

MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics *

Amith R. Mamidala Rahul Kumar Debraj De D. K. Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{mamidala, kumarra, ded, panda}@cse.ohio-state.edu

Abstract

The advances in multicore technology and modern interconnects is rapidly accelerating the number of cores deployed in today's commodity clusters. A majority of parallel applications written in MPI employ collective operations in their communication kernels. Optimization of these operations on the multicore platforms is the key to obtaining good performance speed-ups. However, designing these operations on the modern multicores is a non-trivial task.

Modern multicores such as Intel's Clovertown and AMD's Opteron feature various architectural attributes resulting in interesting ramifications. For example, Clovertown deploys shared L2 caches for a pair of cores whereas in Opteron, L2 caches are exclusive to a core. Understanding the impact of these architectures on communication performance is crucial to designing efficient collective algorithms. In this paper, we systematically evaluate these architectures and use these insights to develop efficient collective operations such as MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. Further, we characterize the behavior of these collective algorithms on multicores especially when concurrent network and intra-node communications occur. We also evaluate the benefits of the proposed intra-node MPI_Allreduce over Opteron multicores and compare it with Intel Clovertown systems. The optimizations proposed in this paper reduce the latency of MPI_Bcast and MPI_Allgather by 1.9 and 4.0 times, re-

spectively on 512 cores. For MPI_Allreduce, our optimizations improve the performance by as much as 33% on the multicores. Further, we observe upto three times improvement in performance for matrix multiplication benchmark on 512 cores.

1 Introduction

Recently, the supercomputing arena is witnessing an explosive growth in terms of raw computational power. This is fueled primarily by two major hardware advances: the emergence of multicore processors and the availability of superior commodity cluster interconnects offering high bandwidth and low latency data transfers. The recent induction of the Ranger [15] cluster exemplifies this phenomenon with core counts in the cluster exceeding 50,000. InfiniBand, which is being deployed widely, is used as the network interconnecting the different nodes of the cluster. Harnessing this massive amount of computational power for different scientific and engineering applications poses a fundamental challenge in the high performance computing domain.

MPI [13] has emerged as one of the primary programming paradigms for writing efficient parallel applications. It provides for a plethora of communication primitives with operations geared towards point-to-point and collective communications. Especially, collective operations such as MPI_Bcast, MPI_Allreduce and MPI_Allgather are important as they are widely used in a variety of application kernels [10]. Thus, improving their performance is key to enabling very high parallel speed-ups for parallel applications.

Significant research has been carried out in the past for improving collective communication using novel parallel algorithms and platform specific optimizations.

*This research is supported in part by DOE grants DE-FC02-06ER25755 and DE-FC02-06ER25749, NSF Grants CNS-0403342 and CCF-0702675; grants from Intel, Sun Microsystems, Cisco Systems, and Linux Networks; and equipment donations from Intel, AMD, Apple, IBM, Microway, PathScale, SilverStorm and Sun Microsystems.

In fact, these are very tightly interlinked with the latter influencing the design of the parallel algorithms employed. In this context, one main question that arises is whether the collective routines currently employed perform optimally on the new multicore platforms or do they require systematic re-thinking and evaluation.

Recent designs of multicore processors feature various architectural attributes resulting in several interesting ramifications. One such architectural feature is the design of multi-level cache hierarchies. The two broad strategies deployed in the current day multicores exist in processors from Intel and AMD. Intel processors provide a shared L2 cache where as AMD multicores deploy HyperTransport links for quick data transfers. Also, these architectures employ efficient cache coherency protocols for cases where the caches cannot be shared across the nodes. However, irrespective of these different hierarchies, both the systems enable very fast sharing of data across the cores. Further, to scale the bandwidth available to the coherency traffic, the cores are either connected via multiple buses as in Intel or by 2-D mesh HyperTransport links. Apart from providing good data movement capabilities across the processing cores, the caching hierarchies are useful in reducing the pressure on the memory bandwidth. This is especially true for scenarios when concurrent network transactions and intra-node communications occur.

Understanding the effects of these multicore specific characteristics can potentially benefit different collective operations and guide the design process to achieve optimal algorithms. In this paper, we optimize the widely used collective operations such as MPI_Bcast, MPI_Allgather, MPI_Alltoall and MPI_Allreduce over the multicore clusters. Further, we characterize the behavior of these algorithms taking into account the impact of different architectural attributes of multicores. Specifically, we aim to answer the following questions.

- How can the current algorithms for MPI_Bcast and MPI_Allgather be re-designed or adapted to existing multicore architectures?
- What optimizations can be carried for collectives such as MPI_Allreduce and MPI_Alltoall within a node taking into account data movements across the different cache/memory modules?
- What is the interplay between cache-to-memory and memory-to-network data transfers for the existing Intel Clovertown clusters?
- What are the benefits of the collective operations for an important application kernel, matrix multiplication?

We have implemented our designs and integrated them into MVAPICH [9] which is a widely used MPI implementation for InfiniBand clusters. We have evaluated our designs over the dual socket, quad-core Intel Xeon Clovertown and quad socket, dual-core AMD Opteron systems. The optimizations proposed in this paper reduce the latency of MPI_Bcast and MPI_Allgather by 1.9 and 4.0 times on 512 cores respectively. For MPI_Allreduce, our optimizations improve the performance by as much as 33% on the multicores. We also see performance gains upto three times for the matrix multiplication benchmark. Moreover, we also demonstrate the benefits of Opteron's architecture for intra-node MPI_Allreduce compared to Intel's Clovertown.

The rest of the paper is organized in the following way. In Section 2, we provide the background of our work. In Section 3, we explain the motivation for our scheme. In Section 4, we discuss the detailed design issues. We evaluate our designs in Section 5 and talk about the related work in Section 6. Conclusions and Future work are presented in Section 7.

2 Background

In this section, we first provide an overview of Intel Clovertown and AMD Opteron multicore platforms. We then present the standard algorithms currently used for different collective operations.

Intel Clovertown Architecture: We describe some of the important details of Intel Clovertown multicore architecture [6] in this section. These chipsets consist of dual sockets with quad cores in each of the sockets and a pair of adjacent cores share a common L2 level cache of size 4MB. Also, each of the socket has its own Front Side Bus(FSB) with a bus based snooping protocol to handle cache coherency.

AMD Opteron Architecture: AMD multicore Opterons [1] are based on NUMA architecture with each of the sockets sharing independent memories. The number of cores in each of the sockets can vary from one to four with the latest Barcelona systems. All of the cores have independent L2 caches. Point-to-point HyperTransport links provide the required bandwidth scalability between the cores. Further, these links are connected by a 2-D mesh topology providing for scalable and less congestion-prone on-chip interconnection network.

Collective Algorithms: In this section we present the different algorithms used for MPI collectives. MPI_Bcast uses multiple algorithms based on the message size. For small messages, binomial tree algo-

rithms are widely employed. Large message broadcast involves a more complex approach of doing a scatter followed by an allgather. For shared memory systems, pipelining is employed to allow overlap and improve buffer utilization [17]. The most popular algorithms for MPI_Allgather are recursive doubling algorithms for short messages and ring algorithm for large messages [3]. All of these algorithms are currently employed in MPICH [5, 16]. The recursive doubling algorithm consists of $\log(n)$ steps where the data for each step doubles. For MPI_Allreduce, pair-wise exchange is used for small messages across the network and reduce-scatter followed by an allgather is used for large messages. In reduce-scatter, the data is recursively halved and reduced after each step for a total of $\log(n)$ steps. For intra-node MPI_Allreduce, the most popular implementation uses shared memory where a specific process reduces the data in a shared buffer to be copied by everyone else afterwards. MPI_Alltoall [16] uses a pair-wise algorithm for large messages where every process directly exchanges messages with every other process. There are a total of $(n - 1)$ steps where in step i process with rank r communicates with process ranked $r \oplus i$.

3 Communication in Multicores

The advent of multicore processors presents several opportunities and challenges for designing efficient collective operations. For efficient design of these operations, it is important to first understand the communication trends over the multicores. In this section, we conduct different experiments involving both inter- and intra-node communication followed by those involving intra-node communication. Our rationale to focus also on intra-node optimizations is that the core count is expected to increase rapidly in the coming years. Thus, depending on the applications' characteristics, significant amount of communication time can be spent in exchanges messages within the node. Our first test measures the impact of L2 cache sharing in Intel Clovertown architecture.

L2 cache sharing in Intel Clovertown: To elucidate the impact of L2 cache sharing, we perform a simple micro-benchmark evaluation. In this test, a message of size 16,384 bytes is exchanged between a set of nodes as shown in Figure 1. The message buffers on each node are shared across all the cores. In this way, one process per node initiates the message transfer and all the other processes on the nodes copy data directly from this buffer. The test is run by varying the number of processes on each node to a maximum of eight processes

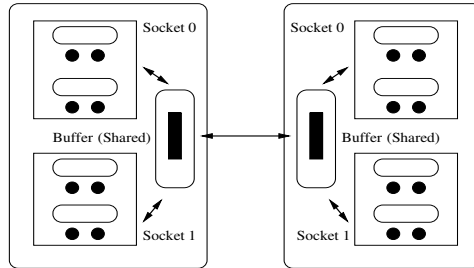


Figure 1. Send-Recv latency test

per node. The results of this benchmark are shown in Figure 2(a). As seen from the graph, latency remains constant for processes upto four on each node and then steps to a higher level. This corresponds to core count of eight in the Figure. In our test all the four processes are scheduled on the same socket. This is clearly due to the impact of L2 cache sharing and faster cache-coherency protocol within each socket of the Clovertown platform. This attribute potentially benefits MPI level broadcast operations where the same data is sent to multiple processes.

Reduction operations within a node: One of the primary methods of doing reduction within a node has been to use shared memory segments to copy the data into and do the necessary arithmetic. For example, in MPI_Allreduce all the processes copy the data into a shared buffer, followed by reduction by one of the process. After the reduction, the data is copied back into the respective receive buffers. Though this technique is optimal for short messages, it does not scale for large messages. This is shown in the Figure 2(b) where a more distributed approach of point-to-point reduce-scatter followed by allgather performs the best. This approach fares better than the pair-wise exchange and the basic shared memory method though there are extra memory copies involved. Thus, more optimized shared memory approaches need to be investigated. Further, the cache and memory hierarchies play a crucial role in determining the optimality of these approaches.

Scheduling data movement over HyperTransport: In this test, we conduct different tests across two pairs of communicating processes. Two of the processes are running on one socket while the other two are on a different socket. Each process in a pair sends and receives message from the other process separated by one hop distance over the HyperTransport. However, there are three different methods of exchanging these messages. The first method is to use non-blocking send/receive functions provided in MPI. The other method is to use the

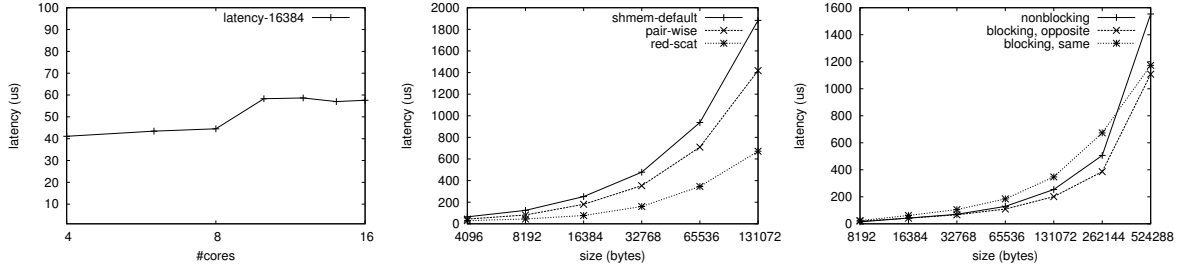


Figure 2. (a)Effects of L2 cache sharing, (b)Intra-node Reductions and (c)Scheduling communication over HyperTransport

blocking modes of these function. Even with this case, the sends can be issued by both the pairs in one direction over HyperTransport followed by those from other direction. Or, to utilize the full bidirectional bandwidth of HyperTransport, sends can be issued in opposite directions. We test all these cases and notice that the last method which utilizes the full bidirectional bandwidth performs the best, as shown in Figure 2(c).

In the remaining sections of the paper, we explore all these issues and use the insights obtained to optimize different collective operations such as MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall.

4 Design

In this section, we discuss our proposed optimizations for collective functions: MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. For these collective operations, we focus on large message optimizations in this paper.

4.1 Optimized MPI_Bcast and MPI_Allgather

As discussed in earlier sections, the multicore clusters provide faster sharing of data within a node. Further, data transfer operations can be overlapped within a node and across the nodes. These useful insights can be channeled to develop efficient MPI_Bcast and MPI_Allgather collective operations.

From section 2, the default algorithm used to design MPI_Bcast for large messages is a scatter operation followed by an all-to-all broadcast or MPI_Allgather of the data. This algorithm performs optimally for clusters having one process per node. However, when multiple processes are launched on the same node this algorithm needs to be modified.

The basic building block of our re-designed broadcast algorithm is the shared memory based all-to-all broadcast of the data using a ring topology. In this design,

we allow for a single process per node to perform all the inter-node collective communication while the remaining processes are involved in intra-node communication. The process which does all the inter-node communication is called the *leader*. In the new design for MPI_Bcast, the collective communication can be broken down into three steps. In the first step, the root process copies the message to the shared buffer and the leader of the node scatters the message to all the other leaders. A binomial recursive halving algorithm is used to scatter the data. In the second step, all the leaders are involved in inter-node ring communication as shown in Figure 3. This step corresponds to the MPI_Allgather of the data. The number of steps involved would be $n-1$ where n is the number of leaders. The third step involves a copy of the data from the network buffer to the user buffers. This step can be done after the entire network communication is done or can be overlapped. We have taken the latter approach by overlapping network transfers with shared memory copying of data. In our approach the network buffers are shared across all the processes to facilitate direct copy of data.

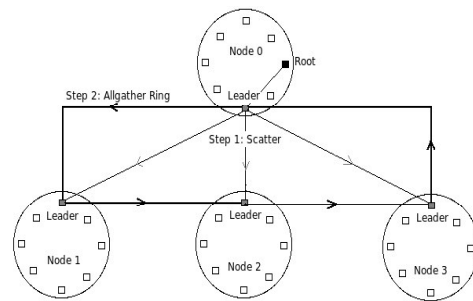


Figure 3. Optimized Broadcast Algorithm

Impact of L2 cache sharing of Intel Clovertown:

In this section, we try to understand the behavior of the above mentioned algorithm on an Intel Clovertown multicore cluster. As discussed above, after each communication step of the ring, the local processes copy the data

from the shared buffer to the respective receive buffers. Since the processes share L2 Caches and they copy the same chunk of data from shared memory after each step, several interesting observations can be made about the behavior of the collective algorithm. Firstly, the intra-node communication overhead would be greatly reduced due to the high bandwidth of L2 caches, provided the communication buffers are appropriately cached. Secondly, with the increase in the number of cores participating in the collective, we do not expect the latency of the collective to change till the intra-node copy time exceeds the inter-node latency. For example, if the extra cores participating in the operation belong to the same socket, the latency should marginally increase as the caches within the socket are efficiently shared. Only when the cores added are derived from two different sockets do we expect an appreciable jump to occur in the latencies. Thus, we observe that the shared L2 Cache architecture effectively reduces the contention on the memory especially when concurrent network transactions occur together with intra-node message copying. We experimentally evaluate these effects in the performance evaluation section of the paper.

4.2 Optimized MPI_Allreduce

In the default shared memory algorithm for allreduce, only a single core does the reduce, while other cores are idle during that time. In this section, we propose two algorithms for allreduce collective that utilize the computational power of multiple cores inside a node. The first algorithm is a very basic approach of parallelizing the computation. In the second algorithm, we propose mechanisms to improve the performance even further by leveraging the architecture of multicores.

In the basic approach of doing MPI_Allreduce, the computation is delegated to all the cores in a straight forward manner. We illustrate this with an example consisting of four processes within a node, Figure 4. As described in the figure, each process copies its data into different blocks of the shared memory region based on their respective ranks (step 0 of the figure). Please note that after this step, the respective data blocks will be located closer to the process either in the cache in the case of Intel Clovertown and AMD Opteron or also in memory as in the case with NUMA Opterons. In the next step, each block of shared memory is divided into four sub-blocks corresponding to the number of processes. Now, every process operates on one sub-block from every block of the shared region. For example process 0 reduces sub-blocks 0a, 0b, 0c and 0d to the memory space of 0a and the result is sub-block 0R. Similarly process

1 reduces sub-blocks 1a, 1b, 1c and 1d resulting in sub-block 1R and so on. In step 3, each process copies the entire zero block, containing sub-blocks 0R, 1R, 2R, 3R, to their own local memories.

Now, a close observation leads to the fact that the last algorithm can suffer from cache/memory access contention. This is because all the processes follow the same order of accessing the blocks in the shared memory region. Thus, we design another allreduce algorithm that keeps the parallelism of the previous one, but in addition utilizes a cyclic approach to eliminate the drawback of the previous algorithm. In this approach, steps 0 and 1 are same as above except that the reduction and gathering of the data is done in a cyclical manner. In the example considered, process 0 reduces in order 0a, 0b, 0c, 0d whereas process 1 reduces in order 1b, 1c, 1d, 1a. Similarly, in step 3, each process copies the sub-blocks of the result into its local memory in a cyclic fashion. This important optimization is expected to yield significant benefits in modern multicore machines having distributed cache and memory hierarchies.

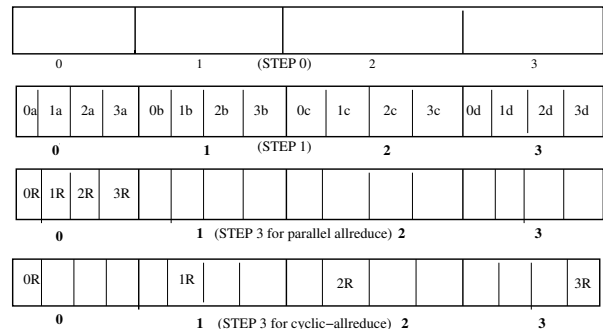


Figure 4. Optimized shared memory allreduce schemes

4.3 Optimized Shared Memory MPI_Alltoall

In this section, we explain the scheduling of the operations for MPI_Alltoall over the HyperTransport links of Opteron multicore architecture. The basic idea behind our approach is to completely utilize the bidirectional bandwidth of the links interconnecting the different cache and memory modules of the Opteron architecture.

As explained in the earlier section 3, we convert all the non-blocking operations into their blocking counterparts. Since MPI_Alltoall uses a pair-wise algorithm with every process talking to every other process, there would be $n(n - 1)$ pairs in total. For each of the pairs,

communication schedule is constructed in such a manner that the data transfer happens in both directions at any given point in time. This schedule would keep the links busy through out the whole operation.

5 Performance Evaluation

In this section we compare the performance of the new designs of MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. We first explain the testbed used in our evaluations.

5.1 Experimental Testbed

We have conducted our tests on an Intel cluster of 64 nodes for inter- and intra-node collectives. Each node is a dual Intel Xeon Clovertown processor with quad core possessing a shared 4MB cache for two cores. The nodes are interconnected by InfiniBand HCAs in DDR mode. The operating system used is Redhat Linux AS4. For opteron, we did not have access to a large scale cluster. For intra-node collectives, we evaluated our designs on a quad-socket, dual core opteron. Each of the cores contains a 1MB L2 cache. We have evaluated our designs on varying configurations such as 64X4 (i.e.64 nodes with 4 process on each node) and 64X8.

5.2 MPI_Bcast Latency

We measure the MPI_Bcast Latencies and the results are as shown in Figures 5(a) and 5(b) for 64 nodes of the cluster. In each of the graph we compare our new design with the original design. As discussed earlier, the original design employs a point-to-point reduce-scatter followed by allgather without taking into account the locality of the processes. The results for original design show different performance when the processes are scheduled in a block and cyclic fashion. As can be seen the new design proposed in this paper gives better performance for any distribution of processes. We obtain speed-ups of upto 1.78 and 1.90 for the two configurations.

In the second experiment, we measure MPI_Bcast latency on four Intel nodes by gradually increasing the cores participating in MPI_Bcast from each node. We go from one to eight processes per node since each node has 8 cores. The results can be seen in the Figure 5(c). In each of the configuration, two experiments were conducted with processes being scheduled in a node in socket-block or socket-cyclic manner. As can be seen from the Figure 5(c), initially for 4X1 both the tests show the same latency since both the configurations are

same. As we increase the processes to 4X2 the socket-cyclic method of scheduling causes greater latency than socket-block. This is because, in socket-block method the processes share L2 Cache and hence the copy time is negligible which is not the case in socket-cyclic. This continues till 4X4 configuration and then after that both the methods produce the same latency as both the methods have processes on each of the sockets. Please note that in Figure 5(a), the new scheme performs poorly for message less than 1 MB. This is because by default the processes are distributed in socket-cyclic manner in our test cases.

5.3 MPI_Allreduce Latency

We have evaluated the performance of the newly proposed parallel algorithms with the existing allreduce algorithms for the two different multicore architectures. Figures 6(b) and 6(c) show the performance of existing and the two new allreduce algorithms for the AMD Opteron and Intel Clovertown architecture, respectively. The legend “shmem-parallel” corresponds to the basic algorithm where all processes accesses different blocks in the same order. The “shmem-cyclic” legend refers to the optimized parallel algorithm where the processes access the different blocks in a cyclical manner. From the figures it can be seen that the shmem-cyclic performs the best of all the three. We improve the performance by 23% on NUMA opteron and by 32% on Intel Clovertown. One interesting observation to make is that on Intel Clovertown, both shmem-cyclic and shmem-parallel perform identically where as on NUMA Opteron, shmem-cyclic performs the best. Owing to the NUMA and the different HyperTransport links connecting the different cache and memory modules, AMD Opteron fares better due to increased level of parallelism and lower contention.

5.4 MPI_Alltoall Latency

In this section, we compare the proposed optimization of scheduling the send/receive operations over the bi-directional HyperTransport links connecting different sockets in AMD Opteron. The results are described in Figure 6(c). As shown in the figure, the new optimization improves the latency of MPI_Alltoall by around 15% compared to the original scheme. We expect to see much higher performance gains as the core counts increase per socket.

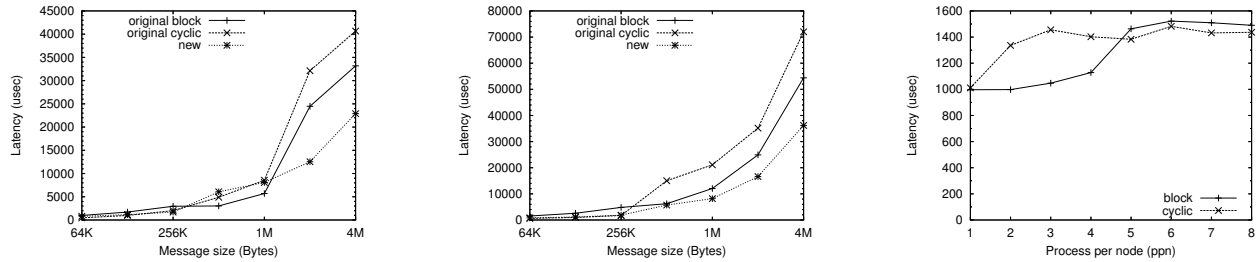


Figure 5. Bcast Latency: (a)64X4, (b)64X8 and (c)Effect of Cache on 4 nodes

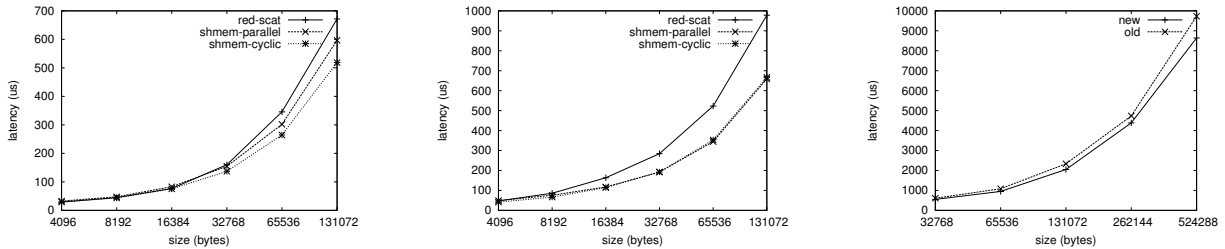


Figure 6. (a)Allreduce: Opteron, NUMA, (b)Allreduce: Intel Clovertown and (c)Scheduling in Alltoall

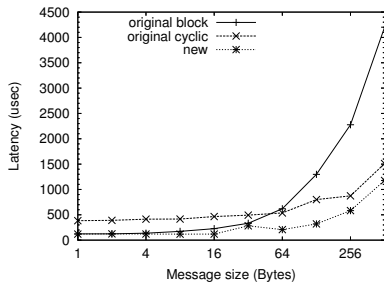


Figure 7. Allgather Latency

5.5 MPI_Allgather Latency

Figure 7 shows the results for MPI_Allgather latency on 64 node Intel Clovertown cluster. Depending on the message size, the new design outperforms the current one by a factor of up to four on 64x8 configurations.

The benefits of MPI_Allgather can also be seen from the Matrix Multiplication Kernel which uses MPI_Allgather collective communication. The application is run with different matrix sizes (double data type matrix elements) starting from 16x16 and up to 512x512. Figure 8 shows the performance gain with increasing message sizes. For 512x512 sizes, we improve the per-

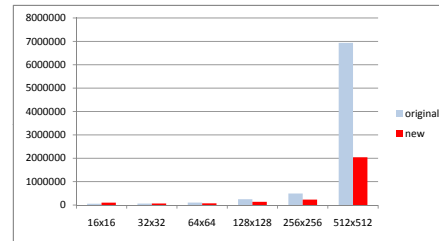


Figure 8. Matrix Multiplication

formance by as much as three times compared to the default implementation.

6 Related Work

Utilizing shared memory for implementing collective communication has been a well studied problem in the past. In [17], the authors propose using remote memory operations across the cluster and shared memory within the cluster to develop efficient collective operations. They apply their solutions to Reduce, Bcast and Allreduce operation. The authors evaluate their approach on IBM-SP systems. In this paper, we evaluate the SMP-based collectives taking into account the multicore aspect of the clusters. Specifically, we develop al-

gorithms and gain insights into their performance on the latest platforms from Intel and AMD. Optimizing collectives on hierarchical architectures have also been studied by Bull [4] in the context of Itanium based NUMA machines. In our work, we focus on the Intel and AMD based multicore systems.

In [2, 12], the authors implement collective operations over Sun systems. However, in their design and evaluation they do not take into account any cache effects. Instead, their designs focus on optimal utilization of memory bandwidth by taking advantage of multiple memory banks. In [18], the authors improve the performance of send and recv operations over shared memory and also apply the techniques for group data movement. Collectives for single process per node over InfiniBand have been studied for MPI_Barrier, MPI_AlltoAll, MPI_Allgather [7, 11, 14] based on RDMA techniques. Shared memory collectives using RDMA have been explored in [8]. The benefits obtained here were due to savings from avoiding copy costs.

7 Conclusions and Future Work

Optimizing MPI collective communication on emerging multicore clusters is the key to obtaining good performance speed-ups for many parallel applications. However, designing these operations on the modern multicores is a non-trivial task. On the other hand, modern multicores such as Intel's Clovertown and AMD's Opteron feature various architectural attributes resulting in interesting ramifications. For example, Clovertown deploys shared L2 caches for a pair of cores where as in Opteron, L2 caches are exclusive to a core. Understanding the impact of these architectures on communication performance is crucial to designing efficient collective algorithms. In this paper, we have systematically evaluated these architectures and used the insights to develop efficient collective operations such as MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. Further, we characterized the behavior of these collective algorithms on multicores especially when concurrent network and intra-node communications occur. We also evaluated the benefits of multicore Opteron's architecture for intra-node MPI_Allreduce compared to Intel's Clovertown. The optimizations proposed in this paper reduce the latency of MPI_Bcast and MPI_Allgather by 1.9 and 4.0 times on 512 cores respectively. For MPI_Allreduce, our optimizations improve the performance by as much as 33% on the multicores. Further, we see performance gains upto three times for the matrix multiplication benchmark on 512 cores. For our future work, we plan to carry

out more in-depth design and evaluation of optimal algorithms for all the important collectives taking into account the on-chip interconnection topologies.

References

- [1] AMD. <http://www.AMD.com/opteron>.
- [2] M. Bernaschi and G. Richelli. MPI Collective Communication Operations on Large Shared Memory Systems. In *Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop*, 2001.
- [3] 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 in Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.
- [4] BULL. <http://www.bull.com/hpc/>.
- [5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.
- [6] Intel. <http://www.intel.com>.
- [7] S. P. Kini, J. Liu, J. Wu, P. Wyckoff, and D. K. Panda. Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *EuroPVM/MPI*, Oct. 2003.
- [8] A. R. Mamidala, A. Vishnu, and D. K. Panda. Efficient Shared Memory and RDMA based design for MPI-Allgather over InfiniBand. In *EuroPVM/MPI*, 2006.
- [9] MVAPICH. <http://mvapich.cse.ohio-state.edu>.
- [10] NASA. NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Software/NPB/>.
- [11] A. R. Mamidala, J. Liu, and D. K. panda. Efficient Barrier and Allreduce InfiniBand Clusters using Hardware Multicast and Adaptive Algorithms. In *Proceedings of Cluster Computing*, 2004.
- [12] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI Collectives on Clusters of Large-scale SMP's. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.
- [13] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition*. The MIT Press, 1998.
- [14] S. Sur, U. Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda. High performance RDMA based All-to-All Broadcast for InfiniBand Clusters. In *(HiPC)*, 2005.
- [15] TACC. <http://www.tacc.utexas.edu/resources/hpcsystems/>.
- [16] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. In *Euro PVM/MPI*, 2003.
- [17] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast Collective operations using Shared and Remote memory Access Protocols on Clusters. In *International Parallel and Distributed Processing Symposium, 2003*, 2003.
- [18] M.-S. Wu, R. A. Kendall, and K. Wright. Optimizing Collective Communications on SMP clusters. In *ICPP 2005*, 2005.