Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies

Analysis of Algorithm Selection for Optimizing Collective Communication with MPICH for Ethernet and Myrinet Networks

Nor Asilah Wati Abdul Hamid[1,2] and Paul Coddington[1]
1. School of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia
2. Department of Communication & Technology, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia.
asila@fsktm.upm.edu.my, paul.coddington@adelaide.edu.au

Abstract

MPICH is probably the most widely used implementation of MPI. Recent versions of MPICH optimize the performance of some collective communications by switching between different algorithms depending on whether the message size is greater or less than a given change-over point, which is currently hard-coded in MPICH, based on measurements on clusters with one CPU per node. We have used MPI benchmarks to find the optimum change-over points for different systems, and found that they can vary significantly for different networks and different numbers of processes per node. In some cases significant performance improvements can be obtained by enabling MPICH to be customized in this way, particularly on clusters with more than one CPU per node.

KEY WORDS
MPI benchmarks, parallel computer, network performance, collective communication.

1. Introduction

There has been much research into new algorithms aimed at improving the performance of collective communications for parallel programming using message passing. Recently, the developers of MPICH [2], probably the most commonly used implementation of the Message Passing Interface (MPI) standard, have released implementations of the best known collective communication algorithms that provide significantly improved performance [1]. The new algorithms have been applied in MPICH 1.2.6, the following versions, and also in MPICH2 [3].

For some of the MPI collective communication routines, the new MPICH implementation utilizes multiple algorithms, which are differentiated based on the size of the message and the number of processes. The message sizes mainly divide into two ranges, with short message sizes using algorithms that aim to minimize latency, while the long message sizes use algorithms that aim to minimize the bandwidth usage [1]. For example the implementation of broadcast in MPICH has changed from using the standard binomial tree algorithm in all cases, to using a combination of three algorithms: binomial tree for small message sizes or small numbers of processes; scatter followed by allgather for larger message sizes; and for allgather, either using recursive doubling or ring algorithm depending on the message size and whether the number of processes is a power of two.

Currently, the message sizes where the algorithm changes in MPICH are fixed to be the experimentally determined change-over points based on the work of Thakur et al. [1], which used a Linux cluster connected with Myrinet and an IBM SP, with one process per node. In their paper, they acknowledged that the optimum change-over points will probably be different for different parallel computers, and particularly for different networks, and planned to develop models to allow the change-over points to be set automatically based on system parameters. However, the MPICH 1.2.6 and MPICH2 1.0.4 source code shows that the message sizes where the algorithm is changed are still defined as constants and hard coded, so that the same values would be used for any MPICH installation on any parallel computer.

The aim of this study was to investigate the feasibility of using MPI benchmarks to provide an automated process for selecting the optimal choice of change-over points for different collective communication algorithms for a particular parallel computer and communication network, and to see if this approach is worthwhile by comparing the performance of the optimized MPICH implementation with the current MPICH implementation where the change-over points for algorithm selection are hard coded.

In this study, we have measured performance over a range of message sizes for all of the collective

0-7695-3049-4/07 $25.00 © 2007 IEEE
DOI 10.1109/.65

communications in MPICH that use different algorithms. Measurements were done on a cluster of dual processor machines using two different commonly-used networks, Myrinet with GM and Ethernet with TCP. In order to compare the different algorithms for all message sizes, the MPICH code was modified so that the changeover points were no longer constants, but variables that were initialized to the current static values in MPICH, which could then be overridden by reading from a configuration file or set using environments variables. For each collective communication routine, an MPI benchmark can be run to measure the performance for each possible algorithm, by varying the change-over parameters (e.g. by setting them to be the shortest or longest message sizes possible) to ensure that only a single algorithm is used for each benchmark run. Then the benchmark results for all the different algorithms for a particular collective communication routine can be compared and the optimal changeover points for that particular parallel computer can be determined.

Our results indicate that there can be big differences between the optimum change-over points on different platforms, and that in some cases significant performance improvements can be obtained by customizing MPICH in this way. In future, this approach could be developed further to create an automated process for setting the optimal change-over points for a particular parallel computer, which could be run as part of the MPICH install process.

2. Related Work

Thakur et al. [1] reported on improved implementations of collective communication algorithms in recent versions of MPICH and MPICH2. They compared the performance of different algorithms over a range of message sizes, for a Linux cluster with a Myrinet network and an IBM SP. In both cases the measurements were done using one process per node. The results from these measurements were used to fix the selection of algorithms for different message sizes in MPICH and MPICH2. Our work does similar measurements on a machine with more recent processors, and for more than one process per node, which is typical of modern parallel computers. Thakur et al. say that in future work they aim to develop models to allow the selection of changeover points between algorithms to be customized based on system parameters, whereas our work enables customization to be done based on benchmark results.

Recently there have been several efforts aimed at automatically tuning the performance of collective communications algorithms for the particular parallel computer being used [4,5]. These approaches are primarily aimed at tuning the implementation of each collective communication algorithm, for example selecting the optimum buffer size or communication topology for a particular machine

The most relevant of these studies to our work is that of Vadhiyar et al. [4]. They have developed automatically tuned collective communication by conducting several experiments on the system to obtain the optimum algorithm and optimum buffer size for a given collective communication using HARNESS FT-MPI, which is a fault tolerant MPI implementation. Basically, their approach followed three phases. In the first phase, the best buffer size for a given algorithm and a given number of processors is determined by evaluating the performance of the algorithm for different buffer sizes. For the second phase, the best algorithm for a given message size is chosen by repeating the first phase with a known set of algorithms and choosing the algorithm that gives the best results. In the third phase, the first and the second phase are repeated for different number of processors. Thus, the best algorithm will be chosen for the system and in certain cases there will be several algorithms for each collective communication which are differentiated by different message size, either small or large message sizes. Based on their results from a number of different parallel computers, the use of the tuned collective communication resulted in about 30% to 650% improvement in performance over MPICH or the vendor MPI implementation.

The main differences between the work presented here and the work of Vadhiyar et al. are that they use HARNESS FT-MPI, which in many cases does not use the best known algorithms that are used in MPICH, which is a much more widely used MPI implementation; their results were for a fairly small number of processors (usually 8 and in some cases 16); and their work (and related work on automatic tuning) is mainly focused on optimizing the performance of the implementation of each collective communication algorithm. Our work adopts a simpler approach to tuning performance of collective communication routines for a particular machine, by enabling MPICH to be configured to use the best existing algorithm implementation for small or large message sizes, based on the results of MPI benchmarks for a particular machine.

3. Methodology

In this study, MPI benchmark measurements were taken to compare the performance of the different algorithms for collective communication used in the latest versions of MPICH and MPICH2. Results were obtained for two commonly-used network intercon-

nects, Myrinet with GM and commodity Ethernet with TCP, measured on the same machine.

The measurements were done on an IBM eServer 1350 Linux cluster with 128 compute nodes connected by a Myrinet 2000 network as well as a 100 Mbit/s Fast Ethernet network. Each of the nodes are IBM xSeries 335 servers with dual 2.4 GHz Intel Xeon processors, so the machine has a total of 256 CPUs.

The Myrinet configuration has 8 nodes connected to each switch, and the switches connected together in a fat tree topology. The Ethernet configuration is that each rack of the cluster has a Fast Ethernet switch (100 Mbit/s full duplex) connecting all the nodes in the rack. Each of these switches has a Gigabit Ethernet (full duplex) uplink to a Cisco Gigabit switch. All measurements were done on nodes in a single rack. The cluster nodes were running Redhat Enterprise Linux version 3.2.3-47 with kernel 2.4.21-27.ELsmp of the Myrinet drivers.

This analysis used the latest MPICH, which is MPICH2 1.0.4 for Ethernet, however MPICH-GM 1.2.7 (based on MPICH1) is used for Myrinet since MPICH2 was not available for GM when the tests were done. However, the collective communication algorithms used are the same in each case.

The experiments involved benchmarking all of the collective communication routines in MPICH that use different algorithms, which are:

- MPI_Bcast – binomial tree, recursive doubling and ring with scatter algorithms;
- MPI_Alltoall - store-and-forward and pairwise exchange algorithms;
- MPI_Allgather - a variant of the distribution algorithm for barrier, recursive doubling and ring algorithms;
- MPI_Reducescatter - recursive halving, recursive doubling and pairwise exchange algorithms.

Thakur et al. [1] provide a detailed description of all of these algorithms.

In order to allow customization of the change-over points between different algorithms, and to enable us to benchmark the performance of all the different collective communications algorithms in MPICH for any message size, some changes have been made to the MPICH code. This just required converting the change-over points that are set as constant parameters in the MPICH code to be variables, and was straightforward to achieve.

For example, the implementation of MPI_Bcast in MPICH specifies a constant parameter MPIR_BCAST_SHORT_MSG. For message sizes less than this constant value, the binomial tree algorithm for broadcast is used, which provided the best performance for short message sizes in the measurements of Thakur et al. [1]. Our modified version of MPICH redefines this parameter to be an integer variable, and defines a new constant parameter MPIR_BCAST_SHORT_MSG_DEFAULT which is set to be the fixed value specified in MPICH. MPIR_BCAST_SHORT_MSG is initialized to this constant value, but then a new function is called to override this default value with a customized value if one is specified in a configuration file or an environment variable. These changes have been made to MPICH2 1.0.4 and MPICH-GM 1.2.7.

For each benchmark measurement, the changeover points were set so that measurements over a range of message sizes for each collective communication used only a single algorithm, so that results for each algorithm can be compared for all message sizes, in order to find the optimum changeover points for a particular parallel computer.

The MPI benchmark program used for the measurements was SKaMPI 4.1 [6] and all measurements were done using the default settings for SKaMPI. SKaMPI was chosen since it has a bigger variety of collective communication routines compared to other MPI benchmarks.

It was not possible to run the measurements with dedicated access to the cluster. In order to ensure the accuracy of the results, the measurements of different algorithms were taken one after another and using the same set of CPUs. At least three measurements were taken for each test, and as a check, at least one measurement for the same test was also taken using MPIBench [7] and Pallas MPI Benchmarks (PMB), in the cases where the MPI routine is provided by these MPI benchmarks. The measurements were done up to 16 nodes (32 CPUs) with 2 processes per node (ppn) and using 100 repetitions as a default setting for SKaMPI. For some routines, results were also obtained using 64 CPUs (32 nodes) for one and two processes per node, and also for numbers of processes which are not a power of two, in order to check for unusual results. In these cases only a single measurement run was done, and the results were compatible with the multiple runs done using 32 CPUs, so we present only the 32 CPU (16 node) results here.

In the following sections, the formulas for the approximate time expected for the different algorithms are taken from Thakur et al.[1], or (in some cases where that paper does not specify a formula) from comments in the MPICH2 1.0.4 source code [3]. The latency and bandwidth values for 32 CPUs used in the formulas are based on point-to-point communication for 32 CPUs using MPIBench [7], where the results should be more accurate than other MPI benchmarks since they consider the contention effects from communicating on all 32 CPUs. The measured latency for Myrinet is 18μs, while on Ethernet it is 102μs. The bandwidth for Myrinet is 180Byte/μs, while on

Ethernet it is 7Byte/μs. The analysis of Thakur et al. was done using one process per node, while the analysis here will also consider the performance for shared memory nodes since there are two CPUs per node used for the measurements.

4. Broadcast

MPICH2 1.0.4 and MPICH-GM 1.2.7 use some new broadcast algorithms. For small message size (<12KByte) or for less than 8 CPUs the standard binomial tree algorithm is used and the time taken for this algorithm is approximately $T_{tree} = [\lg p](\alpha+n\beta)$, p is the number of processors, α is the latency, β is the bandwidth and n is the message size. For medium (12KByte < medium < 512KByte) and for long (>512KByte) message sizes it uses scatter followed by allgather algorithm, which for medium message size and for power of two (POF2) number of processes uses recursive doubling algorithm and the time taken is $T_{recursive} = \lg p\ \alpha + (p-1)/p\ n\beta$, while for medium message size and for non power of two number of processes and also for long message size, a ring algorithm is used and the time taken is $T_{ring} = (\lg p + p-1)\ \alpha + 2(p-1)/p\ n\beta$.

For 8 processes (4 nodes with 2 ppn) on both Myrinet and Ethernet the performance of scatter and allgather becomes close to the binomial tree algorithm when the message size increases to around 16 Kbytes. However for both networks, the binary tree algorithm remains the best algorithm for all message sizes measured (up to 1 Mbyte), so on this machine the binary tree algorithm should be chosen when the number of processes is less than or equal to 8, rather than less than 8 as in standard MPICH. It is also interesting to note that for medium message sizes (between 12 and 512 Kbytes) where MPICH uses recursive doubling for the allgather, using the ring algorithm for allgather gives up to 50% better performance for Myrinet and even more for Ethernet.

The same effect can also be seen for larger numbers of processors. Figure 1 and Figure 2 show the performance of MPI_Bcast for different message sizes for 32 CPUs (16 nodes with 2 ppn) on Myrinet and Ethernet using different algorithms. Again, the time for the binomial tree algorithm starts to exceed that of the other algorithms at approximately 16Kbytes. For Ethernet, the binomial tree and the scatter with ring allgather perform better than scatter with recursive doubling allgather for all message sizes. For medium message sizes where recursive doubling is the default in MPICH, the improvement is approximately 40% to 50%.

For Myrinet, scatter with recursive doubling is (marginally) the best algorithm for message sizes in the range 8KByte to 32Kbyte. The change-over point to using the ring algorithm for allgather for large message is therefore much lower than the MPICH default of 512Kbytes, and the improvement in using the ring algorithm rather than recursive doubling between 32 and 512 Kbytes is approximately 30% to 40%. This result is a bit surprising, since the fixed change-over values in MPICH were taken from measurements on a Linux cluster with Myrinet. The only difference is that the experimental results of Thakur et al., and their theoretical models for estimating the communications time, are all based on one process per node, whereas our measurements were using two processes per node, since the cluster had dual processor nodes.

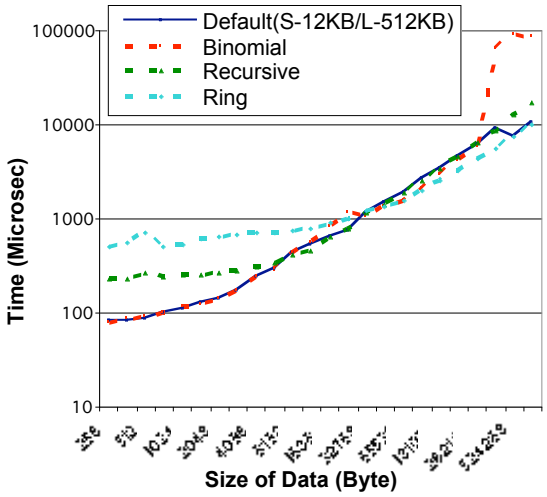

Figure 1: 32 CPU Broadcast for 2 processes per node (ppn) on Myrinet

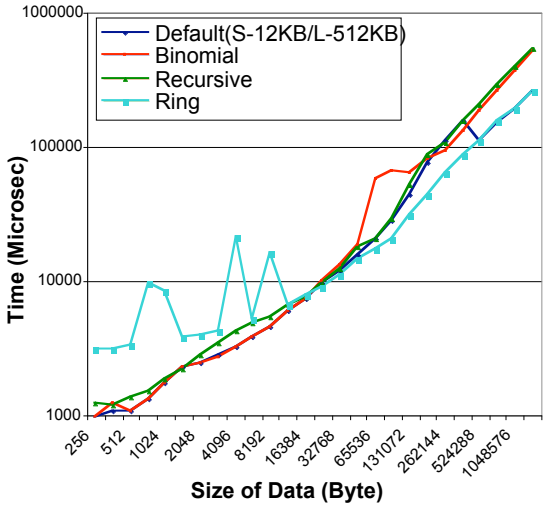

Figure 2: 32 CPU Broadcast for 2ppn on Ethernet.

Figure 3 shows our results from running the benchmarks using 1 process per node, and the change-over values now agree with the MPICH defaults. Figure 4 shows the predictions of the theoretical models for the communication times for each algorithm. They are a close fit to the measured results for 1 process per node, but a very poor match to the results for 2 processes per node. This indicates that in order to be able to specify customized change-over points based on system parameters, as suggested by Thakur et al. [1], we need new models that will provide good time estimates for multiple processes per node.

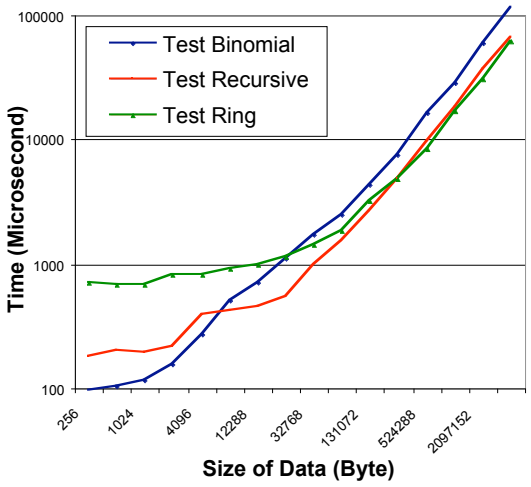

Figure 3: 32 CPU Broadcast for 1 ppn on Myrinet

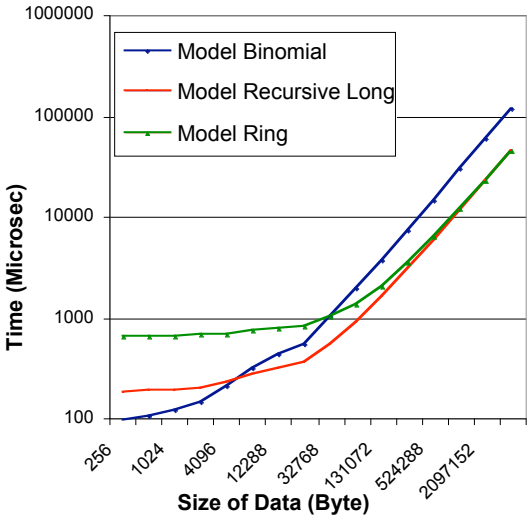

Figure 4: Model for 32 CPU Broadcast on Myrinet (α = 18μs; β = 180 Byte/sec).

5. Reduce Scatter

The algorithm for reduce scatter considers two type of reduction operations, commutative or non-commutative. If the operation is commutative, for short (<512 bytes) and medium-size messages (between 512 bytes and 512 Kbytes), it uses a recursive-halving algorithm in which the first p/2 processes send the second n/2 data to their counterparts in the other half and receive the first n/2 data from them. This procedure continues recursively, halving the data communicated at each step, for a total of lgp steps. So the time taken will be $T_{rec\_half} = lgp.\alpha + ((p-1)/p)n.\beta + ((p-1)/p)n.\gamma$. The time required for a typical arithmetic operation such as multiply or add is indicated by γ. If the number of processes is not a power-of-two, it will convert to the nearest lower power-of-two by having the first few even-numbered processes send their data to the neighboring odd-numbered process at (rank+1). Those odd-numbered processes compute the result for their left neighbor as well in the recursive halving algorithm, and then at the end send the result back to the processes that do not participate. The time taken for non-power-of-two is $T_{rec\_halv} = ([lgp] + 2)\alpha + 2n\beta + (1+(p-1)/p)n.\gamma$.

For commutative operations and very long messages (>512Kbytes) a pairwise exchange algorithm is used, similar to the one used in MPI_Alltoall. At step i, each process sends n/p amount of data to (rank+i) and receives n/p amount of data from rank-i) and the time taken is $T_{long} = (p-1)\alpha + ((p-1)/p)n.\beta + ((p-1)/p)n.\gamma$.

For non-commutative operations a recursive doubling algorithm is used for very short message sizes (<512 bytes), which takes lgp steps. At step 1, processes exchange (n-n/p) amount of data; at step 2, (n-2n/p) amount of data; at step 3, (n-4n/p) amount of data, and so forth. So the time taken will be $T_{short} = lgp.\alpha + n(lgp-(p-1)/p)\beta + n(lgp-(p-1)/p)\gamma$. The algorithm for medium (between 512 bytes and 512 Kbytes) and long (>512Kbytes) messages is the same as commutative for long message sizes, which is pairwise exchange, and the time taken will be the same as above.

The default setting of SKaMPI uses non-commutative operations, so the change-over point should have an effect at smaller message sizes. SKaMPI Reduce Scatter operation performs a tree-wise data reduction operation (Bitwise OR) on all participating processes and then distributes the result partially to all participating nodes, where every node receives a different part of the result-array.

In order to test commutative operations, measurements using PMB were performed, which uses MPI_FLOAT as the data type and MPI_SUM as the MPI operation. The results for PMB show that the optimum change-over point is decreased to 100KByte from the 512KByte default, however between 100 and 512 Kbytes the pairwise exchange algorithm is only around 5% to 10% better than the recursive-halving algorithm. So, for commutative operations there is little benefit in using a different change-over point.

For the non-commutative operation measured by SKaMPI, the results for Myrinet for 8 CPUs show that recursive doubling performs better than pairwise exchange up to 8 Kbytes, rather than the MPICH default of 512 bytes. As the number of CPUs is increased, the cross-over between the two algorithms increases also, as shown in Figure 5 for 32 CPUs, where the change-over point is 16 KByte. The improvement in performance from using the optimum change-over point increases as the number of CPUs increases. At 32 CPUs it is more than a factor of 4 between 512 bytes and 4 Kbytes, and more than a factor of 2 up to 8 Kbyes.

On Ethernet there is little difference between the two algorithms for 8 CPUs, but as with Myrinet, as the number of CPUs is increased the improvement by using recursive doubling is increased. For 8 CPUs the change-over occurs at 2KByte instead of 512 Byte, while at 32 CPUs the change-over point is at 8KByte. As shown in Figure 6, the performance improvement from moving the change-over value is significant, although not as large as for Myrinet.

Figure 7 shows the performance for 32 CPUs on Myrinet using one process per node (1ppn) rather than two, which again shows results consistent with the default change-over point in MPICH.

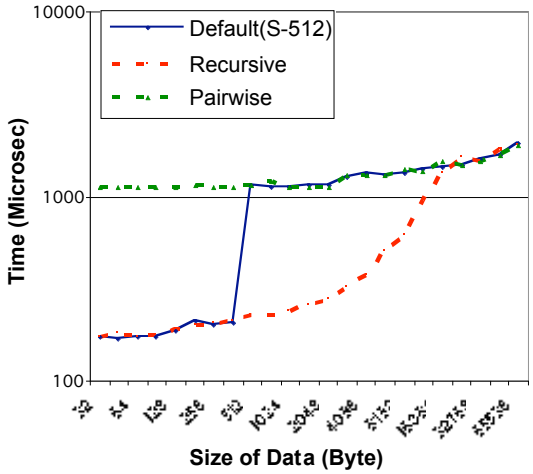

Figure 5: 32 CPUs Reduce Scatter for 2 ppn on Myrinet

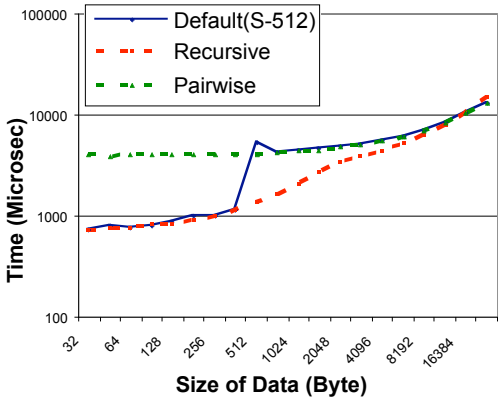

Figure 6: 32 CPUs Reduce Scatter for 2 ppn on Ethernet

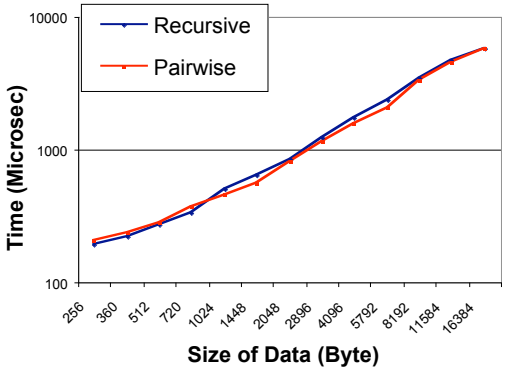

Figure 7 : 32 CPUs Reduce Scatter for 1 ppn on Myrinet.

6. All-to-All

MPICH uses four algorithms for MPI_Alltoall. For short messages (<=256Bytes) and (number of processors >= 8) it uses a store-and-forward algorithm. For medium size messages (256Bytes < medium message size =< 32KBytes) and (short messages for number of processes < 8) it uses an algorithm that posts all irecvs and isends and then does a waitall, then scatter the order of sources and destinations among the processes. For long messages and power-of-two number of processes, it uses a pairwise exchange algorithm. For long messages and a non-power-of-two number of processes, it uses an algorithm in which, in step i, each process receives from (rank-i) and sends to (rank+i).

Measurements using Myrinet and 2ppn show that the default settings are close to optimal, although moving the small message change-over point from 256Bytes to 512Bytes gives an improvement of 30% to 50% for messages in that range (Figure 8).

The improvement for Ethernet is much greater. In that case, the store-and-forward algorithm turns out to be slower than isend/irecv even for small messages, so using isend/irecv improves performance be around a factor of 2 for messages of size 256 bytes or less (Figure 9).

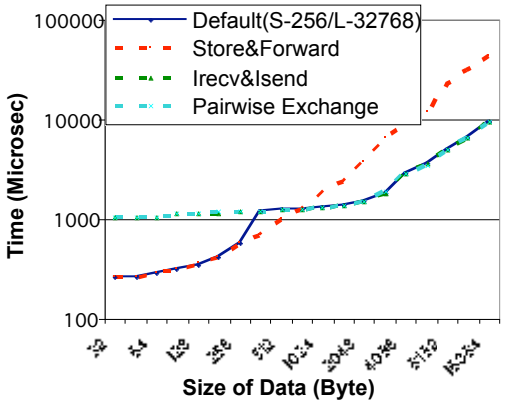

Figure 8 : 32 CPUs Alltoall for 2 ppn on Myrinet.

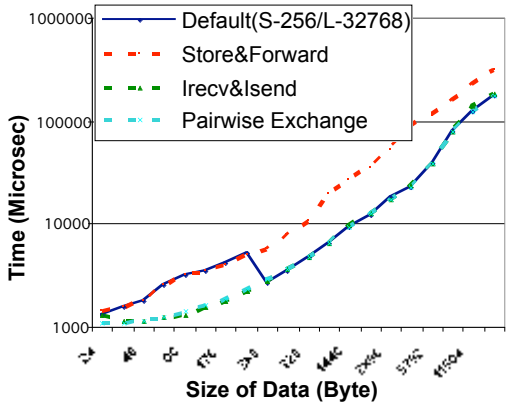

Figure 9 : 32 CPUs Alltoall for 2 ppn on Ethernet.

7. Allgather

In MPI_Allgather, for short messages and non-power-of-two number of processes, MPICH uses a variant of the distribution algorithm for barrier. For short or medium-size messages and power-of-two number of processes a recursive doubling algorithm is used. For long messages, or medium-size messages and non-power-of-two number of processes, a ring algorithm is used. In comparing with the change-over points in MPICH that define short, medium or long messages for Allgather, in this case the message size is multiplied by the number of processes, to give the total data transfer size.

Measurements on both Myrinet and Ethernet networks showed that significant performance improvements could be obtained by reducing the change-over point for large message sizes, and consequently using the ring algorithm rather than recursive doubling down to much smaller message sizes. This gave performance improvements for Myrinet of up to 70% on 8 CPUs and a factor of 2 for 32 CPUs (Figure 12), and a little more than that for Ethernet, with up to 2.5 times improvement for 32 CPUs (Figure 11).

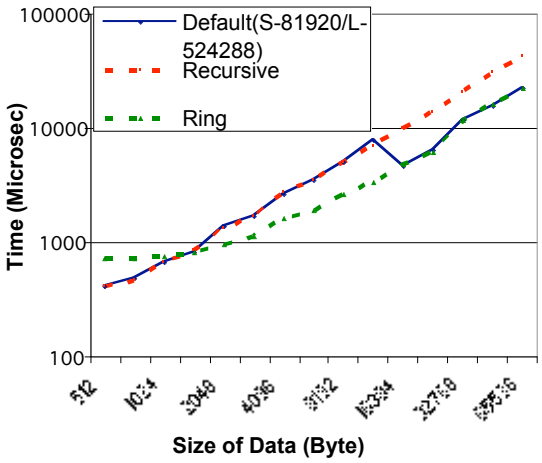

Figure 10 : 32 CPUs Allgather for 2 ppn on Myrinet.

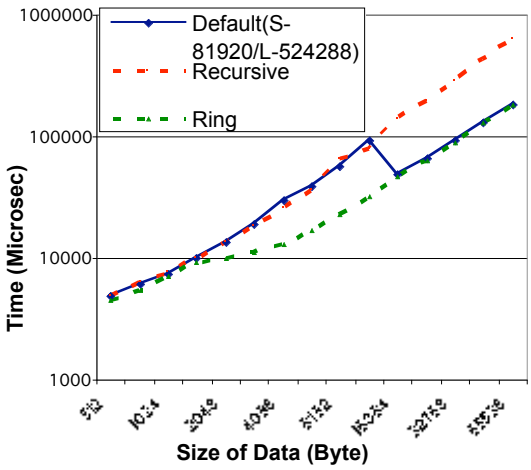

Figure 11: 32 CPUs Allgather for 2 ppn on Ethernet.

8. Conclusions

MPICH provides a mechanism for selecting between different algorithms for a particular collective communication routine based on whether the message size

is greater than or less than a specified change-over point. In current versions of MPICH this value is hard-coded, based on experimental results and theoretical models that assume a single process per node.

This study has demonstrated that it is straightforward to modify the MPICH code to allow the change-over points between different algorithms to be customized. This enables the change-over values to be set so that MPI benchmarks can be run to measure the performance of each different algorithm over any range of message sizes, and to therefore be able to find the optimal change-over points for any parallel computer. These customized change-over points can then be set in a configuration file and used by MPICH, in order to optimize the performance of collective communications for a particular parallel computer.

We have shown that the values of the optimal change-over points can vary significantly for different networks and different numbers of CPUs per node, and that using these customized change-over points can provide significant performance improvements for collective communications routines in the range of message sizes between the default MPICH change-over point and the optimum change-over point for the particular machine. All of the collective communications routines for which MPICH implements multiple algorithms showed improvements of over 50% for some message sizes, and in some cases improvements of a factor of 2 or more.

One of the main factors in determining the change-over point was the number of processes per node. With the advent of multi-core processors, all modern clusters will have more than one CPU per node, so it will be useful to be able customize collective communications rather than use the default change-over points that are based on measurements for a single process per node. We therefore recommend that this customization capability be added to MPICH and other MPI libraries.

9. Future Work

In future a fully automated mechanism for configuring the change-over points for each collective communication to maximize the performance will be developed. The idea is run an MPI benchmark for each algorithm used in each collective communication routine over a wide range of message sizes. Then the results will be processed to compute the optimal change-over points, which will be written to a configuration file for use by MPICH.

With the move to multi-core CPUs, new clusters are likely to have many cores per node. We plan to repeat these measurements on a new cluster with dual quad-core processors (i.e. 8 CPUs) per node, to see if there is an even greater variation from the default values in MPICH which are based on measurements with 1 CPU per node, and also to obtain results for Infiniband and Gigabit Ethernet networks.

Acknowledgements

This research was supported by computational resources supplied by the South Australian Partnership for Advanced Computing (SAPAC). Thanks to Paul Martinaitis for his work on modifying the MPICH code to support customization.

References

[1] Rajeev Thakur, Rolf Rabenseifner, and William Gropp, Optimization of Collective Communication Operations in MPICH, *Int. Journal of High Performance Computing Applications*, (19)1:49-66, 2005.

[2] MPICH - A portable implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich1/

[3] MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2

[4] S. Vadhiyar, G. E. Fagg and J. Dongarra. Automatically tuned collective communication. *In Proceedings of SC99: High Performance Networking and Computing*, November 1999.

[5] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proc. of Int. Conf. on Supercomputing ICS'05*, Cambridge, Mass., June 2005.

[6] R. Reussner, P. Sanders, L. Prechelt, and M. Muller. SKaMPI: A Detailed, Accurate MPI Benchmark. In Parallel Virtual Machine and Message Passing Interface, *Proc. of 5th European PVM/MPI Users' Group Meeting*, 1998.

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