MPI Project Report

Brian Satzinger

December 18, 2009

Contents

1 Introduction

The message passing interface (MPI) is a standard for communication between processes in a parallel computer [3]. It allows for processes to coexist on the same computer, as well as on different computers which are members of the same cluster. In a cluster, distinct computers communicate over a common network in order to perform some task in parallel. MPI is designed to be as flexible as possible, and to support diverse hardware platforms. Programs utilizing MPI can be executed on a single computer (with the MPI libraries installed), as well as on high performance supercomputer clusters.

MPI is a standard which outlines the identity and behavior of MPI function calls, as well as special MPI data types. Specific implementations of MPI use this standard. For this report, MPICH2 was selected, because it is a prominent free and open-source MPI implementation.

MPICH2 is installed on three different single-computer environments. The first is directly on a single processor computer. The second is on four virtual machines configured into a cluster running on the same computer. The third is on a dual processor server with 16 processor cores. A non-virtual cluster was not available for performance comparison. Thus, this report compares two methods of using MPI on a single computer (either through a direct installation of MPICH2 libraries, or by building a virtual cluster), as well as the effects of a single processor core vs multiple processor cores. Notes are also provided on the configuration of the 3 example MPICH2 installations.

2 MPI Overview

MPI is a standard created by the MPI Forum, which is supported by a variety of universities, corporations, and scientific institutions. The current version of MPI is 2.2, and this version was released in September 2009. The MPI Forum gives a number of goals for MPI, including that communication should be efficient, that MPI should support heterogeneous clients, that use in common languages (C, C++, and Fortran) should be convenient, that it should be an inherently reliable communication standard, that it is thread safe, and that the interface is easily implemented on various platforms [3]. In general, MPI can be used with either commodity computer and network equipment (such as intel based PCs connected with an ethernet network), or with specialized hardware developed for supercomputing clusters.

MPI began development in 1992, with the release of MPI-1.1 in 1995. By 2001, MPI-2.0 was released. In general, each release since 1.1 is backwards compatible with the earlier versions, although they will often add new features or optimizations [3].

MPI specifies the functions used to send and receive variables between processes. The underlying process

of communication (whether by network or by shared memory or by some other mechanism) is abstracted from the MPI function call. This allows the same MPI program to be used on clusters with different hardware platforms. In general, MPI processes are each assigned a unique rank when they are started. Communication between processes is done by specifying the rank of another process with which to send or receive a message [3].

MPI provides blocking and non-blocking send and receive operations. My own experience suggests that care should be taken when using blocking send and receive operations. When many ranks attempt to exchange messages at the same time, it is possible for the order in which the messages are sent to create a deadlock among the ranks. Care must be taken with the order that send and receive operations occur. Non-blocking operations can solve many of these problems, although non-blocking operations require the program to check that the send/receive operation actually occurred before using reusing the variable. MPI provides functions for performing this check [3].

In order to guarantee cross platform compatibility, MPI defines its own data types with specific byte lengths. It is necessary to convert the data types used locally on a particular platform to the data types expected to MPI before sending them, in order to guarantee consistent behavior on all platforms. For example, MPI defines MPI_LONG_DOUBLE to be 64 bits, although C compilers may produce long doubles of different length on some platforms [3].

3 MPICH2 Overview

While MPI defines a standard, an implementation of the standard is necessary to realize an MPI program. MPICH2 is one such implementation. It is developed by the Argonne National Laboratory, and the source code is freely available online [2]. In my experience, it was compiled and installed easily on several commodity-PC platforms.

MPICH2 installs libraries on the host computer, as well as a compiler (mpicc) which automatically links programs to those libraries. A standard compiler such as GCC can also be used to compile MPI programs if it is configured with the location of the libraries. MPICH2 also includes a number of tools, such as *mpiexec*, which is used to start a parallel program, *mpdboot*, which is used to start mpd (a daemon which handles MPI communication) on a cluster. The behaviors of these tools is part of the MPI standard. MPICH2 provides implementations of these tools, written using Python [2].

The installation of MPI on Linux is straightforward for someone experienced in basic Linux operations (using make, setting file permissions, configuring network settings, etc). I will outline my experiences and procedures for installing MPICH2 in several different configurations in sections 4, 5, and 6.

4 MPICH2 On A Single Processor Host

This is the simplest case for installing MPICH2. In this configuration, parallel MPI programs will be realized as multiple processes on the same host. MPICH2 will handle inter-process communication with a shared memory space. This is useful for testing and developing MPI algorithms without access to a cluster.

The procedure for installing MPICH2 is well documented in the MPICH2 Installer's Guide [1]. In general, it assumes that Linux (or some other POSIX operating system) is the platform, although versions of MPI are available for Windows platforms. It outlines the procedure for installing MPICH2, which is to download the source from the MPICH2 website, configure the compilation scripts using the *make configure* command, compile MPICH2 using the *make* command, and install the libraries and executable using the *make install* command [1]. In my experience, MPI could be successfully installed using only these three commands. Special configurations (where MPI is installed to a user's home directory) which do not require root access are also possibly by specifying a custom destination to the *make configure* command. These variations are outlined in the Installer's Guide [1].

One caveat is that the host must be able to resolve its hostname into its IP address for MPICH2 to work correctly. If the host does not do this on its own, it should be configured with an appropriate entry in the */etc/hosts* file. In my experience, problems starting MPI programs can sometimes be attributed to incorrectly configured hosts files. One final configuration detail is that the host must have a file called *~/.mpd.conf*, which contains a line *secretword=<secretword>*, where <secretword> is anything. For a cluster containing only one computer, the secret word is not important. However, the file is necessary to correctly start the mpd.

To run MPI programs, the message passing daemon (mpd) is first started by running the command *mpd*. Then, an MPI program is started using the command *mpiexec -n <N> <executable>*, where <N> is the number of processes to start, and <executable> is the filename of an executable file. The program will then be started in parallel.

The platform and operating system used for this configuration are as follows. The host is an AMD Athlon 64 4000+ (single core, 2.6 GHz) based computer, with 4GB of ram. The operating system used is Ubuntu 9.10 for the AMD64 platform. This configuration is also used as the host for the virtual cluster in the next section.

5 MPICH2 On A (Virtual) Cluster

To create a virtual cluster, one must first install software to host virtual machines. I chose VirtualBox, a free open-source product from Sun Microsystems. However, the details of setting up a virtual cluster are not specific to the virtualization tool used.

Four virtual machines were created. Ubuntu 9.10 Server Edition (AMD64) was installed in each virtual machine. Networking was configured so that each VM had a unique static IP address and hostname (cluster1-cluster4). As stated above, the configuration of the */etc/hosts* file is important. Each VM, as well as the host, must have an entry in the hosts for itself and for every other VM in the cluster. An error in one hosts file was responsible for some difficulties in running MPI programs on the virtual cluster. SSH certificates must also be set up on the VMs so that the host is able to SSH into them without requiring a password. This is used to start parallel programs.

For convenience, I set up an NFS server on the host computer, and configured each VM to mount it on boot. This allowed MPI programs to be compiled on the host and accessed directly by the virtual machines through the shared file system. One final configuration detail is that every VM as well as the host must have a file called *~/.mpd.conf*, which contains a line *secretword=<secretword>*, where <secretword> is something common to all nodes in the cluster. If the nodes do not all have identical *~/.mod.conf* files, *mpdboot* will not run successfully [1].

To transform the VMs from a group of networked computers into a cluster, the *mpdboot* command must be run on the host. The syntax is *mpdboot -n <N> -f <FILE>* where <N> is the number of mpds to start (usually equal to the number of computers in the cluster), and <FILE> is the location of a file containing a list of hostnames to include in the cluster, one per line. If this is successful, programs can be run using *mpiexec* as before [2].

For my experiments I included the host as well as the virtual machines in the cluster. However, it is also possible to not include the host in the cluster. In this case, one of the VMs in the cluster takes on the role of managing the cluster. It must have its SSH certificates placed on the other members of the cluster, and the *mpdboot* command must be executed from it. Whether the host or a virtual machine is used to run the *mpdboot* command, *mpiexec* will always run the process with rank 0 on this computer. Rank 0 is often used as a coordinator in a distributed algorithm, so fixing its location can be convenient.

6 MPICH2 On A Multiprocessor Host

Installing MPICH2 on a multiprocessor host is identical to installing it on a single processor host, as described in section 4. The computer used for these tests has 2 Intel Xeon CPUs. Each CPU has four cores, and each core supports two simultaneous threads. The clock speed used is 2.27GHz. Enough RAM was available in this configuration, as well as in the other two, so that no pages were swapped to disk during the tests.

7 Performance Evaluation

Since my interest in MPI is related to running parallel genetic algorithms, and the performance of an MPI installation is related to the exact mix of computation and communication used in a program, I decided to use my parallel genetic algorithm as a benchmark to compare the performance of the 3 configurations. One problem with using a genetic algorithm as a performance benchmark is that it is an inherently random process. However, I modified the program so that the random seed used by the pseudo random number generator was the same for each execution of the program (although it still varied between each rank). This caused the genetic algorithm to behave deterministically, and complete in the same number of generations (81) each time. Each execution corresponds to 81 generations in 16 populations, each of which has 256 individuals. The algorithm involves MPI communication in two places. The 16 populations are logically arranged in a 4x4 grid, and each population exchanges one individual with each of its neighbors each generation. Additionally, each population exchanges control and status messages with rank 0 at the end of each generation. I was able to run the same executable file on all 3 platforms without having to recompile it, eliminating the possibility of a compiler optimization affecting the results.

Three configurations were used to test the performance. The first was a multi-processor computer with 8 cores. The second was a single processor computer with one core. The third was a virtual cluster running on the single processor computer. Two other configurations are possible. The virtual machine could have been run on the multi-processor host to provide another data point. This was excluded for time reasons, as well as because the data from the single-processor host tests suggests that virtualizing a cluster has a large performance penalty. It also would have been interesting to compare this performance to a a real cluster. This was not done because a cluster was not available. Therefore this data is a comparison of MPI performance on a single computer only.

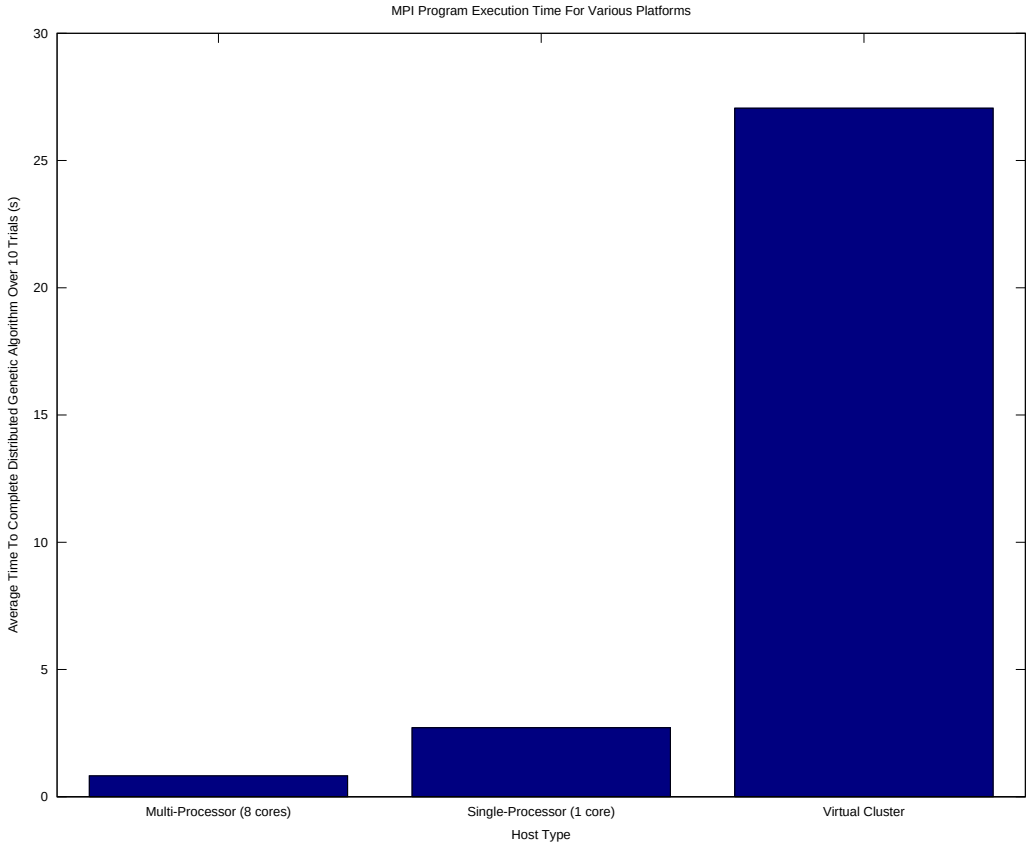

Figure 1: Mean execution time for 3 configurations over 10 trials each. Smaller times indicate better performance

Configuration	Mean (10 Trials) (s)	Standard Deviation (s)
Multi-Processor	.82	.074
Single-Processor	2.71	.017
Virtual Cluster	27.06	1.14

Table 1: Source data for Figure 1 with standard deviations

It is clear from Figure 1 and Table 1 that the choice of platform strongly affects MPI performance. As expected, the multi-processor computer was faster than the single-core computer, which was in turn faster than the virtual cluster. The genetic algorithm is inherently parallel, and was able to take advantage of multiple cores well, giving a three-fold performance increase from the single processor host to the multi-processor host. What is surprising is the degree of the performance penalty associated with virtualization, which increased the execution time ten-fold compared to the same program on the same host without virtualization. Clearly this is due to the increased overhead of the virtualization technology.

A virtual cluster can still be useful in a few situations. First, it allowed me to familiarize myself with the process of configuring a cluster without having to purchase any computer hardware. For an MPI course, a virtual cluster may be a very useful instructional tool. A virtual cluster could also be used to make a cluster more flexible. For example, in the event that a host fails or needs to be taken offline for maintenance, a virtual cluster would allow the virtual machine to migrate to another host, reducing downtime. However, the performance penalty is quite severe for virtualization in this case, so the increased flexibility may not be worth the decreased performance.

8 Conclusion

This was an interesting exploration of installing, configuring, and using MPICH2 in several different configurations. It allowed me to measure the performance penalty associated with using a virtual cluster as opposed to running multiple MPI processes on a single computer. It also demonstrated that MPI programs scale well to computers with different numbers of processor cores. It seems like the trend is toward increasing numbers of CPU cores, implying that parallelism will be required to appreciate the performance benefits that these cores can provide. MPI can provide interprocess communication and parallel execution tools on a single computer. Additionally, writing a program with MPI allows it to be run on a cluster for even greater performance.

References

[1] Argonne National Laboratory. *MPICH2 Installer's Guide*, November 2009.

[2] Argonne National Laboratory. *MPICH2 User's Guide*, November 2009. http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-1.2.1-userguide.pdf.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*, September 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.