Comparison of MPI Benchmark Programs on an SGI Altix ccNUMA Shared Memory Machine

Nor Asilah Wati Abdul Hamid, Paul Coddington and Francis Vaughan

School of Computer Science, University of Adelaide
Adelaide, SA 5005, Australia
{asilah,paulc,francis}@cs.adelaide.edu.au

Abstract

The results produced by five different MPI benchmark programs on an SGI Altix 3700 are analyzed and compared. There are significant differences in the results for some MPI operations. We investigate the reasons for these discrepancies, which are due to differences in the measurement techniques, implementation details and default configurations of the different benchmarks. The variation in results on the Altix are generally much greater than on a distributed memory machine, due primarily to the ccNUMA architecture and the importance of cache effects, as well as some implementation details of the SGI MPI libraries.

1. Introduction

Several benchmark programs have been developed to measure the performance of MPI on parallel computers. The programs were primarily designed for, and have mostly been used on, distributed memory machines. However it is interesting to measure MPI performance on shared memory machines such as the SGI Altix, which has become a popular system for high-performance computing. The hierarchical non-uniform memory architecture (NUMA) that is typical of large shared memory machines means that analysis of the performance of shared memory machines is likely to be more complex than distributed memory machines, which are typically clusters with a fairly uniform communications architecture.

In some recent work, we measured the MPI performance of the SGI Altix 3700 using MPIBench [6], a recently-developed MPI benchmark program. In order to check the results, and in particular to investigate anomalies in some results, we did similar measurements using other MPI benchmarks, which are reported here. We found that for some MPI routines, the results for different MPI benchmark programs were significantly different, and showed a greater variation than might have been expected, given that our experience with similar comparisons on distributed memory machines showed much smaller variation in the results from different benchmark programs. In this paper we compare the results of different MPI benchmarks on the SGI Altix, and investigate the reasons for these variations, which are due to differences in the measurement techniques, implementation details and default configurations of the different benchmarks. Many of these differences concern cache effects and communication patterns, which are much more important on shared memory machines with a cache-coherent NUMA architecture. However some of the effects appear to be artifacts of the implementation details of the SGI MPI libraries.

2. MPI Benchmark Software

There are several different MPI benchmark programs that are in common use. They typically measure the average times to complete a selection of MPI routines for different data sizes on a specified number of CPU using the following basic approach:

```
loop over different MPI routines
    loop over different message sizes
        get start time
        loop over number of repetitions
            if this is a collective communication routine, do
            a barrier synchronization call the MPI routine
        end loop over repetitions
        get finish time
            average time = (finish time - start time) /
                           number of repetitions
end loop over message sizes
end loop over MPI routines
```

Most benchmarks use the standard MPI timer MPI_Wtime, and get accurate results by making lots of repetitions of the measurements in order to compute the average value. Most benchmarks have a fixed number of message sizes (at least by default), but some also provide adaptive message length refinement in order to focus on message sizes where the communication time is changing rapidly. Some benchmarks also provide error control mechanisms to handle potentially large variations in communication times that may be caused by external influences, such as other programs that are using the communications network.

Most benchmark programs measure the time for collective communications on the root process. However, since the root process finishes first for many collective operations, this can bias the results due to "pipelining" effects where the root process finishes earlier and can start the next repetition of the operation before other processes have completed the previous operation. This is usually avoided by adding a barrier synchronization after each collective communication call.

An important point that can significantly affect the results is whether the message to be sent is in cache memory. Most benchmarks provide an option for specifying whether or not the data to be sent is in cache. The default setting for most benchmarks is that the data is in cache, and they do some preliminary repetitions of the MPI routine, which are not measured, in order to warm up the cache.

The following subsections summarize the important features of the main MPI benchmark programs, and highlight the areas where they differ from the basic approach described here.

2.1 Mpptest

The fundamental design philosophy of Mpptest [1, 11] is that the results of performance benchmarks should be reproducible. To reduce biases due to external influences, Mpptest spreads the test for each message length over the full time of the benchmark run, and measures the minimum average time over a number of repetitions in order to reduce variations. The structure of the measurement process for each MPI routine is:

```
loop over number of repetitions
    loop over different message sizes
        get start time
        loop over a small number of iterations
            call the MPI routine
        end loop over iterations
        get finish time
        average time = (finish time - start time) /
                            number of iterations
        if this is the fastest average time yet, accept it
    end loop over message sizes
end loop over repetitions
```

Mpptest provides a basic selection of MPI communication measurements: MPI_Send, MPI_Bcast and MPI_Scatter. It allows some options in the measurements for some of these operations, such as overlapping communication and computation. It provides adaptive message length refinement in order to focus on message sizes where the communication time is changing rapidly.

2.2 MPBench

MPBench [3, 12] follows the basic MPI benchmark approach, except that it uses the Unix timer *gettimeofday()*. MPBench measures the performance of MPI_Send, MPI_Bcast, MPI_Reduce, MPI_Allreduce, and MPI_Alltoall. It also measures bidirectional bandwidth using non-blocking sends (MPI_Isend) in both directions between two CPUs.

2.3 Pallas MPI Benchmark (PMB)

PMB [5] is a thorough, well-documented, user-friendly and commonly-used benchmark program. PMB measures a selection of common MPI routines: MPI_ Send, MPI_Sendrecv, MPI_Bcast, MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Reduce, MPI_ Reduce_Scatter, MPI_Allreduce and MPI_Barrier. It provides a number of different options for measuring these routines, including multiple communicator groups running the operations concurrently, and measuring bidirectional point-to-point communication. For MPI_ Bcast and MPI_Reduce, instead of using barrier synchronization to avoid biases due to pipelining effects, PMB changes the root node for each repetition.

2.4 SKaMPI

SKaMPI [4, 10] probably provides the most functionality of all the MPI benchmarks, with a large number of user definable parameters and MPI routines that can be measured. SKaMPI has divided the measurements into five categories: Point-to-Point, Master-Worker, Barrier Measured Collective, Synchronous Measured Collective and Simple pattern. The Point-to-Point includes all types of MPI Point-to-Point communication. The purpose of the Master Worker pattern is to test the network throughput and its handling of simultaneous communication, using MPI routines such as MPI_Waitsome, MPI_Waitany and MPI_Any_Source. The Barrier Measured Collective pattern is an older version that uses the standard approach of a barrier synchronization after each collective communication. The new version is the Synchronous Measured Collective pattern, which uses a globally synchronized clock to specify the time that each CPU should call the collective routine, and uses the time taken by the slowest process as the time for each repetition. This is expected to give more accurate results for collective communications, however it takes about twice as long to run [8]. Both the new and old versions can measure essentially all the MPI collective routines. Finally, the Simple pattern covers MPI routines that involve only one process and without any communication, such as MPI_Wtime and MPI_Comm_rank.

SKaMPI has more sophisticated error controls than the other MPI benchmarks. SKaMPI handles problems cause by external delays such as operating system interrupts by providing the option to ignore the 25% lowest and highest results to get the average. It also allows the user to specify a maximum statistical error (the default is 0.03%), and the measurements are repeated until the statistical error drops below this value, or the number of repetitions reaches a specified maximum value. SKaMPI also allows adaptive refinement of message sizes.

Unlike the other MPI benchmarks, by default SKaMPI ensures that the messages are not in cache. SKaMPI provides a detailed configuration file to change this default as well as many options, and to enable the user to choose which MPI routines to measure. Results are presented in a detailed report file.

2.5 MPIBench

MPIBench [2, 13] is the most recently developed MPI benchmark. The main feature of MPIBench is that it uses a very accurate, globally synchronized clock that is based on CPU cycle counters. This allows accurate measurement of individual MPI communications. MPIBench is therefore able to provide distributions (histograms) of communication times, rather than just average values, which can provide additional insight into communications performance. Rather than using a simple two processor ping-pong for point-to-point communications, MPIBench measures results for N processors communicating concurrently, and can therefore take into account effects of network contention. For collective communications, it can measure the different completion times for each process.

MPIBench measures the most common MPI communications: MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Sendrecv, MPI_Bcast, MPI_Barrier, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Alltoall and MPI_Reduce. In addition, MPIBench defines ***each*** and ***total*** keywords to identify message sizes, where each CPUs sends a fixed amount of message data or the total amount of message data is divided equally between all available processes, respectively.

Originally MPIBench assumed the message data was in cache, however a newer version has been developed that provides the option of using data that is not in cache. MPIBench can optionally handle outliers by discounting measurements that are larger than a specified factor above the average value.

3. Related Work

There has been surprisingly little work on comparing the results produced by different MPI benchmark programs. The papers describing the different MPI benchmark programs [1,2,3,4,5] typically provide a discussion of the differences in some of the measurement techniques used by the different benchmarks, but give little or no results comparing measurements from the different benchmark programs on different machines.

Mierendorff et al. [7] compare the results of PMB, SKaMPI, MPBench and Mpptest on an SGI Origin 2000, but only for point-to-point communication and only for 4 CPUs. However they provide useful insights into communication performance issues related to cache effects on ccNUMA architectures. Some related work that we have previously reported [6] was primarily concerned with the results from MPIBench measurements on the SGI Altix shared memory machine, and a comparison with MPIBench measurements taken on a distributed memory machine, an AlphaServer SC with Quadrics network. That paper included a brief comparison of MPIBench results with other MPI benchmarks for MPI_Send and MPI_Bcast operations, in order to illustrate the effects of some of the MPIBench measurement techniques. However the work presented here focuses on the comparison between results from different MPI benchmarks on the Altix, with a lot more benchmark comparison results, including results for more MPI operations, and a more detailed analysis of the differences in the results between different benchmarks.

4. Benchmark Experiments on an SGI Altix

The SGI Altix 3000 series has a cache coherent non-uniform memory architecture (ccNUMA) based on the hierarchical composition of two basic building blocks, or *bricks*: compute nodes (C-bricks) and routers (R-bricks). The C-bricks contain two compute nodes, each with two Itanium-2 CPUs connected to a custom network and memory controller ASIC, known as the SHUB. The two CPUs share a 6.4 GB/s bus to a SHUB. The two SHUBs in each C-brick are linked by a further 6.4 GB/s link. Each SHUB has one SGI NUMAlink channel to the outside, with a bandwidth of 3.2 GB/s (1.6 GB/s each direction) for NUMAlink3. These external links provide the cache coherent interconnect between C-Bricks. A set of routers (the R-Bricks) are used to expand the network in a scalable way. Each R-Brick provides 8 connections of 1.6 GB/s in each direction. The R-Bricks are configured so that four ports connect to C-Bricks, and the other four interconnect with other R-Bricks to form a fat tree network. Figure 1 shows a 128 CPU Altix.

The benchmark results reported in this paper were carried out on an SGI Altix 3700 managed by the South

Australian Partnership for Advanced Computing (SAPAC), with 160 1.3 GHz Itanium 2 CPUs, a total of 160 Gbytes of memory, and a NUMAlink3 network. At the time of the benchmarks, it was running SGI Linux ProPac3. The benchmark programs were compiled using Intel compilers and the SGI MPI libraries.

On shared memory machines, the operating system can switch processes between CPUs to try to improve overall system utilization. However this can adversely affect parallel programs, since after process migration, data will no longer be available in local cache. The performance of MPI programs on the Altix can be improved significantly by binding each process to a particular CPU. For our benchmark measurements we set the MPI_DSM_CPULIST environment variable, which assigns MPI processes in order to a specified list of CPUs. For each measurement we used a contiguous set of CPUs, starting with number 32, in order to maintain the hierarchical pattern of 32 CPU groups shown in Figure 1, while avoiding the use or CPU 0, which is used to run system processes, and so would affect the results.

By default, the SGI MPI implementation buffers messages, but uses single copy (i.e. no buffering) for large message sizes in most collective communication routines and in MPI_Sendrecv, which significantly improves performance [9]. The message size where the communication changes to single copy is not specified in the documentation but our measurements indicate it is around 2 Kbytes. By default, single copy is not used for MPI_Send, however it is possible to force the use of single copy by setting the environment variable MPI_BUFFER_MAX *n*, where *n* is the maximum message size where buffering will be used, so messages larger than *n* will use single copy. The choice of buffering or single copy can give a big difference in the performance of MPI_Send for large message sizes.

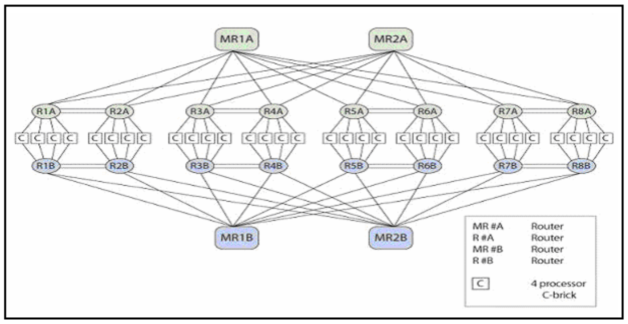

Figure 1: SGI Altix 3000 communications architecture for 128 CPUs.

5. Point-to-Point Communication

All MPI benchmark applications measure point-to-point communication using MPI_Send/ MPI_Recv. The main difference between the benchmark applications is the communication pattern. Figures 2-4 illustrate the communication patterns of the different benchmarks for 8 CPUs. Figure 2 shows the default point-to-point communications pattern for PMB and Mpptest, which involve CPUs 0 and 1 only. In order to measure communications times between CPUs that are not on the same node, the locations of the CPUs would have to be specified when calling mpirun. Mpptest also provides an option for specifying the distance between CPUs. PMB also provides an option for having multiple sets of CPUs communicating at once, however these are neighbouring pairs of CPUs (0-1, 2-3, etc) so this does not allow an accurate measure of contention in a hierarchical network.

Figure 3 is for SKaMPI and MPBench, which use the first and last CPUs. SKaMPI actually does short tests on all CPUs to find which has the slowest communication with CPU 0, and then does its timings using that CPU. However for the communication network on the Altix, this would be equivalent to choosing the last CPU, assuming processes are allocated to a contiguous set of CPUs, as was done in our experiments.

MPIBench measures not just the time for a ping-pong communication between two CPUs, but also measures the effects of contention when all CPUs simultaneously take part in point-to-point communication. The default communication pattern used by MPIBench is shown in Figure 4. MPIBench sets up pairs of communicating CPUs, with CPU p communicating with CPU (p + n/2) mod n when n CPUs are used. Half of the CPUs send while the other half receive, and then vice versa. The send/receive pairs are chosen to ensure that for a cluster of SMPs, or a hierarchical network (such as on the Altix), the performance of the full network hierarchy can be measured, not just local communications within an SMP node (or a brick on the Altix). MPIBench also allows the user to specify another communication pattern by specifying a list of communication partners.

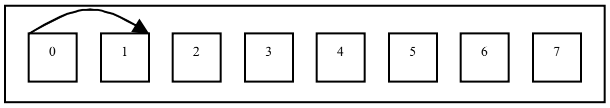

Figure 2: PMB and Mpptest Send/Recv.

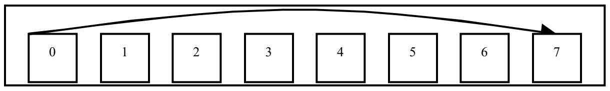

Figure 3: SKaMPI and Mpbench Send/Recv.

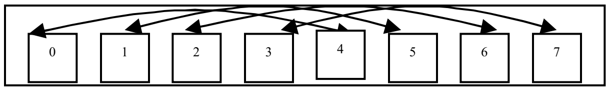

Figure 4: MPIBench Send/Recv.

5.1 Send/Receive

The difference in communication patterns between the different benchmarks leads to different results, as shown in Figure 5 for the default settings of the SGI MPI implementation for the Altix (i.e. buffered copy for MPI_Send). MPIBench has the highest results due to the contention effects from all 8 CPUs, while MPBench and SKaMPI obtain the second highest results since they are measuring the communication times between two C-Bricks. The lowest results are obtained by Mpptest and PMB, since they just measure intranode communication within a C-Brick. By carefully selecting the CPUs that are used (e.g. P0 and P7), it is possible to force each of the benchmarks to measure the same thing, i.e. point-to-point communication between two CPUs across any level of the communication hierarchy, and then the results for different benchmarks agree fairly closely, within a few percent. This is similar to benchmarking clusters of SMP nodes, where care must be taken (particularly for PMB and Mpptest) in choosing the CPUs to ensure measurement of internode rather than intranode communication.

On the SGI Altix it is possible to significantly improve the benchmark results for MPI_Send by enabling the option of single copy (i.e. non-buffered) sends in the SGI MPI implementation, as shown in Figures 5 and 6. As shown in Figure 6, it is best to set the MPI_BUFFER_MAX value to be very small, although there is little or no effect below about 256 bytes. Note that the improvement from using single copy can be large, up to a factor of 10, however it is much less for very large message sizes.

In measuring the results using single copy MPI_Send, we were surprised to find that while most of the benchmarks gave the expected improvement in performance, the results for SKaMPI and MPIBench were the same as for the default MPI setting that uses buffered copy. After much experimentation and comparison of the benchmark codes, we concluded that this problem is because both SKaMPI and MPIBench use the same array to hold send and receive message data. When we changed the MPIBench code to declare different arrays for send and receive data, the results showed the expected improvement, as shown in Figure 5. This appears to be an artefact of the SGI MPI implementation. We did not change the SKaMPI program, so we do not present SKaMPI results for the single copy option.

5.2 Combined Send and Receive

Only MPIBench, PMB and SKaMPI provide measurements for MPI_Sendrecv. For SKaMPI and PMB, each process sends to the right neighbour and receives from the left neighbour in a ring of N CPUs. MPIBench has a different approach, using the same communication pattern as it does for MPI_Send/ MPI_Recv (see Fig. 4), however each CPUs does a combined MPI_Sendrecv to its communication partner, rather than alternating sends and receives. Most communication networks are capable of providing the same bandwidth if messages are sent simultaneously in both directions on the same communications link. MPI_Sendrecv provides a good way of testing that the MPI implementation can indeed provide this bidirectional bandwidth. The MPIBench approach means that if this is the case, then the results for MPI_Sendrecv and MPI_Send/MPI_Recv should be similar. The results for MPI_Sendrecv in Figure 7 show that in general this is the case for the SGI Altix, e.g. the result for 256 Kbyte message size for 8 CPUs is similar to the result for 8 CPUs in MPI_Send/ MPI_Recv with Single Copy option (see Figure 5).

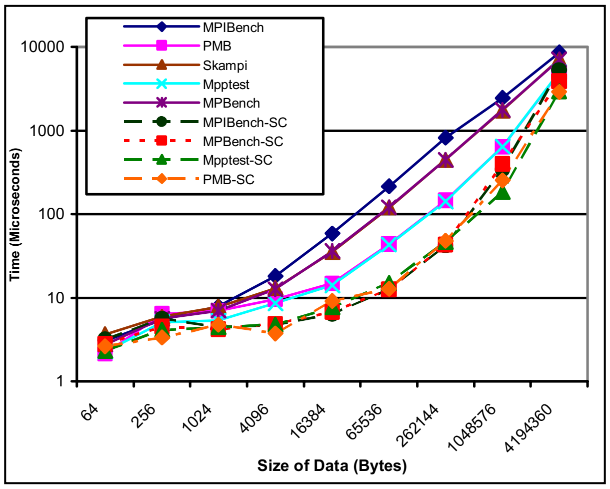

Figure 5: Point-to-Point (send/receive) using 8 CPUs for default settings and Single Copy (SC).

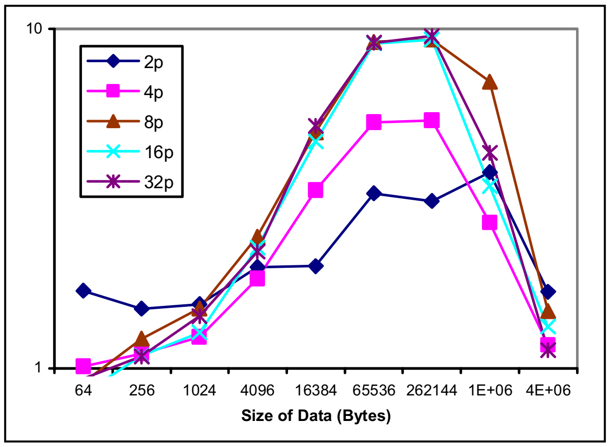

Figure 6: Ratio of MPI_Send times using buffered copy and single copy for PMB.

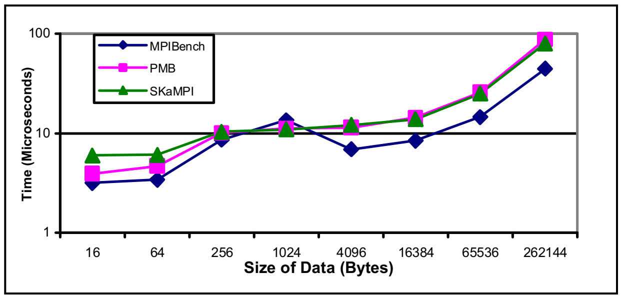

Figure 7: MPI_Sendrecv on 8 CPUs.

However MPIBench results show a hump around 1 Kbyte, since by default the SGI MPI implementation uses single copy for MPI_Sendrecv, but only for message sizes of 2 Kbytes or more. The approach used by SKaMPI and PMB cannot be used to test bidirectional bandwidth, however both these benchmarks provide other options for doing this type of measurement. Results for SKaMPI and PMB are a little higher than for MPIBench and also higher than the results for single copy MPI_Send. We are not sure why this is the case.

6. Barrier

Figure 8 shows the MPI_Barrier results for SKaMPI, MPIBench and PMB. The result for SKaMPI is a bit higher than MPIBench and PMB. This is probably due to the global clock synchronization that is set by default for their measurement. The developers of SKaMPI argue that this is a more accurate result since it avoids pipelining effects where some processes (e.g. the root) finish the barrier earlier and can start the next barrier operation before other processes have exited the barrier [8].

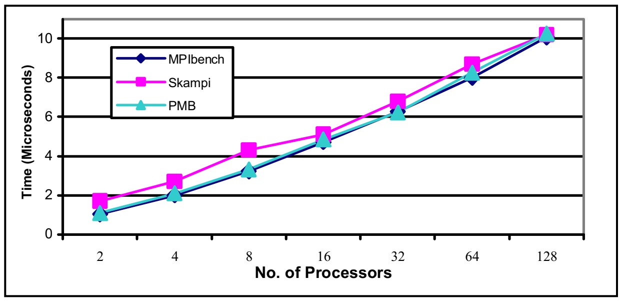

Figure 8: MPI_Barrier for 2 to 128 CPUs.

7. Broadcast

All benchmark applications measure MPI_Bcast. There are some differences in the measurement technique between the benchmark applications. The main difference is that by default SKaMPI makes the assumption that data should not held in cache memory, so it ensures data to be broadcast is not in cache before each measurement repetition. MPIBench, on the other hand, always sends the same data for each repetition, and does some preliminary “warm-up” repetitions (that are not measured) to ensure that the data is in cache before measurements are taken. The other benchmarks allow the user to choose whether or not data to be broadcast is in cache, although the default is that data is in cache memory. In a real application, data to be broadcast may or may not be in the cache, so there is really no “right” choice for whether or not an MPI benchmark should place the data in the cache.

Another difference is how the broadcasts are synchronized. Most MPI benchmarks measure collective communication time on the root node. However for some collective operations, such as broadcast, the root node is the first to finish, and this may lead to biased results due to pipelining effects. Most benchmarks get around this problem by inserting a barrier operation (MPI_Barrier) after each repetition of the collective communication operation. This provides an additional overhead which will affect the average time, although only for very small message sizes, since broadcast of a large message takes much longer than a barrier operation. Mpptest and PMB adopt a different approach to avoid this problem – they assign a different root node for each repetition.

Figure 9 shows the average times reported by the different MPI benchmarks to complete an MPI_Bcast operation. Clearly there are significant differences in the measured results due to the differences in measurement technique. Mpptest and PMB give the highest results, presumably due to the overhead of changing the root node at each iteration. We are not sure why Mpptest is so much higher than PMB. The only difference between the two approaches seems to be that PMB uses different arrays for the broadcast data on the root node and the other CPUs. SKaMPI has the next highest result, since it uses data that is not in the cache, while MPIBench and MPBench obtained the same results with the same measurement techniques.

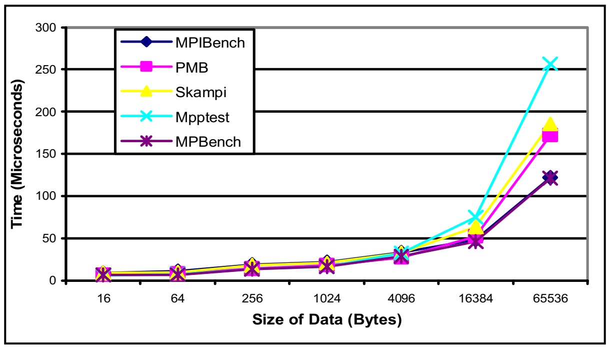

Figure 9: MPI_Bcast on 8 CPUs.

On a distributed memory cluster the effects of changing the root and having messages in cache has little affect on the results, however because of the cache coherency protocol on the shared memory Altix, moving the root to a different CPU has a significant overhead, which is reflected in the results. To check that these differences in the benchmark measurement techniques were causing the difference in broadcast times, we enabled the option to warm up the cache in SKaMPI, and for Mpptest and PMB we commented out the code to move the root process at each repetition, and then reran the benchmarks. The results after modifying the programs were very similar, with all results within a few percent.

By default, both SKaMPI and MPIBench use a barrier operation to synchronize the start of all collective communications. However they also have the option of avoiding the overhead of the barrier operation by using a synchronized start, where each CPUs starts each broadcast at a prescribed time, and the time reported for each repetition is the time taken by the slowest process. Clearly this requires a globally synchronized clock, which is provided by MPIBench and SKaMPI. Since they both use a globally synchronized clock, they are able to generate average times for each process in a collective communication, which can be significantly different. Figure 10 shows a figure from the SKaMPI report for an MPI_Bcast operation on 8 CPUs (using cache warmup to enable a direct comparison to MPIBench results), which shows the average completion time for each CPUs. The SKaMPI report also states that the average time for the MPI_Bcast is about 9500 µs, which is very different to the largest times for each node shown in Figure 10. We are not sure why this is the case. Figure 11 show the distribution results for MPI_Bcast on 8 CPUs for the same data size. This figure shows the combined results for all 8 CPUs, although recently MPIBench has been modified to allow distributions to be generated individually for each CPUs, so we are able to check that the overall distribution shown in Figure 11 shows peaks that are consistent with a binary tree broadcast algorithm, with the first peak corresponding to completion times for CPUs 0 and 1, the second peak is for CPUs 2 and 3 and final peak is for 4-7.

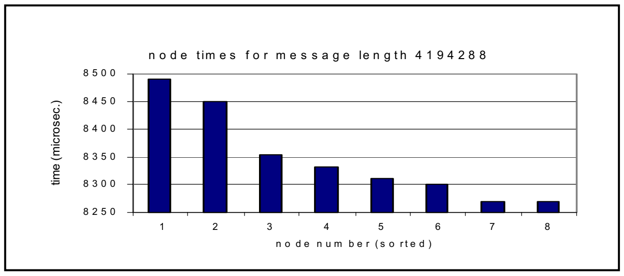

Figure 10: Node time produced by SKaMPI for MPI_Bcast at 4MBytes on 8 CPUs.

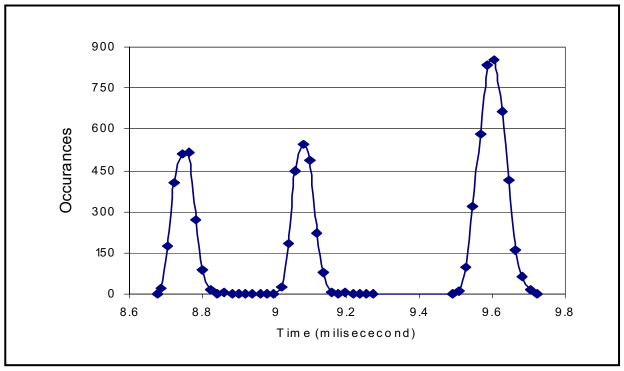

Figure 11: Distributions from MPIBench for MPI_Bcast at 4MBytes on 8 CPUs.

8. Other Collective Communications

Only MPIBench and SKaMPI provide measurements for MPI_Scatter and MPI_Gather, and both benchmark applications apply the same measurement technique. Scatter and gather are typically used to distribute data at the root process (e.g. a large array) evenly among the CPUs for parallel computation, and then recombine the data from each CPUs back into a single large data set on the root process. Figure 12 shows the comparison between MPIBench and SKaMPI for MPI_Scatter on 32 CPUs. The results show that MPIBench and SKaMPI report very similar times. The results also show an unexpected hump at a data sizes between 128 bytes and 2 KBytes per process, so that the time for scattering larger data sizes than this is actually lower. This is because by default SGI MPI uses buffered communications for message sizes less than 2 KBytes. Note that overall, the time for an MPI_Scatter operation grows remarkably slowly with data size.

The performance of MPI_Gather is mainly determined by how much data is received by the root process, which is the bottleneck in this operation. Hence the time taken is expected to be roughly proportional to the total data size for a fixed number of CPUs, with the time being slower for larger numbers of CPUs due to serialization and contention effects. Figure 13 shows comparison results between MPIBench and SKaMPI for 32 CPUs. As with MPI_Scatter, MPIBench and SKaMPI agree closely with each other.

The final collective communication operation that we measured is MPI_Alltoall, where each process sends its data to every other process. MPI_Alltoall is measured by MPIBench, PMB and SKaMPI. Figure 14 shows the results for 32 CPUs which are similar to MPI_Scatter, but with a sharper increase for larger data sizes probably indicating effects of contention. The results from the different benchmarks for different data sizes and numbers of CPUs mostly agree within about 10%.

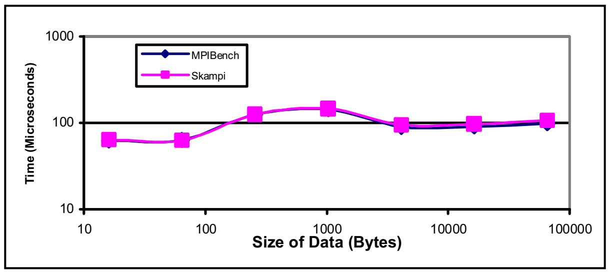

Figure 12 : MPI_Scatter on 32 CPUs.

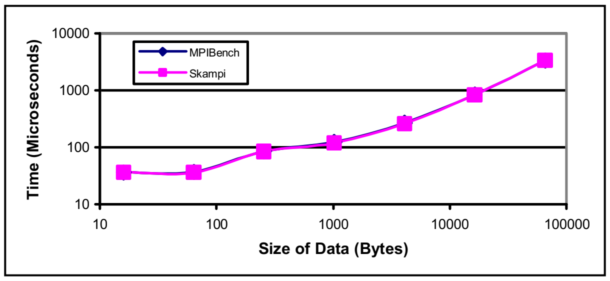

Figure 13: MPI_Gather on 32 CPUs.

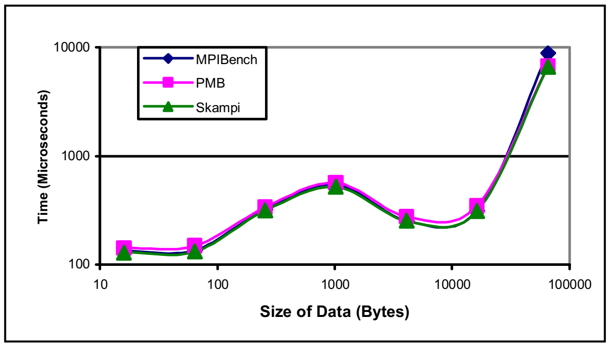

Figure 14: MPI_Alltoall on 32 CPUs.

Another collective communication that is measured by PMB, SKaMPI and MPBench is MPI_Reduce. MPI_Reduce does a reduction operation such as summation of data distributed over processes and brings the results to the root process. SKaMPI and MPBench use MPI_SUM as the parameter to MPI_Reduce, and therefore do a global sum. PMB uses a null operation and therefore only measures the communication involved in the reduction operation, and hence gives very different results to the other two benchmarks.

9. Summary

We have found that different MPI benchmarks can give significantly different results for certain MPI routines on the SGI Altix. This is primarily due to the Altix having a hierarchical ccNUMA architecture, which can enhance the variations due to different measurement techniques employed by the different benchmarks compared to a typical distributed memory architecture. For point-to-point communications, the variations are due to the different communications patterns used by the different benchmarks, and differences in how averages are computed, There are also significant effects due to implementation details of SGI MPI on the Altix, which affects whether single copy of buffered copy is used, which has a major impact on communications speed. There are also significant differences in measurements of some collective communications routines, particularly broadcast, due to differences in use of cache and in synchronizing the calls to the routines on each CPUs.

MPI benchmarks were designed primarily for use on distributed memory machines, and our results show that some of the design decisions can significantly affect the results for ccNUMA shared memory machines. Users of MPI benchmarks on such machines should therefore be careful in the interpreting the benchmark results, and developers of MPI benchmarks might consider making some minor modifications to their benchmark programs to provide more accurate results for ccNUMA machines.

References

[1] W. Gropp, E. Lusk. *Reproducible Measurements of MPI Performance Characteristics*. In Proc. of the PVM/MPI Users' Group Meeting (LNCS 1697), pages 11-18, 1999.

[2] D.A. Grove and P.D. Coddington. *Precise MPI Performance Measurement Using MPIBench,* in Proc. of HPC Asia, September 2001.

[3] P.J. Mucci, K. London, and J. Thurman. *The MPBench Report*. Technical Report UT-CS-98-394, University of Tenessee, Dept of Computer Science, November 1988.

[4] R. Reussner, P. Sanders, L. Prechelt, and M. Muller. *SKaMPI: A Detailed, Accurate MPI Benchmark.* Proc. of 5th European PVM/MPI Users' Group Meeting, 1998.

[5] Pallas MPI Benchmark (PMB) Homepage. http://www.pallas.de/pages/pmbd.htm

[6] N.A.W Abdul Hamid, P.D. Coddington and F. A. Vaughan *Performance Analysis of MPI Communications on the SGI Altix 3700*, Proc. Australian Partnership for Advanced Computing Conference (APAC'05), Gold Coast, Australia, September 2005.

[7] H. Mierendorff, K. Cassirer and H. Schwamborn. *Working with MPI Benchmarking Suites on ccNUMA Architectures,* Proc. of the 7th European PVM/MPI Users' Group Meeting, 2000.

[8] T. Worsch, R. Reussner and W. Augustin. *On Benchmarking Collective MPI Operations*, Proc. of 9th European PVM/MPI Users' Group Meeting, 2002.

[9] SGI Altix 3000. http://www.sgi.com/products/servers/altix/.

[10] SKaMPI. http://liinwww.ira.uka.de/~skampi/.

[11] Mpptest. http://www-unix.mcs.anl.gov/mpi/mpptest/.

[12] MPBench. http://icl.cs.utk.edu/projects/llcbench/mpbench.html

[13] MPIBench http://www.dhpc.adelaide.edu.au/projects/MPIBench