Hybrid Programming Fun: Making Bzip2 Parallel with MPICH2 & pthreads on the Cray XD1

Charles Wright, *Alabama Supercomputer Center*

ABSTRACT: The author shares his programming experience and performance analysis in making a parallel file compression program for the Cray XD1. Discussion includes a practical example of combining MPI and pthreads in a single application. Issues with thread safety and MPI relative to the example program and the XD1 are covered. The author includes an analysis of MPICH2 using TCP/IP over RapidArray.

KEYWORDS: *XD1, MPI-2, MPICH2, pthreads, shared-memory, distributed memory, hybrid programming, RDMA, bzip*

1. Introduction

As an XD1 administrator and a student in a parallel programming class, the author came up with the idea of doing file compression in parallel for a programming project. He quickly learned that implementing a program with MPI alone could be very difficult, especially if coding for speed and efficiency. The MPI-2 standard has some advances such as one-sided communications that might make MPI more complete and easier to program. However, using MPI alone does not take full advantage of shared memory available on SMP cluster nodes and MPI-2 implementations are just now becoming available. A reasonable approach would be to combine pthreads and MPI on the XD1. Using this hybrid model, the author was able to parallelize non-computational tasks such as I/O and communication easily. There seemed to be no obvious solutions for achieving the level of parallelism needed using MPI alone.

A minor roadblock to this hybrid programming model is the fact that the Cray XD1 does not come with a thread-safe implementation of MPI. Installing MPICH2 on the XD1 overcomes this roadblock at the expense of both network performance and extra CPU utilization. Without RDMA the CPU limits the RapidArray Network to less than 20 percent of the throughput that would otherwise be attainable through an optimized MPI implementation. For this project, paying this heavy performance penalty turned out to be acceptable given increased efficiency and speedup of the hybrid model program. It remains unclear whether or not someone could achieve the same speedup and efficiency with MPICH alone that has been achieved with the hybrid model program. It is clear that using MPICH alone would certainly have been more difficult for this program.

This paper focuses on how pthreads were used to extend MPI in a natural way to improve the speed and efficiency of the program. Experimenting with more complete MPI-2 implementations is something to look forward to, but for the moment, combining pthreads and MPICH2 seems to be the best approach. A Rapid Array port of MPICH2 will hopefully be obtainable at some point in the near future so that choosing to use pthreads with MPI does not involve a major performance penalty.

2. Goals of the Project

Since the author's programming class was about parallel programming and not about file compression, he decided to use the bzip2 library instead of implementing his own compression algorithm. The bzip2 library is a lossless data compression library that achieves higher compression ratios than does the popular gzip program. However, bzip2's usefulness is often limited by the massive amounts of CPU time required to perform the compression. Bzip2 was measured on a 2.2 GHz Opteron CPU as able to process uncompressed data at a rate of only 4.8 GB/hour. Compressing a terabyte sized file or file system at this rate could take over eight days! The immediate goal of the project was to employ multiple processors to decrease bzip2 compression time as much as possible. (Note: Rate of compression varies depending on the data being compressed.)

Using the bzip2 library allowed the author to focus on parallel programming rather than file compression algorithms and provided compatibility with existing programs. For example, anything compressed by the program could be decompressed on any system that had a serial version of bunzip2. In fact, the author did not write a matching decompression program at all. In his environment, decompression is not a time sensitive task, as he planned to employ the parallel bzip program to perform file system backups.

To support the ability to perform a backup the program would need to take its input from STDIN. From the program's point of view input would be a possibly infinite length STDIN stream. When backing up a large file system, creating an intermediate tar archive and then compressing that archive with a parallel bzip program would not be an option due to the addition storage requirement. The input stream could be generated from the output of a tar command. This way, during an actual backup the only output of the program would be a compressed *tar.bz2* file. A recovery of the file system would be possible by executing a single
tar –jxf filename.tar.bz2 command.

After some research, it was found that a shared memory parallel version of bzip2 exists and can be found at http://compression.ca/pbzip2/. The author did not find an implementation using MPI. Using MPI was a good choice for a few reasons. Shared memory machines can support MPI but shared memory parallelized programs do not operate across distributed memory machines. It seems that large SMP machines are generally more expensive than MPI Clusters on a dollar per calculation basis and do not scale to the point that a MPI cluster can. At the Alabama Supercomputer Center this is reflected in the fact that the Altix cluster has less than half the number of CPUs that the XD1 has and the largest SMP node that we have is only 16 CPUs. An indicator of success for the MPI parallel bzip program in the author's environment would be the achievement of at least a 16x speedup.

3. Implementing a MPI-Only Version

The first step, in designing a parallel bzip2 program, was to write a serial bzip2 program making use of the bzip2 library. This provided important insight into how the library could be used in parallel. When compressing a file that is larger than a system's memory, a serial program has to work with only a buffer's worth of the file at a time. The serial program has to read enough of the file to fill a buffer, compress it, and then write it out before filling the buffer again with more of the file. To compress a file in parallel, the master process simply sends pieces of the file to different processors. As long as the compressed answers were written out in the correct order, a MPI parallelized version of bzip2 would be possible.

The first parallel version was written using only MPI. It was rather simple, in that the master process worked as presented in the following pseudo code.

```
while (!eof()) {
  Read from file
  Send 1 MB buffers to each slave to compress
  Compress Master's 1 MB piece
  Recv from all pieces (MPI_Gatherv)
  Write compressed buffers out
}
```

4. Problems with the MPI-Only Version

With the MPI-Only version the best performance achieved was an 8x speedup using twenty processors. This was not terrible considering this was the first attempt and it was mainly done as a proof of concept. The author was able to quickly look at the code and start finding ways to make it better. He assumed that it would take roughly about the same time to compress a buffer of the same size. In reality, this turned out to be a poor assumption, as the times measured for compressing a 1MB buffer ranged from 0.1 to 1.0 seconds. This fact alone could account for the inefficiency in the first version, as the master would always wait on the slowest slave before sending out the next set of buffers to compress. Slaves were idle during the time their buffers were in transit on the network and during the time it took the master to process their requests. The master process would not read in the next buffers to send until it was actually ready to send them. Therefore, the master process would block on file I/O at the worst possible time; when all slaves were idle. When planning some improvements to the first version there were no obvious solutions that involved using MPI alone.

5. Implementing a MPI + pthreads Version

Threading the master and slave MPI processes could help overcome the problems in the first MPI-only version. A mechanism to perform dynamic load balancing was added to account for the variable compression time of equally sized buffers. The goal of this mechanism was to give a slave the next buffer to compress immediately after it returned the one that was previously compressed. The master process was designed with the worst case scenario in mind. One slave might process ten buffers in the time it took another slave to process one buffer. An array of buffers was added to allow the master process to receive compressed buffers out of order from the slaves. One thread per MPI slave managed MPI communications with the slave processes. This setup allowed the use of blocking MPI_send and MPI_recv calls to stall only a thread while it waited on its slave process to receive, compress, and return a buffer. No complicated polling mechanisms would be needed to deal with non-blocking MPI routines.

Two threads were added to the master MPI process in order to achieve asynchronous file I/O. One thread's job would be to keep a set of input buffers full. The other I/O thread monitored the output buffer array and wrote out all the compressed buffers in the order that it could. This output thread would be woken up only when a communication thread received the buffer that it needed to write out. By using POSIX semaphores the master process coordinated the input thread with the slave communication threads in a producer/consumer fashion.

Support was added to allow a MPI slave to overlap communication and compression. A MPI slave would also need to be threaded to allow it to check out more than one buffer to compress simultaneously. The MPI slave used three threads, one to request, and receive buffers to compress, one to do the actual compression, and one to return the previously compressed buffer. This design allowed each MPI Slave to keep its CPU busy while buffers were in transit on the network.

6. Compiling and using a thread-safe MPICH2

Trying to combine the system MPICH with pthreads resulted in runtime errors such as the following:

```
pbzip:/tmp/igorodet/rpm/BUILD/mpich1.2.6/mpid/rai/dre
g.c:307: dreg_decr_refcount: Assertion `d->refcount > 0'
failed.  mpiexec: Error: read_full: EOF, only 0 of 4 bytes.
```

The solution to these run time errors would be installing MPICH2 configured with the flags --enable-threads and –with-thread-package=posix as well as using the MPI_Init_thread to initialize the MPI environment as opposed to MPI_Init. MPI_Init_thread specifies one of four methods of thread/communication operation. The one that the program needed was the worst case scenario for MPI, the MPI_THREAD_MULTIPLE method. This method tells MPI that multiple threads call MPI routines with no restrictions. The following code snippet was used to test if a particular MPI implementation could provide the MPI_THREAD_ MULTIPLE method.

```
// NOTICE THIS IS NOT MPI_Init!
MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIP
LE,&provided);
if (provided == MPI_THREAD_MULTIPLE )
    cout << "This version of MPI is thread safe" << endl;
```

Abandoning the RapidArray optimized MPICH on the system is not something one should do without some thought and planning. But if a program is not bandwidth limited and can be improved with the use of threads it is something that should be considered.

7. Requesting the right resources from the queue system

A goal of the project was to achieve a reasonable speedup efficiently. The program needed to request the right resources from the queue system in order to best achieve this goal. Since the master MPI process of the program was threaded, it was possible that it needed to use more than one processor. As one might expect, scalability is limited by the master process, so it is important to ensure that the master process be allocated more than one processor if needed. An appropriate request to the queue system would involve running the first MPI process exclusively on one XD1 node, giving its threads access to all the processors on that node. MPI Slave processes were threaded as well, but only one thread would do the CPU intensive compression. This thread could only use a maximum of one CPU. At run time, a slave's other threads might use a few extra cycles on an idle processor to do communication tasks, but the extra CPU utilization should be insignificant. The request to the queue system; therefore, only requested one processor per MPI slave.

8. Scalability Analysis

The majority of the scalability analysis focuses on the bandwidth requirements of the master process. Each slave worked with a 1MB sized buffer. The time a slave took to compress a 1MB buffer was measured. This time ranged from one tenth of a second to a full second. Of importance to this project, was the number of slave processes that the master process should be able to support in the worst case scenario that all buffers only took one tenth of a second to compress. The minimum number of slave processes could be calculated by dividing the bandwidth the master process is capable of pushing by the maximum bandwidth required per slave process. The author calculated the maximum bandwidth for a slave process to be 160 Mbits/Sec ((1MB*2xfers*8Mb/MB)/.1sec). This calculation assumes that no compression occurs, which is also a worst case scenario. An XD1 node without the expansion fabric is capable of about 10,000 Mbits/Sec worth of full-duplex bandwidth using the system MPICH. Using MPICH2 compiled generically on the XD1, a node is only capable of about 1800 Mbits/Sec. The performance difference is explained by the fact that MPICH2 compiled generically does not use RDMA; therefore, bandwidth becomes limited by the processor and memory. Using the system MPICH a master process would be capable of supporting at least 10000/160 or 62 worst case slaves. Using MPICH2 generically, the minimum number of worst case slaves supported would be 1800/160 or eleven slaves.

Supporting only eleven slaves did not meet the initial goal of a 16x speedup, but it did provide a valuable sanity check. If the program could not achieve at least an 11x speedup, then the problem was somewhere other than the network. To estimate actual performance, one could assume the average time to compress a 1 MB buffer to be ½ of a second and an average compression ratio of 2:1. The bandwidth requirement for a typical slave then becomes 24 Mbits/sec ((1.5MB*8Mb/MB)/.5sec). A realistic expectation for a master process would be for it to support 1800/24 or 75 slaves with MPICH2. Should a MPICH2 port to the XD1 become available, one could realistically expect up to 300 typical slave processes to be supported by a single master MPI process based on bandwidth requirements.

9. Profiling the Code

After implementing important features such as dynamic load balancing, asynchronous I/O and communication/compression overlapping, testing revealed that the program still was not able to achieve better than an 8x speedup. Efficiency had been greatly improved as the program achieved an 8x speedup with only 9 processors, but there still seemed to be a hidden barrier to achieving a speedup of greater than 8x. The next step was to determine where the unexpected bottlenecks were.

The TAU (Tuning and Analysis Utilities) package proved to be helpful. The general idea behind TAU would be to take my source code, run it through the TAU source code instrumentor, compile, execute and analyse the resulting profile data. For this program, nothing useful could be learned by simply using the automatic instrumentor. TAU became confused on various overlapping function calls called from inside program threads. The code would have to be instrumented by hand, which turned out to be not too difficult. TAU was then able to graph timing characteristics of all threads across all MPI processes.

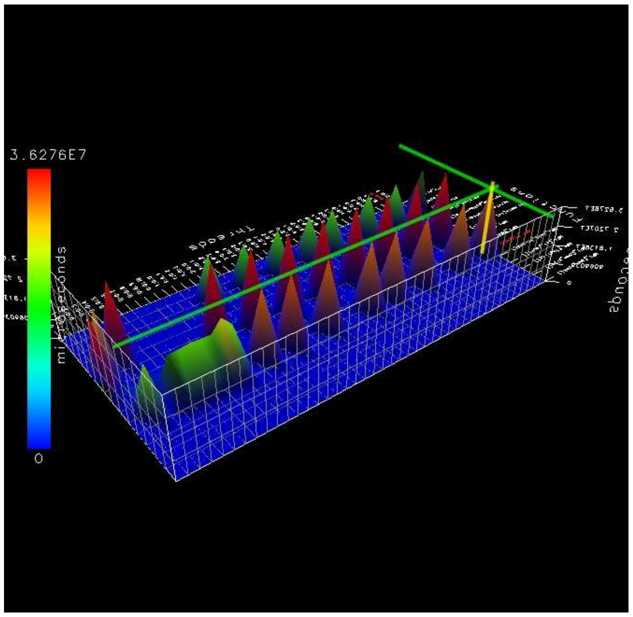

Figure 9.1 – A 3D Tau Graph with timing information for all threads across all MPI processes.

The three axes of the graph in figure 9.1 are time, threads, and functions. The green and yellow lines highlight a particular point on the graph, in this case, the time spent in a thread of the last MPI slave process in the MPI_Probe function. All the orange “trees” are slave processes. The green, tent shaped figure shows the time spent in MPI_Probe for each of the master threads that service communications. The yellow peak in this tent shows an anomaly uncovered by Tau. The last thread of the master MPI process gets preference over the others for some reason. This is most likely an undocumented OS scheduling “feature”.

These graphs indicate that the dynamic load balancing mechanism was doing a good job keeping the slaves MPI processes busy. The green “trees” in the graph represent the amount of time the compression was actually taking place. The red “trees” represent the total time the MPI slave process actually lived.

Figure 9.2 Important lesson learned from Tau

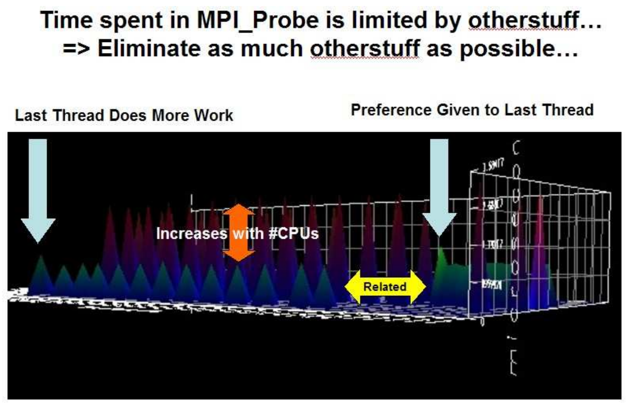

Figure 9.2 summarizes the information learned from using Tau. The time spent in communication by the master MPI process was being limited by something. This in turn, limited the amount of time slave processes spent compressing. With this information, the plan became to evaluate and eliminate all the unnecessary code from the master process.

10. Top to the rescue

Careful study of the master process resulted in the elimination of unnecessary source code. However, the improvements did not result in a speedup larger than 8x. The top command was used to watch the master process interactively. On the XD1 the system top command was not capable of displaying CPU utilization by individual thread. The system top only reported the combination of all the master MPI process's threads as using 160% CPU time. Determining which threads were responsible for utilizing the largest amount of CPU time would need to be known in order to understand what was happening. It was not difficult to download and install the procps package to get a version of top capable of displaying CPU usage by an individual thread. By telling this version of top to filter by user, show individual threads, and sort by Unix process id; it was determined which thread had the highest CPU utilization. It turns out the service_inbuffers thread was using 100% of the 160% reported by top. This was a big surprise in that the service_inbuffers thread was responsible for only reading from STDIN and filling up a set of buffers.

After some research on using C++'s cin function, the unexpected CPU utilization was traced to the cin function using a small buffer by default. The author attached a larger buffer to cin with the following two lines of code:

```
char mybuffer [bufferlength];
cin.rdbuf()->pubsetbuf(mybuffer,bufferlength);
```

A few test runs confirmed that the 8x speed up barrier had finally been broken!

11. Final Performance Results

The best test run for speedup and efficiency achieved a speedup of 19.78x using 20 processors. This seems incorrect as only 18 of the 20 processors were used to actually do compression. However, this is a correct measurement of an accidental superlinear speedup. The superlinear speedup results from the bzip library having to perform less data sorting than in the serial version. The compression ratios were very slightly different from the serial bzip2 program due to varying compression statistics being available to the bzip library. The compression ratios of a 4.4 Gig test file were measured to be 2.6179:1 for the serial version verses 2.6135:1 for the parallel version. Compatibility was not affected, as there was no problems observed decompressing any of the data the program compressed using the serial bunzip2 program.

Test runs using up to 30 processors were performed, but efficiency dropped to the 20 to 30 CPU range. Given the previous network analysis and goals of the project this was a good stopping point. Further testing may be performed after a MPICH2 RapidArray port becomes available.

12. Conclusions

The combination of pthreads and MPICH2 can result in many benefits ranging from easier programming to more effective use of system resources. Practical programming issues such as matching MPI_send and MPI_recv calls from multiple MPI slaves can be effectively addressed with the addition of threads. Profiling MPI and pthread code is possible with the TAU tool, when resorting to hand instrumentation. In the case of the parallel bzip program, the resulting improvements in both speedup and efficiency overshadow the lack of hardware support for MPICH2 currently available on the XD1.

13. Acknowledgments

The author would like to thank the Alabama Supercomputer Authority for the grant of CPU time, as well as Computer Sciences Corporation for the opportunities they provided.

Gratitude is extended to the authors of the bzip2 library as well as the authors of TAU. TAU is a joint project between the University of Oregon Performance Research

Lab, The LANL Advanced Computing Laboratory, and The Research Centre Julich at ZAM, Germany.

14. References

University of Oregon Performance Research Lab, The LANL Advanced Computing Laboratory, and The Research Centre Julich at ZAM, Germany(2005). TAU - Tuning and Analysis Utilities. Retrieved February 13, 2006, from http://www.cs.uoregon.edu/research/tau/home.php

jseward@bzip.org(2005). bzip2. Retrieved February 17, 2006 from http://www.bzip.org

MCS Division, Argonne National Laboratory, University of Chicago (2005).MPICH2. Retrieved February 19, 2006 from http://www-unix.mcs.anl.gov/mpi/mpich/index.htm

About the Author

Charles Wright is an employee of Computer Sciences Corporation, where he supports the Alabama Supercomputer Center as the lead System Administrator for a 60 processor SGI Altix 350 Cluster and a 144 processor Cray XD1. He holds a Bachelors Degree in Computer Engineering from Auburn University and is currently pursuing a Masters in Computer Engineering from the University of Alabama in Huntsville. He also holds a Cisco CCNP certification and has worked as a Network Administrator for the Alabama Research and Education Network.

His contact information is as follows:

Charles Wright
686 Discovery Dr
Huntsville Al 35806
256-971-7429
charles@asc.edu