calculation, a call to *Reduce* discovers the intensity of the brightest point. *Gather* collects the image data to allow process zero to output a portable graymap file.

Conclusion

MPI is extremely powerful. Amongst other things MPI provides non-blocking communication, additional collective communication functions, derived datatypes and special topologies. [9] provides a discussion of several generic aspects of parallel programming. ■

Listing 3: Demonstration of collective message exchange methods

```
// collective.cc
//
// Sample program for collective
communication

#include <cstdlib>
#include <fstream>
#include <vector>
#include "mpi.h"

using namespace std;

int main(void) {

  const int count=8;
  vector<int> rand_nums(count),
max(count), min(count),
sum(count),
    all_rand_nums;

  MPI::Init();  // initialize MPI

  int
rank=MPI::COMM_WORLD.Get_rank();
// ascertain own rank
  int
size=MPI::COMM_WORLD.Get_size();
// and number of processes

  srand(7*rank);  // initialize
random number generator
  for (int i=0; i<count; ++i)  //
throw dice
    rand_nums[i]=rand()%1000;

  if (rank==0)  // allocate space
at receiving process

all_rand_nums.resize(count*size);
  // Collect data

MPI::COMM_WORLD.Gather(&rand_nums
[0], count, MPI::INT,

&all_rand_nums[0], count,
MPI::INT, 0);

  // Calculate maximum, minimum,
and sum and send to rank 0

MPI::COMM_WORLD.Reduce(&rand_nums
[0], &max[0], count, MPI::INT,
MPI::MAX, 0);

MPI::COMM_WORLD.Reduce(&rand_nums
[0], &min[0], count, MPI::INT,
MPI::MIN, 0);

MPI::COMM_WORLD.Reduce(&rand_nums
[0], &sum[0], count, MPI::INT,
MPI::SUM, 0);

  if (rank==0) { // Save results
    ofstream
out("collective.out");
    for (int i=0; i<size; ++i) {
      out << "Process " << i << "
:";
      for (int j=0; j<count; ++j)
        out << "\t" <<
all_rand_nums[i*count+j];
      out << endl;
    }
    out << endl << "Maxima :";
    for (int i=0; i<count; ++i)
out << "\t" << max[i];
    out << endl << "Minima :";
    for (int i=0; i<count; ++i)
out << "\t" << min[i];
    out << endl << "Summen :";
    for (int i=0; i<count; ++i)
out << "\t" << sum[i];
    out << endl;
  }

  MPI::Finalize();   // terminate
MPI

  return EXIT_SUCCESS;
}
```