Algorithms for Collective Communication Operations on SMP Clusters (EXTENDED ABSTRACT)

Maciej Gołębiewski Rolf Hempel Jesper Larsson Träff
C & C Research Laboratories, NEC Europe Ltd.
Sankt Augustin, Germany
{maciej,hempel,traff}@ccrl-nece.technopark.gmd.de

Abstract

We argue the need for algorithms for collective communication operations that can exploit the hierarchical structure of contemporary parallel systems like clusters of symmetric multiprocessors (SMPs). We support the argument by results achieved on a 16 node, dual-processor PC cluster with very simple improvements to commonly used implementations of collective communication operations.

1 Introduction

Recent years have seen a proliferation of clusters of workstations or PCs as a cost-effective alternative to dedicated massively parallel processors for computation intensive applications. Such clusters are often structured as a collection of Symmetric Multi Processor (SMP) nodes, interconnected by a high-speed communication network. Each SMP node consists of a small number of processors, typically 2 to 8, connected to a shared memory via a common bus or crossbar. In practice intra-node communication via shared memory is considerably faster and has smaller latency than inter-node communication even with a high-speed network. To complicate matters further, intra-node performance is highly sensitive to memory network contention, and cache effects can be considerable. Thus, such systems are not well represented by conventional uniform parallel models, be they either shared or distributed memory models. Consequently, algorithms designed using a uniform model are not likely to perform optimally on such machines. This abstract is intended to raise discussion of some issues coming up in the design of communication algorithms for SMP clusters, and generally for hierarchical multiprocessors. Modeling such systems will not be addressed explicitly; one useful generic model capturing both memory hierarchy and interprocessor communication is proposed in [1, 2].

From the application programmer's point of view, a shared-nothing, process-oriented programming model, as provided by the widely used Message Passing Interface (MPI) [10, 15], will often be the right one for programming also hybrid, more tightly coupled systems as described above. MPI defines a set of collective communication operations for tasks like barrier synchronization, broadcast, reduce, all-to-all exchange etc., where all processes within a communication domain must participate. In the most common MPI implementation, MPICH [11], the collective operations have straightforward, non-optimal implementations, all based on the assumption that communication between any pair of processes has the same cost. These implementations are not efficient for machines where the uniformity assumption does not hold.

In MPI only processes belonging to the same communication domain described by a *communicator* can communicate. Within a communicator each process has a unique rank which identifies it to the other processes in the communicator. Collective communication operations can be performed among processes belonging to the same communicator, and involve all processes of the communicator. MPI provides mechanisms for dynamically changing communicators. Within this programming model the following two observations are central for the design of better algorithms for hierarchical multiprocessors:

- Processes on the same node communicate more efficiently than processes on different nodes. Furthermore, there may be bandwidth problems when several processes on a node simultaneously communicate with processes on other nodes. Algorithms should exploit the fast intra-node communication, and as far as possible avoid the problems of the second sort.
- When a collective operation is executed it cannot be assumed that processes are evenly distributed over the nodes: it is not known how the scheduler allocates resources, and the MPI capability to define communication domains dynamically makes arbitrary process (rank) to processor allocations possible. Good algorithms must be adaptive to this environment. So far this seems to be a neglected issue.

A methodology (SIMPLE) for designing algorithms for SMP clusters, not strictly supporting the MPI model, is proposed in [3].

In the following we substantiate the points raised above. We first describe the PC cluster we set up to gain experience in cluster-based parallel computing. We then describe simple algorithmic improvements for broadcast, all-to-all exchange and all-gather which give noteworthy performance gains over the MPICH implementations of these operations.

2 The LAMP PC cluster

At the NEC C&C Research Lab an SMP cluster has been set up in 1997, based on previous work at NEC Research Institute (NECI) on the Local Area MultiProcessor (LAMP) [8, 13]. With its symmetric multiprocessor (SMP) architecture, the LAMP can be regarded as a small-scale counterpart of NEC's parallel-vector supercomputer SX-4, and is, therefore, well-suited for experiments with clustered shared memory machines.

The cluster is a 16 node SMP PC cluster interconnected by a Myrinet network [12]. Dual 8-Port Myrinet-SAN switches, each serving 4 PCs, provide a tetrahedral network topology. Each node has two Pentium Pro 200MHz CPUs installed on a Tyan S1668 motherboard with Intel's 440FX chip-set and 512 KB of L2 cache per CPU. The PCs have 256 MB of RAM each, so that the cluster has a total of 32 CPUs and 4 GB of RAM. The nodes are equipped with two network interface cards: an Ethernet card and a LANai board for accessing the Myrinet network. The LANai cards are basic 32 bit PCI models equipped with only 256 KB of SRAM. The cluster runs under Linux (Red Hat 5.1 distribution) with kernel version 2.0.34 (SMP). The choice of Unix gives full support for services such as remote debugging and administration, and multiple users on each node. The cluster is complemented by a Pentium II PC (300 MHz, 64 MB RAM) with SCSI disk drives, which is used as a file and compilation server.

The MPI implementation used on LAMP is a dedicated port of MPICH 1.1.0 to the Illinois Fast Messages (FM) 2.1 library [6]. The current version uses a native ADI2 device driver with 2 communication protocols (short and rendezvous) optimized for FM. This provides better performance than a previous design based on the generic channel device. The library is a dual-device implementation: the FM device is complemented by MPICH's lock-free shared memory device (`ch_lfshmem`) to achieve high intra-node communication performance. This device has been slightly modified to provide full MPMD support. Both devices operate in a kind of master-slave hierarchy. This is due to the FM device being able to communicate with any process of an application, while `ch_lfshmem` works only within a single node. Also, the proper initialization of `ch_lfshmem` relies on communication facilities provided by the FM device. For more details about design choices for the low-level communication, see [4, 9].

Latency and bandwidth achieved with a ping-pong test program (using `MPI_Send()` and `MPI_Recv()`), for both intra- and inter-node communication are shown in Figures 1 and 2.

Both with respect to latency and bandwidth there is a significant difference between intra- and inter-node communication, intra-node latency being a factor of at least 2 smaller than inter-node latency. The bandwidth for inter-node communication via Myrinet shows a peak at 32 KB and then drops. For 64 KB the performance fluctuates between measurements and is in many cases better than for 32 KB. The reason for this behavior is that the Myrinet communication performance is highly dependent on the LANai control program executed by the NICs. If temporary network conditions cause this program to execute another control path that is only slightly longer than the usual one, performance changes significantly. The fluctuation occurs already at 32 KB, but is less marked than for 64 KB. In case of longer messages, e.g. 128 KB, the dominant factor is cache interaction described below; even if one can still notice the fluctuations, they are very rare compared to 64 KB case.

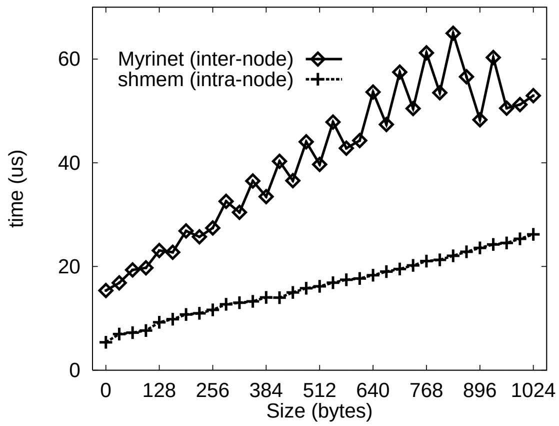

Figure 1: Intra- and inter-node latency measured as time (in micro seconds) to send a short message.

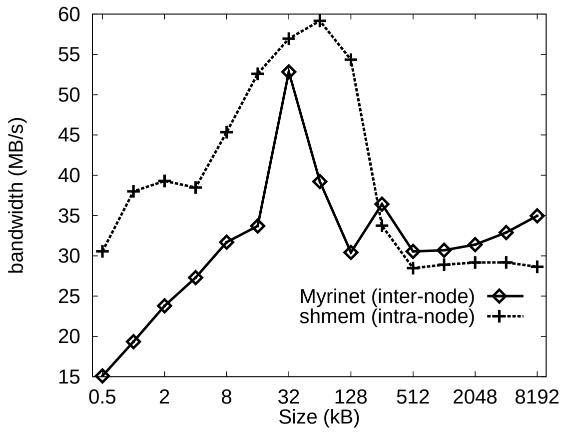

Figure 2: Intra- and inter-node bandwidth (in Megabytes/second).

The dramatic drop in bandwidth around 128 KB for both intra- and inter-node communication is due to cache effects. This is caused by the fact that L2 cache in the Pentium Pro is relatively small, only 512 KB. Since the `ch_lfshmem` device uses `memcpy()` for data transfer and FM uses DMA only on the receive side (and PIO in the sender), data has always to go through the cache. While for shorter messages 512 KB of L2 cache seems to be sufficient, in case of longer ones the number of cache misses rises and performance deteriorates.

3 Collective communication algorithms

The standard MPICH implementations of the `MPI_Barrier()`, `MPI_Bcast()`, `MPI_Alltoall()`, and `MPI_Allgather()` operations have been improved and tailored to the LAMP system. Already the simple improvements described below lead to significant performance gains; however, the algorithms are based on regularity assumptions that are fulfilled for the LAMP system with only 2 processors per SMP node, but which will not hold for systems with more processors in the SMP nodes.

Let P be the number of MPI processes in the current communicator, n the number of SMP nodes, and p the (maximum) number of processors per node. For the LAMP system, $n = P/2$ and $p = 2$. We make no assumption about the mapping of process ranks to processors and nodes. Better collective communication algorithms are designed according to the following two objectives:

1. reduce total inter-node communication by switching to intra-node communication whenever possible

2. reduce inter-node network contention by preventing processors on the same node to either all send or all receive over the network at the same time.

The standard broadcast uses a binary tree of height $\log_2 P$ over the processes rooted at the sending process for distributing data. Depending on the rank to processor allocation it can easily happen that all processes within an SMP node at the same time have to send to children placed at other nodes. A simple way to prevent contention of this sort is to use instead a binary tree over the n SMP nodes. One of the processes on each node is designated as *master*, the other(s) being *slave(s)*. A binary tree is built over the master processes. When a master receives data, it first sends to its at most two children which are masters on other nodes, and then broadcasts to the $p-1$ slaves within the node. On the LAMP system where there is only one slave per node this is trivial, and done by point-to-point communication. This algorithm reduces the amount of communication over the network by a factor of p, and decreases the height of the tree as well. Figures 3 and 4 compare latency and bandwidth achieved with the improved broadcast algorithm with those for the standard MPICH implementation. The improved algorithm gives a considerable decrease in latency, and almost doubles the bandwidth.

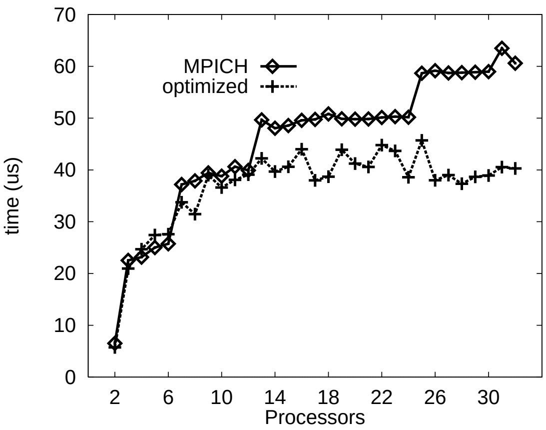

Figure 3: Latency for broadcast for increasing number of processors. Latency is measured by broadcasting a small packet of 1 byte.

A similar idea is used for barrier synchronization. Since little data is being sent around and contention does not occur, the gain is not as dramatic. For 32 processors a decrease in synchronization time of about 25% has been achieved.

Both improved algorithms follow the same outline. Let collop be the collective operation to be performed.

1. local, intra-node collop,

2. global, inter-node collop among masters,

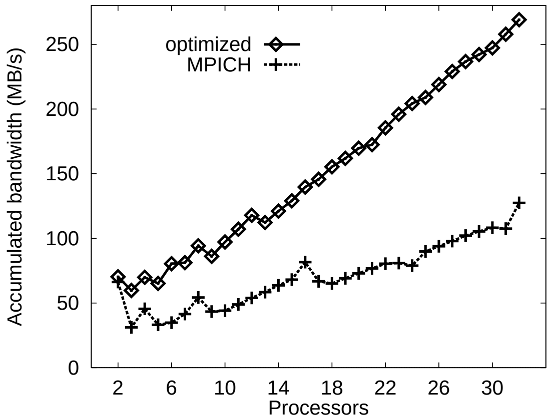

Figure 4: Accumulated bandwidth for broadcast for increasing number of processors. Bandwidth is measured by broadcasting a packet of 65536 bytes.

3. local, intra-node collop

where either the first or the last step is optional, or some slight variation of collop; furthermore, local and global operations should be overlapped as far as possible. However, even for a simple operation like broadcast, it is not clear that this scheme will lead to the best algorithm in case the load (number of processes) per SMP node differs significantly. This will become more prominent for the following collective operations.

In the standard MPICH implementation for all-to-all communication each process exchanges data with P other processes (itself included) in some (fixed) order. Thus P rounds of communication are required in which all processes send m data items to some other process. In the MPICH implementation deadlocks are prevented by using non-blocking sends and receives. As for broadcast, it can easily happen that all processes at a node simultaneously use the network (in both directions).

Following the scheme from broadcast and barrier, a natural procedure would be to solve an extended all-to-all problem only among masters, where in each round each master exchanges data from both itself and its slaves with some other master processes. This might be good for short messages: the number of global inter-node rounds decreases from P to n, and in each round p accesses to the network are traded for $p-1$ intra-node messages and a single network access for a message which is p times longer (containing data from master and all slaves). On the LAMP system, where each master has only one slave, the overhead for internal buffering needed for the masters seems too large for this algorithm to improve over the standard implementation. More importantly, for systems with a varying number of slaves per master the amount of data per master to be exchanged in each global round can vary between m and pm items. In other words, using the design scheme of broadcast the regular all-to-all problem gets transformed into an irregular problem. One way of dealing with this case is to pick more masters for nodes having a large number of processes, such that the number of slaves per master is roughly uniform; the benefit of shorter, more uniform messages has to be balanced against higher competition for the network. Solving the irregular problem optimally in general involves solving a scheduling problem, for instance a coloring problem on a suitably constructed communication graph; see for instance [7, 14].

A different approach to avoid network contention, which works well on the LAMP, is to introduce extra rounds to separate the communication of masters and slaves. The algorithm (for $p = 2$) is as follows:

1. all-to-all among masters (but no master exchanges data with itself)
2. each master sends to all non-local slaves
3. each slave sends to all non-local masters
4. all-to-all among slaves (no slave exchanges data with itself)
5. internal all-to-all on each node

The amount of data communicated is the same as in the standard algorithm, but the number of rounds increases from P to $p(n-1)+p(p-1)(n-1)+p^2$ or roughly pP. The algorithm can be expected to perform better than the standard algorithm when the amount of data to be sent per processor is large enough that saturation of the network would have occurred in the standard algorithm, and large enough to render the extra start-ups insignificant. Figures 5 and 6 show the bandwidth achieved for varying message lengths for 8 and 32 processors. The improvement seems worthwhile. The per processor bandwidth increases from about 4.12 MB/s to about 6.03 MB/s on 32 processors.

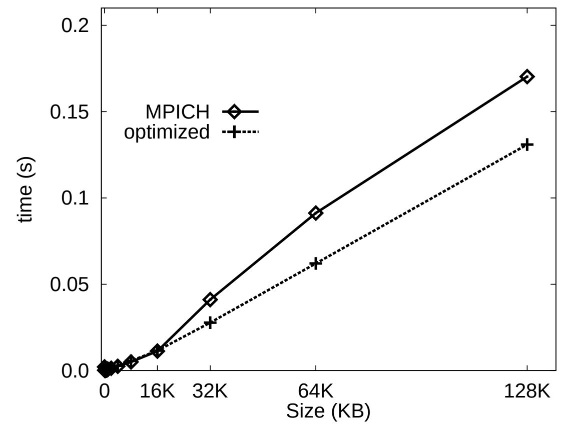

Figure 5: All-to-all communication among 8 processors. Data size is the amount of data exchanged between each pair of processors.

Again, the algorithm does not generalize well to the case where the number of processes per node partaking in the exchange differs by more than 1 (or some small number).

We also mention that the radix-all-to-all algorithm of [5], which gave very good results for smaller message lengths on the NEC SX-4 parallel vector supercomputer, does not perform well on the LAMP system.

We finally discuss improvements to the MPICH implementation of the `MPI_Allgather()` operation. The straightforward algorithm takes P communication rounds while communicating Pm data items per processor in total. This is suboptimal since the problem can be solved in $\log_2 P$ rounds with the same amount of data per processor [5]. On the SX-4 this latter algorithm gave a clear performance improvement over the MPICH implementation, but this is less evident on

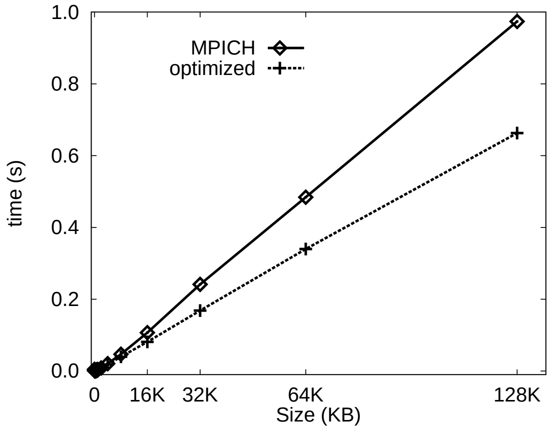

Figure 6: All-to-all communication among 32 processors.

the LAMP. A better implementation of the MPICH algorithm decreases network contention by separating the communication of masters and slaves, much the same way the all-to-all algorithm does:

1. all-gather among masters with master data
2. each slave sends its data to its master
3. all-gather among masters with slave data
4. each master sends gathered data to its slave

For steps 1 and 3 currently the linear, MPICH like implementation is being used. The algorithm is easier to generalize to the case of node imbalance than the all-to-all algorithm. This can be done in two ways. The first approach is to let the masters first gather data from all slaves using an efficient intra-node algorithm, then solving the resulting irregular all-gather problem among the masters, and finish with local broadcasts on all nodes. This approach is in accordance with the simple scheme from broadcast and barrier, and again illustrates how a regular problem turns into an irregular one of the same sort. The other approach is to do p separate rounds in each of which one process of each SMP node participates in an all-gather operation. Rounds where less than n processes take part will have to finish with a broadcast of the gathered data to the (non-full) nodes where no process participated. In absence of a simple performance prediction model, experiments will have to determine which of the two approaches is preferable. Performance of the all-gather implementations are shown in Figures 7 and 8.

4 Concluding remarks

With its powerful, but non-uniform, communication capabilities the LAMP system is representative of current SMP systems. The performance of the presently available (MPICH) implementations for the collective communication operations of MPI as compared to what can be achieved with by simple improvements, illustrate the need for new algorithms which better fit the characteristics of hierarchical multiprocessors such as SMP clusters. For the design and implementation of such algorithms a simple, but more accurate model than a “flat” uniform shared or distributed memory model is needed. A difficulty in supporting an MPI-like, process oriented programming model is that regular communication problems easily turn into their irregular counterparts. From

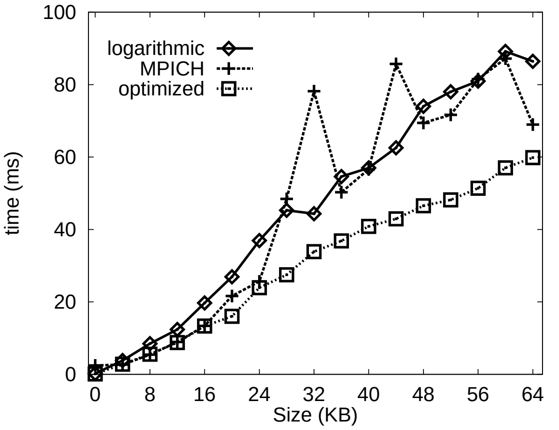

Figure 7: All-gather with 8 processes, short messages.

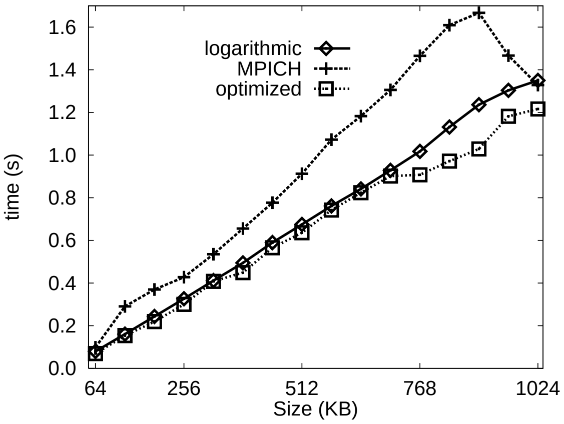

Figure 8: All-gather with 8 processes, long messages.

a practical point of view the irregular problems are most often much more difficult to solve efficiently.

We have emphasized the need for better algorithms by highlighting some important features common to contemporary parallel systems, and showed that there are considerable performance benefits waiting to be exploited.

References

[1] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.

[2] B. Alpern, L. Carter, and J. Ferrante. Modelling parallel computers as memory hierarchies. In *Working Conference on Massively Parallel Programming Models*, 1993.

[3] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). Technical Report UMIACS-TR-97-48, Institute for Advanced Computer Studies, University of Maryland, 1997.

[4] M. Baum, M. Gołębiewski, R. Hempel, and J. L. Träff. Dual-device MPI implementation for PC clusters with SMP nodes. In *Proceedings of the Message Passing Interface Developer's and User's Conference (MPIDC'99)*, 8 pages, in press, 1999.

[5] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[6] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High performance virtual machines (HPVM): Clusters with supercomputing apis and performance. In *8th SIAM Conference on Parallel Processing for Scientific Computing (PP97)*, 1997.

[7] D. Durand, R. Jain, and D. Tseytlin. Applying randomized edge coloring algorithms to distributed communication: An experimental study. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA95)*, pages 264–274, 1995.

[8] J. Edler, A. Gottlieb, and J. Philbin. The NECI LAMP: What, why, and how. In *Heterogeneous Computing and Multi-Disciplinary Applications. Proceedings of the eighth NEC Research Symposium*, 1998.

[9] M. Gołębiewski, M. Baum, and R. Hempel. High performance implementation of MPI for Myrinet. In *Parallel Computation. 4th International Conference of the ACPC*, volume 1557 of *Lecture Notes in Computer Science*, pages 510–521, 1999.

[10] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI –The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable imlementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[12] Myricom. *Myrinet Documentation*. `http://www.myri.com/scs/documentation`.

[13] J. Philbin and K. Li. Rationale and requirements for low cost scalable servers and parallel processors. Technical Report 95-008-3-0060-1N, NEC Institute, 1995.

[14] S. Ranka, J.-C. Wang, and M. Kumar. Irregular personalized communication on distributed memory machines. *Journal of Parallel and Distributed Computing*, 25:58–71, 1995.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI –The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.