Performance of Barriers and NAS Parallel Benchmarks on the Intel Clusters

Raj Krishnamurthy {rk@cc.gatech.edu}

1.0 Introduction

This performance report includes results from two separate sets of experiments. The first set of experiments results from running various barrier implementations on the Danish cluster. The description of each barrier is provided in [17]. The second set of experiments corresponds to running the NAS parallel benchmarks on the Intel clusters. We provide descriptions of each benchmark and also the corresponding performance results.

2.0 Barrier Performance on Switched Ethernet

It is important to understand the process architecture of MPICH. MPICH "forks" an executable into a "communications server" and "computation server". Each server must be scheduled for communication and computation and this introduces extra delays. Also, the MPICH device ch_p4 uses XDR for data representation and mapping. This introduces extra delays because, the data conversion from native format to XDR has to be done both on the sender-side and the receiver-side.

2.0.1 MPICH - Solaris x86

The barrier performance for the algorithms described in the section above were measured using an MPICH-Solaris x86 implementation. Table 1. shows the performance of the barriers on a 16 node Quad Pentium Pro cluster. We have implemented five basic bar-

rier structures, with multiple implementations on each structure to yield a total of ten barrier algorithms.

TABLE 1. Performance of Switched Ethernet Barrier Algorithms (1,000,000 iterations)

Algo.	Number of Cluster Machines (Num Procs) 1	3	5	7	9	12	14	16
Base-line(non-block send and receive) Cost / Msg	1.2 millisecond <---------- Cost per barrier call --------->							
System Barrier(MPI_Barrier) 100 iterations	0.04 ms	15.23 ms	30.31 ms	45.23 ms	65.32 ms	80.34 ms	110.23 ms	150.46 ms
Dissemination(MCS)	0.04 ms	7.12 ms	12.23 ms	23 ms	30 ms	42 ms	60 ms	71 ms
MCS tree	0.03 ms	5.56 ms	9.45 ms	15.56 ms	19.45 ms	25.65 ms	30.31 ms	36.23 ms
N-Way	0.04 ms	6.63 ms	10.33 ms	17.43 ms	22.32 ms	27.45 ms	33.43 ms	39.23 ms
Total Exchange	0.02 ms	9.13 ms	30.12 ms	53.21 ms	86.45 ms	156.34 ms	203.45 ms	271.23 ms
Tournament(MCS)	0.01 ms	7.12 ms	6.34 ms	9.23 ms	11.35 ms	14.45 ms	16.35 ms	19.34 ms

The measurements were recorded in each case for 1,000,000 iterations. Measurements were made using the nanosecond resolution gethrtime call and compared using the MPI_Wtime facility that gives microsecond resolution by reading the real-time clock on the network interface. First the baseline measurements were established. This is the time for a non-blocking send on one node and a blocking receive on other node. The one-way latency obtained is consistent with those obtained by the Ohio Supercomputing center[5].

The **Dissemination** barrier is the Hensgen, Finkel, Manber[HFM] dissemination barrier[1]. This involves exchanging messages for $\log_2$ P rounds as processors arrive at the barrier (P is the total number of nodes). A total of P messages is exchanged, in each round. The communication pattern is such that after the $\log_2$ P rounds are all over, all the processors are aware of barrier completion. A total of P*$\log_2$ P messages are exchanged. Now, based on the baseline measurements, for 16 nodes in the dissemination barrier, a total of 64 messages is likely to be exchanged. This means that the cost due to messages is 64*1

milliseconds . This is close to the measured result of 71 ms. The **MCS dissemination,** merely uses a parity variable that controls the use of alternating sets of flags in successive barrier episodes. This has the advantage that the barrier does not have to be initialized between successive barrier calls. The **MCS tree**, see section [2], uses a pair of P-node trees. There is an arrival tree and a wakeup tree. A 4-ary tree is basically used. Parent processors arrive at intermediate nodes of the arrival tree. The notification of completion is achieved by using the wakeup tree. P-1 messages are exchanged each during arrival and completion for a total of 2(P-1) messages. For this algorithm, for 16 processors, a total of 30 messages will be exchanged. This means that based on the baseline, 30*1 millisecs will be associated with communication costs. The result obtained by measuring the cost of synchronizing 16 nodes using this algorithm, 36.23 ms, is consistent with the baseline extrapolation. MCS Tree with BCAST simply, uses the MPI broadcast facility instead of maintaining a completion tree structure and conserves application space. The results here too are consistent with the baseline measurements.

The **N-WAY** barrier is an implementation where each process sends a message to a single designated process, and waits for a receipt message. The designated process accepts all (P - 1) messages from the other processes, before sending P - 1 receipt messages [16]. Again for sixteen processors, a total of 30 messages will be communicated. The results obtained are consistent with baseline extrapolation. N-Way and MCS tree naturally track each other in terms of performance, but MCS Tree distributes the work better and will scale much better. N-Way uses a single focal point process and is not expected to scale. The central process may get swamped with messages.

Total Exchange is an $O(P^2)$ communication operation. A node sends P -1 messages and expects P - 1 messages. Here again, for four nodes, a total of 225 messages are exchanged and based on baseline extrapolation, the total communication cost is expected to be around 225 milliseconds. This is consistent with the measured result. The BCAST and SendRecv pair optimizations, use broadcast and send/receives for communication. Here too, the results are consistent.

HFM Tournament is the Hansgen, Finkel and Manber tournament algorithm [1]. This is another tree-style algorithm, in which the winner in each round is determined statically. In the tournament algorithm, processors are only at the leaves of the arrival tree. The algorithm, is explained in Section 2 [1]. In round k (counting from zero) of the HFM barrier, processor i makes a non-blocking send and j makes a non-blocking receive, where $i = 2^k \pmod{2^{k+1}}$ and $j=i-2^k$. Processor i then drops out of the tournament and blocks on a broadcast (notice of completion of the algorithm). Processor j participates in the next round of the tournament. A complete tournament consists of $[\log_2 P]$ rounds. Processor 0 broadcasts the completion of the barrier. A total of P messages are exchanged, making this the most efficient barrier. For sixteen nodes, only 16 messages are exchanged and the total communication cost of 19.34 milliseconds is consistent with the baseline measurements. The MCS modification to this includes the use of the parity flag that obviates the need for re-initialization during each barrier call. It performs slightly better than the HFM tournament because of this. This is extremely consistent with the baseline measurement extrapolation.

The system barrier breaks with 1,000, 000 iterations. With 100 iterations, the system barrier MPI_Barrier performs poorly with respect to the HFM- MCS tournament algorithm.

The HFM-MCS tournament algorithm is the clear winner. The graph shows five of the ten barrier algorithms. It shows the five barrier algorithms with the best-performing representative optimizations. The performance of the barrier algorithms is shown in Figure 1. MCS tournament is not much affected by processor scaling. Dissemination-MCS involves twice the message exchange (with respect to the tournament barrier) and has a barrier performance lower than that of the MCS tournament algorithm (nearly twice that of the tournament, with twice the number of messages exchanged).

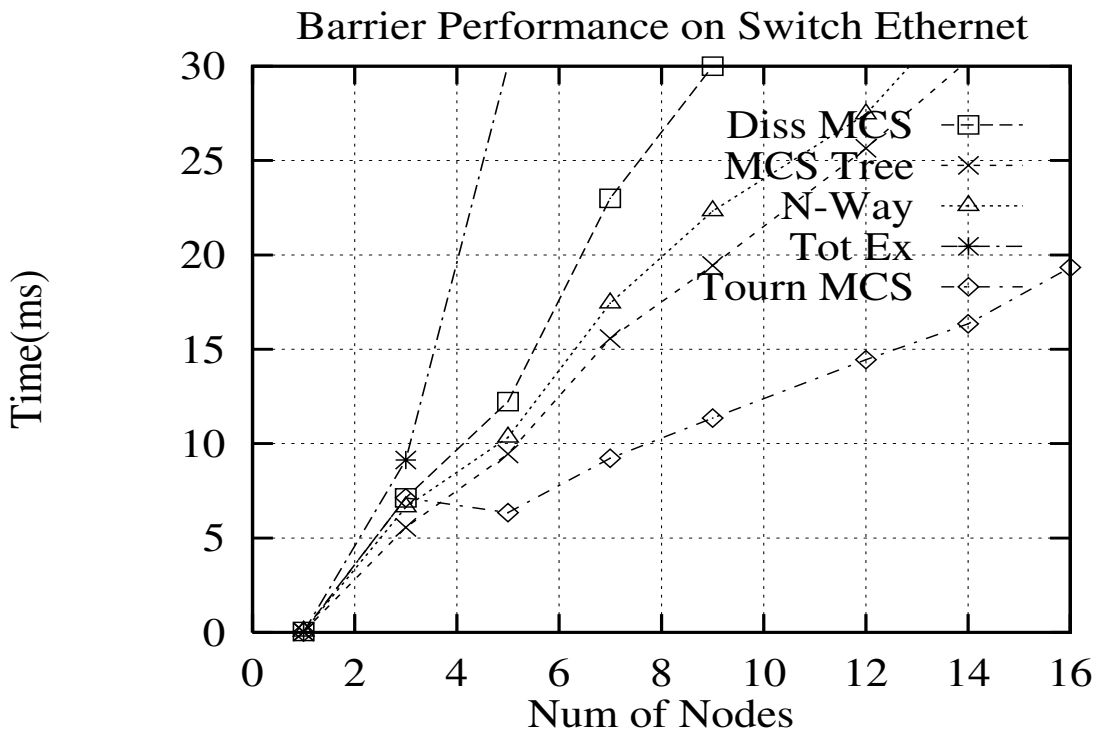

Figure 1 . Comparing Barrier Performance on Switched Ethernet

3.0 NAS Parallel Benchmarks

The NAS benchmarks were developed for the performance evaluation of highly parallel supercomputers. They consist of five parallel kernels and three simulated application benchmarks, which mimic the computation and data movement characteristics of large-scale computational fluid dynamics (CFD) applications. These benchmarks provide an approximation of the performance a typical user can expect to obtain for a portable parallel program on a computer with distributed memory. [10]

EP is the "embarrassingly parallel" kernel. It provides an estimate of the attainable upper bounds for floating-point performance--that is, the performance without significant interprocessor communication. This code implements a random-number generator for the calculation of integrals, and no communication is required to generate these numbers. Communication is required only to combine sums from various processors at the end of the computation, so EP is a good fit for parallel processing. The problem here is typical

of applications used for Monte Carlo simulation. For this kernel, there is no special requirement on the number of processors that must be used. [10]

CG is a kernel that estimates the smallest eigenvalue for a large, sparse, symmetric positive definite matrix with a random pattern of nonzeros. It accomplishes this through use of the conjugate gradient (inverse power) method, from which the kernel gets its name. Using unstructured matrix vector multiplication to test irregular long-distance communication, the CG kernel is representative of code for computations performed on unstructured grids. The kernel can be used with any number of processors that is a power of two [10].

The **IS** kernel implements a large integer sort of a number of keys in parallel. The sorting operation performed here has importance in "particle-in-cell method" codes. What it tests is both speed of integer computation and communication performance. A sequential bucket-sort algorithm is implemented in this kernel. First, every key is read, and the count of its corresponding bucket is incremented. The various bucket counts are then transformed through a prefix-sum operation. Finally, the keys are ranked according to the prefix sums. The initial distribution of the keys can strongly effect the performance of this kernel. Like many of the other benchmarks, IS requires a number of processors that is a power of two.[10]

The 3-D FFT PDE (**FT**) benchmark is implemented according to a mostly standard scheme. The 3-D array of data is distributed according to the array's z-planes, of which one or more are stored in every processor. The forward 3-D FFT is then accomplished by using multiple 1-D FFT's in each dimension, the x- and y-dimensions coming into play first; all this can be done in just one processor, without any communication between processors. Next there is an array transposition--equivalent to an all-to-all exchange--such that every processor relays pieces of its data to each of the other processors. Then the last set of 1-D FFTs is conducted. The 1-D complex FFTs are handled according to a standard Stockham-transpose-Stockham method. For inverse 3-D FFTs, the same

scheme is used but in reverse. This kernel requires a number of processors that is a power of two. [10]

The **LU** application benchmark has its origin in the NX reference implementation from 1991. It too requires a number of processors that is a power of two. The grid is partitioned onto processors in 2-D by halving the grid alternately in the x- and y-dimensions until every one of the processors has been assigned. This results in vertical, pencil-like grid partitions on the various processors. The ordering of point-based operations which make up the SSOR procedure moves along diagonals which range from one corner of a given z-plane to the opposite corner, thus advancing to the next z-plane. The data from partition boundaries is communicated after computation has completed on each diagonal in contact with a neighboring partition. This defines a diagonal pipelining method that is referred to as "wavefront" by its originators.[8] The result is a fairly sizable number of small communications--each one consists of five words. This application is useful in that it is highly sensitive to the communication performance of small messages in an MPI implementation. [10]

The **MG** (Multigrid) is also derived from the NX reference implementation from 1991. There are four key subroutines here used for: the computation of residuals, the projection of residuals, smoothing, and the trilinear interpolation of the correction. Each of these was optimized for both RISC and vector processors. This is another of the kernels requiring a number of processors that is a power of two. The grid is partitioned onto processors as follows: the grid is first halved in the z-dimension, then in the y-dimension, and finally in the x-dimension; this is repeated until all processors (which together number some power of two) have been allocated. [10]

The **SP** (pentadiagonal solver) and **BT** (block tridiagonal solver) application benchmarks have similar structures. Each one solves three sets of uncoupled systems of equations, first in the x-, next in the y-, and then in the z-direction. In the SP code, the systems are scalar pentadiagonal; in the BT, they are block tridiagonal with 5x5 blocks. These two benchmarks solve the three systems through a multi-partition approach [9]. This method

is effective since it gives good load balance and features communication that is coarse-grained. In the algorithm, every processor is assigned to a handful of disjoint sub-blocks of points on the grid. The cells are arrayed so that for every direction of the line-solve phase, cells belonging to a particular processor will be distributed evenly along the direction of the solution; this way each processor can perform some useful work through a line solve. Furthermore, the information in a cell is sent to the next process only when all the sections of linear-equation systems handled in the cell have been resolved. Thus, communication granularity stays large and a smaller number of messages are sent. Both the SP and BT benchmarks require square numbers of processors, but they can handle other numbers as well. [10]

We have selected the following benchmarks for analysis and development. The applications and kernels are summarized in table 2.

TABLE 2. NAS Parallel Benchamarks used in Study and Work Performed with each

Benchmark	Barrier Used	Language of Code / Programming Model	Kernel / Application	Work Performed	Targets
CG	Tournament developed in C	C / SPMD/ Message Passing	Kernel Problem size: 14000 Class W	Development from scratch in C for message passing	Switched Ethernet
EP	Tournament developed in C	C / SPMD/ Message Passing	Kernel Problem Size: 67108864 Class W	Development from scratch in C for message passing	Switched Ethernet

TABLE 2. NAS Parallel Benchamarks used in Study and Work Performed with each

Benchmark	Barrier Used	Language of Code / Programming Model	Kernel / Application	Work Performed	Targets
LU	Tournament developed in C	FORTRAN / SPMD / Message passing	Application Problem Size Used : 33 X 33 X 33 Class W	Modify original benchmark in Fortran to use message passing and allow barrier developed in C to integrate	Switched Ethernet
MG	Tournament developed in C	FORTRAN / SPMD / Message Passing	Kernel Problem Size Used : 64 X 64 X 64 Class W	Modify original benchmark in Fortran to use message passing and allow barrier developed in C to integrate	Switched Ethernet, Myrinet
BT	Tournament developed in C	FORTRAN / SPMD / Message Passing	Application Problem Size Used : 24 X 24 X 24 Class W	Modify original benchmark in Fortran to use message passing and allow barrier developed in C to integrate	Switched Ethernet, Myrinet

We have run the benchmarks on a Switched Ethernet cluster and also the Myrinet cluster. The speedup results obtained for the benchmarks on the Switched Ethernet cluster are shown in Figure 12(A). EP and BT show linear speedup. For LU, MG and CG, the curves show linear speedup till 8 processors. After that, the speedup drops as the applications / kernels, do extra work communicating with the same small problem size.

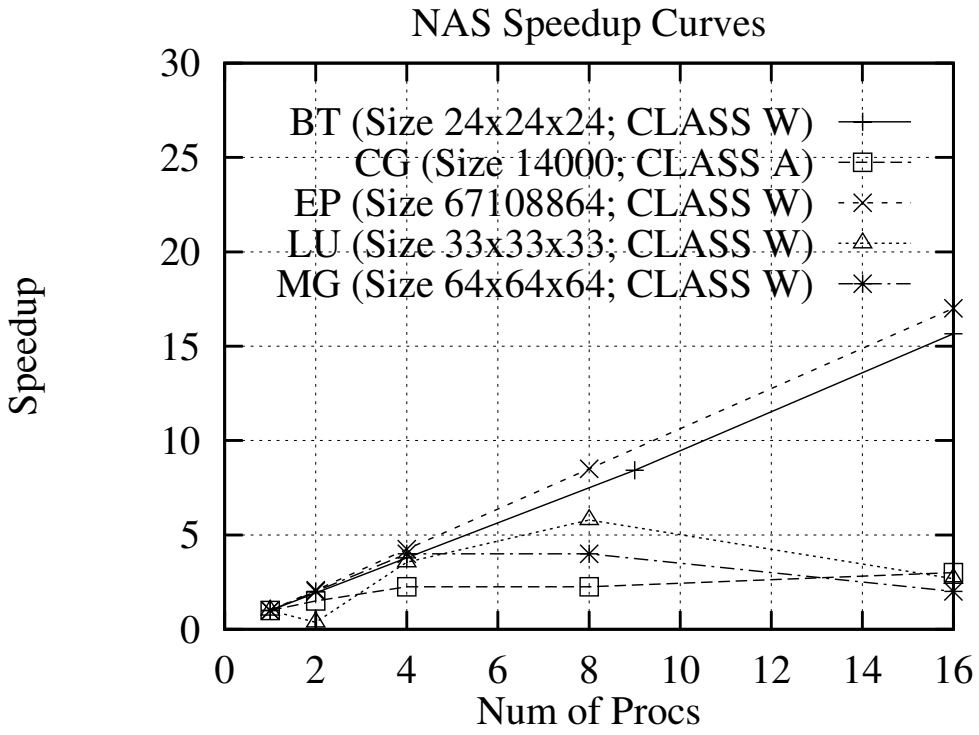

Figure 12(A) Speedup for Switched Ethernet

For Switched Ethernet , MG deserves special mention. MG requires a number of barrier synchronization calls as each grid is computed. The barrier synchronization is implemented using the tournament algorithm developed. The linear speedup in both cases with MG attests to the improved barrier synchronization performed by the tournament barrier algorithm.

References

[1] D. Hensgen, R. Finkel, and U. Manber, Two algorithms for barrier synchronization, *International Journal of Parallel Programming* 17(1) (1988) 1-17.

[2] John M. Mellor-Crummey and Michael L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Transactions on Computer Systems* 9(1) (1991) 21-65.

[3] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy, Scalability study of the KSR-1, *Parallel Computing* 22 (1996) 739-759.

[4] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su, Myrinet--a gigabit-per-second local-area network, *IEEE Micro*, 15(1):29-36, February 1995.

[5] Message Passing Interface Forum, The MPI message passing interface standard, Technical report, University of Tennessee, Knoxville, April 1994.

[6] S. Pakin, M. Lauria, and A. Chien, High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet, *Supercomputing*, December 1995.

[7] M. Lauria, A. Chien, MPI-FM: High performance MPI on workstation clusters, Technical Report, University of Illinois, Urbana-Champaign, December 1995.

[8] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga, Solution of regular, sparse triangular linear systems on vector and distributed-memory multiprocessors, Technical Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA, 94035-1000, April 1993.

[9] J. Bruno and P.R. Cappello, Implementing the beam and warming method on the hypercube, *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan 19-20, 1988.

[10] `http://www.nas.nasa.gov/NAS/NPB`

[11] J. Mellor-Crummey and M. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Transactions on Computer Systems*, January 1991.

[12] D. Johnson, D. Lilja, J. Riedl, and J. Anderson, Low-cost, high-performance barrier synchronization on networks of workstations, *Journal of Parallel and Distributed Computing* 40 (1997) 131-137.

[13] E. Styer, R. Finkel, U. Manber, Designing efficient barriers in communication networks, Technical Report, University of Kentucky.

[14] S. Cheung, V. Sunderam, Performance of barrier synchronization methods in a multiaccess network, *IEEE Transactions on Parallel and Distributed Systems* 6(8) (1995) 890-895.

[15] N. Arenstorf and H. Jordan, Comparing barrier algorithms, *Parallel Computing* 12 (1989) 157-170.

[16] J. Hill, and D. Skillicorn, Practical barrier synchronisation, Technical Report, Oxford University Computing Laboratory, 1996.

[17] R. Krishnamurthy et al, *Implementation and Performance Evaluation of Barrier Synchronization Algorithms on a Network of Workstations and Study of NAS Parallel Benchmark Performance on Workstation Clusters,* Class Report - Parallel Computer Architecture, Spring 1998.