MPI: Early Experiences

Nicholas P. Cardo, Sterling Software, Inc., Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, M/S 258-6, Moffett Field, CA 94035-1000 USA

ABSTRACT: *The Message Passing Interface (MPI) provides an alternative to multitasking for parallel processing. This paper discusses some of the pros and cons of using MPI from a user's perspective and a system's perspective. Sample programs are shown to demonstrate the strengths and weaknesses of MPI.*

Price performance is an important aspect of any computational facility. Harnessing the power of all of the systems provides an effective way of providing cost effective computing. The purchasing of larger expensive supercomputers is not necessarily the right approach. Large scale supercomputers are a very expensive component of a computer facility. However, an alternate approach is to purchase smaller high performance computers and distribute the problem across multiple systems. This provides the potential for high performance at an attractive price.

All discussions contained in this paper are based on an alpha version of Message Passing Interface (MPI) from Cray Research Inc. on a C90 system.

1 What is MPI?

Parallel computing is complicated by nature. By distributing the problem across multiple systems, the combined computing power of all the systems can be applied to the problem. However, individual systems lack the capabilities to synchronize applications and share data between systems. MPI was developed to address this problem area of parallel computing.

MPI provides a portable parallel programming environment. It is intended to provide an interface that is capable of working on machines with different operating systems and architectures.

Cluster computing takes groups of smaller machines for use in distributed parallel computing. The combined computing power of each system in the cluster contributes to the overall performance of applications. Portability plays an important part when working with a cluster of heterogenous systems. The potential is there for each node in a cluster to be of a different architecture. Although portability plays a less important roll on a homogenous cluster, it is still an important aspect of parallel computing. Applications need to be able to scale to fully utilize the processing power available. Part of this scalability is being able to move an application to different platforms. Without a portable interface, this would not be possible.

2 MPI Components

MPI is a complete package which is incorporated into the software development process. An environment variable `MPIR_HOME` is used to identify the location where MPI is installed. This will allow for customized installations of MPI by installing it in site desired locations. In order to use MPI, the `MPIR_HOME` environment variable must be set to the root directory of MPI.

2.1 MPI Directory Tree

`MPIR_HOME` actually is the root directory for the MPI installation which contains header files, libraries, machine identification file and the `mpirun` script. Figure 3 shows the `MPIR_HOME` directory tree.

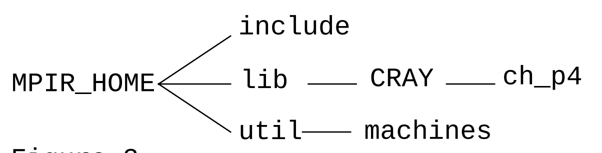


Figure 3.

The files for using MPI are located as follows:

```
$(MPIR_HOME)/include/*.h
$(MPIR_HOME)/lib/CRAY/ch_p4/libmpi.a
$(MPIR_HOME)/util/mpirun
$(MPIR_HOME)/util/machines/machines.CRAY
```

`CRAY` refers to the architecture and the file `machines.CRAY` contains the default information necessary to initiate MPI processes. ch_p4 refers to the specify machine type of the CRAY architecture. Since MPI is portable, being able to differentiate between machines and architectures is crucial.

2.2 Header Files

A series of header files for MPI function prototypes and option definitions are provided. These header files are:

```
binding.h       mpi++P.h        mpi_errno.h
mpir.h          sbcnst.h        dmpi.h
mpi.h           mpif.h          mpisys.h
dmpiatom.h      mpi_ad.h        mpiimpl.h
mpiuser.h       mpi++.h         mpi_bc.h
mpiprof.h       patchlevel.h
```

2.3 Libraries

The MPI functions are contained in the libmpi.a library. All function prototypes and defined function options are contained in the MPI header files. It is important to include the following library when compiling applications:

```
$(MPIR_HOME)/lib/libmpi.a
```

2.4 mpirun

MPI applications are initiated by the mpirun script. Command line arguments define the parallel parameters for the job such as the number processes to create.

Since MPI is portable, the mpirun script provides the ability to specify machine type and architecture. This allows applications to be distributed in a hetergenous computing environment yet still use a common message passing interface. Although defaults are obtained from a configuration file, the command line options allow the defaults to be overriden to maximize the flexibility to fully utilize available resources.

2.5 Special Files

An MPI application can be customized to run a specific number of processes on specific hosts. The process group file contains hostnames, process counts and location of the executable. A copy of the executable must exist on each host. A sample process group file would be:

```
host1     2  /home/user/prog1
host2     6  /home/user/prog1
```

This identifies that the MPI application is to utilize two systems with a total of 8 processes along with what the executable is and its location.

It is not a requirement that the executable reside in the same directory path on both systems. The following examples demonstrate this:

```
host1     2  /home/user/prog1
host2     6  /u/user/prog1
```

3 Using MPI

Parallelizing a code with MPI can be a time consuming and complicated task. MPI consists of a set of functions which are inserted into an application to control interprocess communications and degrees of parallelness.

3.1 Constructing Programs

MPI is not capable of determining how to parallelize an application or constructing the appropriate calls to insert into an application. This is left as an exercise to the user.

A typical Fortran MPI application will begin with a sequence of calls similar to the following.

```
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
```

These three calls will perform MPI execution initialization, process ranking, and establish the size of the process group. The MPI_COMM_RANK() call will establish the process with the highest rank, the root process.

When the computation is finished, the MPI application needs to be terminated. This is accomplished with the following code segment:

```
call MPI_FINALIZE(ierr)
stop
end
```

Communication between processes is accomplished through a series of send and receive operations. The process of rank, the root process, can broadcast a message to all other processes with the MPI_BCAST() call. A sample call to broadcast the variable a to all other processes would be:

```
call MPI_BCAST(a, cols, MPI_DOUBLE_PRECISION,
master,MPI_COMM_WORLD, ierr)
```

The application can also send information from any process to all other processes. For example, to send a to other processes, the call might look like:

```
call MPI_SEND(a, 1, MPI_DOUBLE_PRECISION, master,
anstype,MPI_COMM_WORLD, ierr)
```

A process can receive messages as well. For example, to receive a message into buffer, the call might look like:

```
call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION,
master,MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
```

These calls show a simplistic view of MPI communication. There are over 100 different MPI calls for accomplishing all kinds of interprocess communication and synchronization.

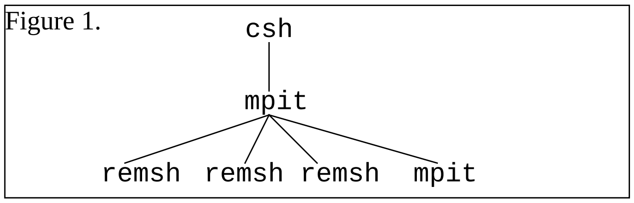
3.2 Starting Programs

MPI job initiation is accomplished with the mpirun shell script. Command line arguements are processed to customize the running of the MPI application. For example, suppose the program mpit is to run with 4 processes on the invoking system. This is accomplished with:

```
% mpirun -np 4 mpit
```

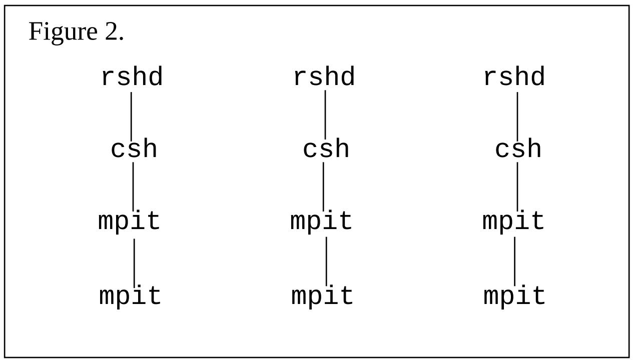
Four process chains are created as a result of this. The first process chain is derived from the process shell that invoked mpirun. Figure 1 shows the process tree hierarchy from the process invoking mpirun. Take note of the two mpit processes related as parent and child. The parent mpit process will create the multiple process streams to be used for computation. The child mpit process will actually be part of the computational algorithm.

Figure 1.



As a result of the 3 `remsh` processes, three additional process chains are formed. Figure 2 shows the process tree associated with each of these 3 process trees. The three `mpit` processes

Figure 2.



attached to the bottom of these process trees are used for performing the computation.

The first instance of the application must run on the system that `mpirun` was invoked on. However, all other processes can run on alternate systems. The control of where these processes run can be accomplished in the process group file specified by the `-p4pg` option of the `mpirun` command. This is where the power of distributed processing with MPI comes from. Suppose there were 5 CRAY J932 systems, all capable of performing computations. This allows for 160 processors to be applied to the application, without any code changes. The process group file, named `pgfile` in this example, for running the `mpit` application would look like:

```
J1      32    /home/user/mpit
J2      32    /home/user/mpit
J3      32    /home/user/mpit
J4      32    /home/user/mpit
J5      32    /home/user/mpit
```

The new invocation of `mpirun` would be:

```
mpirun -p4pg pgfile -np 160 mpit
```

This provides the computational capacity of 160 Y-MP processors to work on a single application.

3.3 Observations

Figure 1 shows that processes are started via `remsh`. For every process stream requested via the `-np`, number of processors, option for `mpirun`, a `remsh` process is created on initiating host. A `rshd` process then forks a shell process to fork the executable. If 160 processes were requested, 159 `remsh` process would be forked. From a system administration perspective, many `remsh` processes can be expected to be seen on the initiating host if MPI were heavily used.

The number of processes running the application is double the number of processes requested. However, only the number requested actually perform computational work and accumulate cpu time. Figures 1 and 2 show the relationship of all the copies of the executable.

3.4 Creating Batch Jobs

Running an MPI application from a batch job requires no additional customization. For example, suppose the mpit application required 100mw and 4 hours of cpu time. The batch job might look like:

```
#QSUB -lT 14400
#QSUB -lM 100mw

mpirun -np 4 mpit
```

The only difference between an MPI versus a non-MPI batch job would be the use of `mpirun` to start the MPI application.

4 Problems Encountered

If the rank process abnormally terminated, the remaining process streams would continue to run. The only way to recover from this problem is to track down and kill all processes on each system.

Since each process stream is created through `remsh`, each stream is itself a session. Therefore any session limits are not inclusive of all the processes performing the work. The end result is that a 4 hour job can actually get 4 hours on each process stream. The total delivered cpu time will then be the product of 4 hours times the number of processes requested with `mpirun`. Furthermore, when the MPI application is run under NQS, the NQS queue limits are not inherited but rather limits from the `udb`. Software Problem Report (SPR) 98158 was opened to address this problem and as a result a fix was developed so that `fork`/`exec` are used on the local system rather than `remsh`. While this addresses the problem on the local system, it does not seem to address processes on remote systems.

5 Summary

Cost effective high performance computing is essential to any computer facility. Advancements in distributed computing are making it possible to apply more processing power to an application without the costs of purchasing larger and faster computer systems. By harnessing the computational power of multiple smaller systems, high performance computing can be accomplished with a significant cost reduction.

The Message Passing Interface is playing a key role in the development of distributed parallel application development. The problems of internode communications and synchronization are being solved with packages such as MPI. The portability of MPI makes it possible for the formation of a heterogeneous cluster of compute servers.

Cray Research is evolving MPI into the Message Passing Toolkit (MPT). This will address the needs for parallel software development with the optimizations necessary to take advantage of multiprocessor parallel vector processor systems.

The compute environment of the future will most likely be a collection of systems of various architectures. This collection of systems may include a high performance vector system, moderately/massively parallel system, low end pre/post processing systems, and a cluster of medium ranged systems. Making these systems work together to solve tomorrow's problems is a key challenge. Packages such as MPI could provide the beginnings of realizing this future compute facility capable of delivering cost effective high performance computing.

The Numerical Aerodynamic Simulation Facility (NAS) at NASA Ames Research Center is active in pursuing the futures of parallel computing. United States sites are welcome to visit NAS on the World Wide Web at `http://www.nas.nasa.gov`.
The parallel efforts at NAS can be found at
`http://lovelace.nas.nasa.gov/Parallel/home.html`.
The author can be reached at `cardo@nas.nasa.gov`.

6 References

1. *USING MPI: Portable Parallel Programming with the Message-Passing Interface*, William Gropp, Ewing Lusk, and Anthony Skjellum, The MIT Press, Cambridge,MA, 1994.
2. *MPI man pages*, Cray Research Inc.
3. Users' Guide to mpich, a Portable Implementation of MPI, Patrick Bridges, Nathan Doss, Ewing Lusk, William Gropp, Anthony Skjellum, and Edward Karrels, 1995.