Hierarchy Aware Blocking and Nonblocking Collective Communications-The Effects of Shared Memory in the Cray XT environment

Richard L. Graham, Joshua S. Ladd, Manjunath GorentlaVenkata
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN
U.S.A.
{rlgraham, laddjs, manjugv}@ornl.gov

Abstract

MPI Collective operations tend to play a large role in limiting the scalability of high-performance scientific simulation codes. As such, developing methods for improving the scalability of these operations is critical to improving the scalability of such applications. Using recently developed infrastructure in the context of the FASTOS program, we will study the performance of blocking collective operations, as well as those of the recently added MPI nonblocking collective operations taking into account both shared memory and network topologies.

***KEYWORDS*—MPI, Collective Optimization, Hierarchy Aware**

1 Introduction

Because of their absolute performance and scalability, MPI collective operations tend to play a significant role in limiting the scalability of high-performance scientific simulation codes. Given that MPI collective routines are beyond the direct reach of an application developer, it is critically important to application scalability that MPI collectives perform well over an arbitrary memory subsystem. Optimizing collective performance is an increasingly complex problem, primarily because of the increasingly heterogeneous memory subsystems found on current and emerging high performance computing (HPC) systems, most of which possess a myriad of access mechanisms replete with varying latencies. Further complicating the matter is the fact that as processor counts increase, network and memory resources become evermore scarce. In order to continue scaling-up, next generation collective implementations will need to additionally be able to effectively utilize and sustain a finite pool of resources.

Hierarchy aware collectives have long been recognized as a partial solution to the problem of optimizing MPI collectives over heterogenous memory subsystems. Some current implementations of hierarchy aware collectives tend to be rigid and non-extensible, hardwired for a specific platform or memory subsystem (e.g. shared memory and point-to-point) thus making it difficult or impossible to extend the software infrastructure to accommodate new and/or emerging memory subsystems, thus resulting in the need to re-implement the suite of collectives from scratch. Other hierarchical implementations have been built in terms of existing MPI routines, e.g. using sub-communicators to create hierarchies. This approach adds a large amount of latency as collective routines must traverse an increasingly deep software stack with each new level in the hierarchy. Beyond high latency, this approach also makes it difficult to pipeline data and consumes a large amount of memory resources. The former approach is unattractive from the perspective of portable performance and code extensibility, the latter approach is simply not scalable. Clearly, there is a need for hierarchy aware collectives that are portable, extensible, and scalable.

In this paper, we present a portable, extensible, and scalable software infrastructure implemented within the Open MPI code base that supports blocking and non-blocking hierarchical collective operations. Our collectives are implemented within the Open MPI modular component architecture (MCA) as a collection of new frameworks that are able to identify available memory subsystems and load system specific collective components. These basic collective, or **BCOL**, components, can be strung together in a hierarchical chain with each component responsible solely for the execution of the collective on its level, the only interaction between bcol levels is when information must be passed from one level of the hierarchy to the next. This approach is easily extensible; in order to accommodate an arbitrary memory subsystem, all that is required is the inclusion of two new components; one which encapsulates a rule for subgrouping processes within a communicator (an **SBGP** component) and another which encapsulates the subsystem specific collective implementations (a **BCOL** component). For example, in our approach we have a point-to-point bcol that encapsulates PML specific collective implementations (MPI's point-to-point transfer mechanism), a bcol that encapsulates InfiniBand specific collectives, and a bcol for optimized shared memory. With only these three bcols we are able to create several hierarchical collectives at runtime by simply by passing the desired MCA parameters such as: shared memory and point-to-point, infiniband over tcp, shared memory and InfiniBand. When InfiniBand recently introduced its new new ConnectX-2's CORE-Direct support for creating high performance, asynchronous collective operations that are managed by the host channel adapter (HCA), we simply added a BCOL component to support this new transfer mechanism.

The remainder of this paper is organized as follows; Section 2 provides an overview of previous work on hierarchical collectives. Section 3 describes our design and implementation of a collection of new MCA frameworks within the Open MPI code base. Results of numerical experiments are presented and discussed in Section 4. Finally, conclusions and future work are discussed in Section 5.

2 Related Work

Much of the work on collective communication hierarchies occurs in the context of grid computing, aimed primarily at handling clusters connected by a relatively low-performance network, with limited connectivity, which is typified by the work described in [9, 3, 4, 5]. On the theoretical side, [3] provides a flexible parameterized LogP model for analyzing collective communication in systems with more than one level of network hierarchy.

In the context of HPC, most of the work on hierarchical collectives has been exploiting the low-latency shared memory communication available between processes on a given host [11, 2]. For example LA-MPI, Open MPI, MVAPICH, and MPICH2 have such support. This includes shared memory algorithms for MPI Bcast, MPI Reduce, and MPI Allreduce. However, an important conclusion draw in [2] is that shared memory collectives remain difficult to implement efficiently because the performance is highly dependent on characteristics of a particular architecture's memory subsystem. Non-uniform memory architectures, cache sizes and hierarchy, together with process placement greatly influence the performance of shared-memory collective algorithms.Implementations that combine both hierarchical and shared memory collectives are discussed in many works [8, 10, 11].

Zhu et al. provides implements hierarchy aware collectives in MPICH2 [14] based entirely on message-passing primitives that exploits the two-level hierarchy found on modern symmetric multiprocessors (SMP) and further develop a performance model to determine the how beneficial it is to use shared memory. Sanders and Träff present hierarchical algorithms for MPI Alltoall [6, 12] and MPI Scan [7] on clusters of SMP nodes. Other than their effort, most hierarchical work has centered around algorithms for MPI Bcast, MPI Reduce, MPI Allreduce, MPI Barrier, and MPI Allgather.

Wu et al. provide an excellent discussion [13] of SMP-aware collective algorithm construction in terms of shared-memory collectives, shared-memory point-to-point communication, and regular network communication. Their algorithmic framework also overlaps inter-node communication with intra-node communication. This approach generally pays the largest dividends in the case of medium-sized messages, where the message is large enough to amortize the additional overhead introduced by the non-blocking communication, yet is small enough that it is not dominated by the inter-node message transfer time.

3 Design Overview

The scalability of many scientific applications is often limited by the scalability of the collective operations. As such, optimizing the scalability and performance of these operations is vital to these applications, and much effort is put into improving the performance of these operations. With fifteen such blocking operations, and a corresponding set of nonblocking collective operations scheduled to be defined in the version three of the MPI standard, the effort involved in optimizing such collective operations is large. In particular, the effort to keep such operations performing close to optimal on new systems continues to be a challenge as new optimization opportunities arise. Therefore, a key design goal of this new implementation of hierarchical collectives is to preserve as much previous optimization work, when new system configurations appear, thereby minimizing the effort to take advantage of new system capability.

Therefore, the following design principles are used: 1) Discovery of system topology is decoupled from the implementation of the MPI collective algorithms. 2) The topology discovery functions are used to create subgroups within the MPI communicator. These subgroups overlap only in that on rank within the group, the local leader, appears in at least one other group. 3) The MPI-level collective operations are network agnostic, and described in terms of collective operations within the local groups. 4) Multiple version of the basic collective operations are supported. 5) Resources, such as data buffers, may be shared between subgroups, with out breaking the abstraction barriers defined for the basic collectives.

As an example, Figure 1 represents the communication

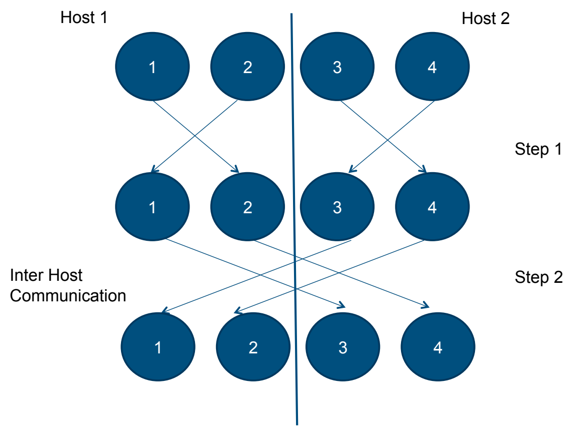

Figure 1. Recursive doubling barrier algorithm. The circles represent MPI ranks running on two different nodes, two ranks per node. The arrows represent data being sent from one rank to another.

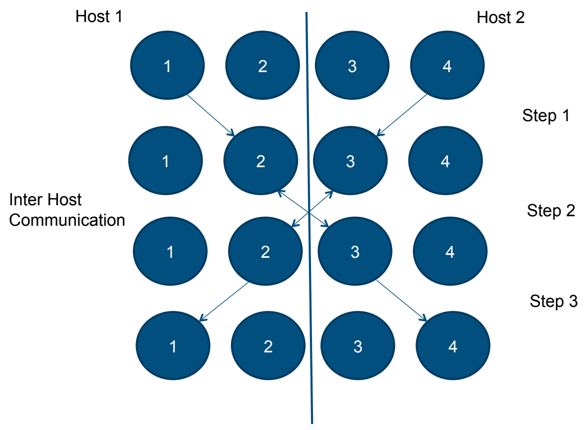

Figure 2. Hierarchical barrier algorithm. The circles represent MPI ranks running on two different nodes, two ranks per node. The arrows represent data being sent from one rank to another.

pattern for a four rank MPI barrier operation using the recursive doubling algorithm. In an implementation of this operation that does not consider topology, the same communication primitives will be used for all data exchange, such as MPI-level point-to-point data primitives.

However, if one exploits the system hierarchy, a different algorithm may be used which minimizes traffic over the network between hosts. Such an approach is depicted in Figure 2. In such an approach, fan-in to the local leader (rank two on host one, and rank three on host two) may be performed using shared memory communications, with the local leaders participating in the recursive doubling barrier, and then the rest of the local ranks notified of completion by their respective local leaders. Such an approach, while involving more communication phases than a simple recursive doubling algorithm, is attractive as it reduces external network traffic, and allows for the utilization of potentially lower latency local communications.

To support such an approach we have taken advantage of Open MPI's modular component architecture [1]. We have added a new hierarchical collective collective operation component to the COLL framework, which is called ML. We also added two new collective frameworks, one which supports hierarchy discovery which is called SBGP, and is short for subgrouping. The other supports basic collective operations, which is called BCOL, and is short for basic collectives. The architecture is depicted in Figure 3.

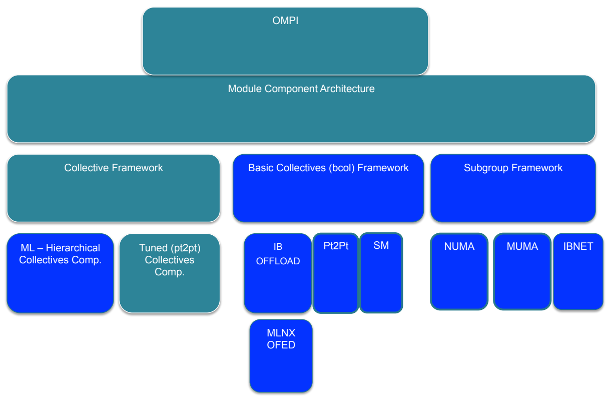
OMPI
Module Component Architecture
Collective Framework
Basic Collectives (bcol) Framework
Subgroup Framework
ML – Hierarchical Collectives Comp.
Tuned (pt2pt) Collectives Comp.
IB OFFLOAD
Pt2Pt
SM
NUMA
MUMA
IBNET
MLNX OFED

Figure 3. Support for the collective operations within the Open MPI code base.

The SBGP framework includes components that define subgrouping rules. Any type of subgrouping rule may be defined, but typically these will include rules relating to communication capabilities. We support rules for finding ranks that share sockets, ranks on the same host, and ranks that can communicate using MPI's point-to-point communication capabilities. One may also implement rules for forming subgroups based on the ability to utilize certain network capabilities, network topology, and the like.

The BCOL framework supports basic collective operations and needs to implement a full set of MPI collective operations. These includes support for mpi-like operations, such as barrier, and others, e.g. fan-in or fan-out. BCOL components are implemented to support specific communication capabilities. We support components that are optimized for shared memory communication, as well as a component that utilizes MPI-level point-to-point communications.

SBGP and BCOL modules are used in pairs to define a group of MPI ranks and the set of basic collective routines these will use when they participate in a give MPI-level collective operation. These pairing are flexible, and are determined at communicator construction time, based on runtime input parameters. In the four rank MPI barrier routine describe above, on-host subgroups will be determined by the discovery of the ranks that share a given host, and the inter-host group is determined by the ability to communicate using MPI-level point-to-point operations. Off host communication will take place using the MPI-level point-to-point communications, however on-host communication can take place either via optimized MPI-level shared memory collective operations, or using the MPI–level point-to-point communications.

This newly developed support will be used to study the performance of on-host barrier operations, as the first set of tests using these new capabilities.

4 Benchmark Results

In this section we describe early results using the new hierarchy aware collective communication capabilities. At this stage we only report results for on-host barrier algorithms. These take into account hierarchies within the shared memory node. Specifically they take into account cores that share the same socket. Future papers will present a much broader set of results.

4.1 Experimental Setup

The shared-memory collective routines are implemented within the trunk of the Open MPI code base. To measure MPI-level barrier performance we wrap the MPI_Barrier() calls in a tight loop, executing this loop for 10,000 iterations. We report the results based on the average time it took rank zero to complete the barrier call.

We ran numerical experiments on Jaguar, a Cray XT5 system, and Smoky, an AMD Opteron cluster, both housed at Oak Ridge National Laboratory. Jaguar is the world's fastest supercomputer available for open science. It has 18,688 compute nodes and in addition has nodes dedicated for providing login and other services. Each compute node contains two 2.6 GHz AMD Opteron (Istanbul) processors, 16 GB of memory, and a SeaStar 2+ router. The routers are connected in a 3D torus topology, which provides the interconnects with high bandwidth, low latency, and scalability. Each AMD Opteron processor has six computing cores and three levels of cache memory – 128 KB of L1 cache and 512 KB of L2 cache per core, and 6 MB of L3 cache that is shared among the cores. The compute nodes run Compute Node Linux micro-kernel, and service nodes run full-featured version of Linux.

Smoky is an 80 node test and development cluster at the Oak Ridge National Laboratory's National Center for Computational Science (NCCS). Each node contains four 2.0 GHz AMD Opteron processors, 32 GB of memory, an Intel gigabit Ethernet NIC, and a Mellanox Infinihost III Lx DDR HCA. Each AMD Opteron processor has four processing cores, and three levels of cache memory – 128 KB of L1 cache and 512 KB of L2 cache per core, and 8 MB of L3 cache that is shared among the cores. The compute nodes run Scientific Linux SL release 5.0, a full Linux operating system based on the popular Red Hat Linux distribution. This system is of interest, as it has a larger number of sockets than Jaguar does, and also has more total number of cores.

4.2 Results

Figure 4 displays the results for measuring the barrier performance as a function of the number of cores used in the calculation on Jaguar. This data was obtained using a two level hierarchy, taking into account how cores are mapped onto sockets. One measurement, labeled Shared Memory, was run using the shared memory optimized BCOL module, and the other, labeled pt-2-pt, was run using Open MPI's point-to-point communication system, which uses shared memory communications in this case. At 2 ranks, the shared memory barrier takes only 0.30 micro-seconds, and the point-to-point barrier takes 1.46 micro-seconds. At 12 ranks, the shared memory barrier takes 1.57 micro-seconds, and the point-to-point barrier takes 8.18 micro-seconds.

Figure 5 displays the results for measuring the barrier performance as a function of the number of cores used in the calculation on Smoky. This data was obtained using a two level hierarchy, taking into account how cores are mapped

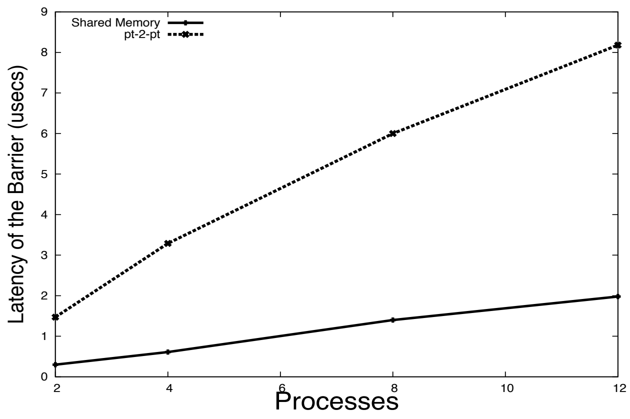

Figure 4. MPI_Barrier() performance on a single node of Jaguar using a two level hierarchy. Shared memory optimized collectives are used, as well as point-to-point based collective algorithms. The data is reported in units of micro-seconds.

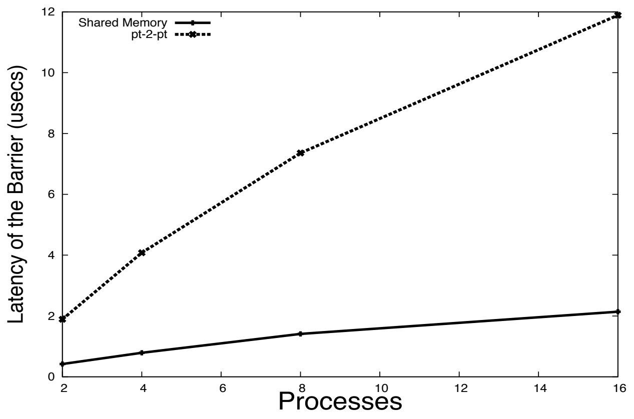

Figure 5. MPI_Barrier() performance on a single node of Smoky using a two level hierarchy. Shared memory optimized collectives are used, as well as point-to-point based collective algorithms. The data is reported in units of micro-seconds.

onto sockets. One measurement, labeled Shared Memory, was run using the shared memory optimized BCOL module, and the other, labeled pt-2-pt, was run using Open MPI's point-to-point communication system, which uses shared memory communications in this case. At 2 ranks, the shared memory barrier takes only 0.42 micro-seconds, and the point-to-point barrier takes 1.89 micro-seconds. At 16 ranks, the shared memory barrier takes 2.14 micro-seconds, and the point-to-point barrier takes 11.89 micro-seconds.

Figure 6 displays the results for measuring the barrier performance as a function of core layout with respect to sockets on Smoky. We see that in the two rank case, if both cores share a socket, the barrier time is 0.42 microseconds, and if they are on different sockets the barrier time is 0.51 microseconds. In the four rank case, the values are 0.79 micro-seconds and 0.90 micro-seconds, respectively. Figure 7 displays the data for a four rank barrier when the ranks are all on one socket, or spread out over four sockets. We see When they all share the same socket we see that the barrier time is 0.90 micro-seconds, but if they are spread out across four sockets, the barrier time is 1.33 micro-seconds.

4.3 Discussion

As the results indicate, taking advantage of the optimization opportunities to use shared memory to communicate directly between ranks in the MPI_Barrier() algorithm, rather than use the MPI point-to-point communication system greatly improves performance, both on Jaguar and on Smoky. On Jaguar the performance at two ranks is almost five times faster using the shared memory optimization, and at twelve ranks is also about five time faster. On Smoky the speed up at two ranks is about 4.5 times faster, and at sixteen ranks it is 5.5 times faster. This is not too surprising, as with the shared memory optimized collectives, the barrier amounts to setting local flags indicating what stage a given rank is in the communications, and reading those same values for the partner-ranks in the recursive doubling algorithm, with a similar approach used to manage the fan-in and fan-out algorithm. In addition, in the hierarchical algorithm we use the same set of control buffers for all shared memory phases of the MPI collective operation. In the point-to-point approach, the full MPI stack must be traversed to communicate with other ranks on the same host.

The performance data also shows that taking advantage of shared caches improves performance, and really comes as no surprise. We see that at on Smoky, when two ranks share a socket, the barrier performance is about 20% better then when they do not share a socket. For four ranks, the performance improvement is about 14% when the ranks share a single core, as compared to when then are distributed over two sockets, and 68% compared to spreading them out over four sockets.

To summarize, taking into account shared memory hierarchies does improve collective operation performance. In addition, bypassing the full MPI point-to-point matching logic leads to big performance gains.

5 Conclusions

This paper presents very early results from work to create a framework that supports hierarchical collective opera-

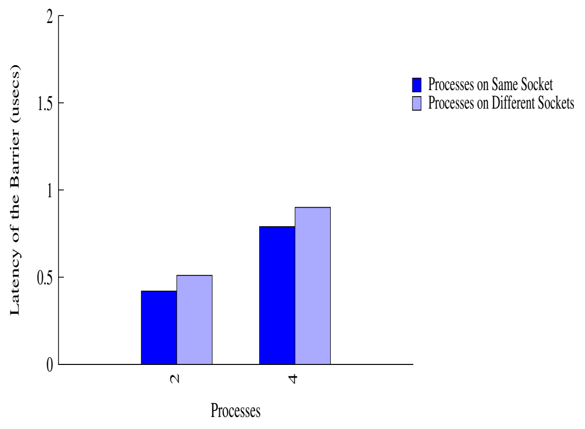

Figure 6. MPI_Barrier() performance as a function of process layout, using one and two sockets. The data is reported in units of micro-seconds, for two and four rank barriers.

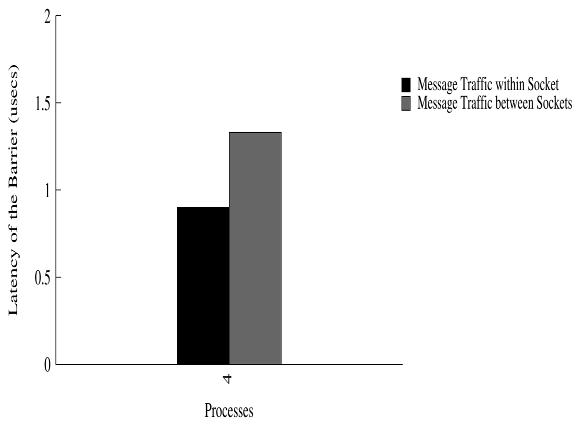

Figure 7. MPI_Barrier() performance as a function of process layout, using one and four sockets. The data is reported in units of micro-seconds, for a four rank barrier.

tions. This work has focused on on-node hierarchies, and initial results show the approach to be promising. Work continue to expand the support for cross-host communications, as well as for developing highly optimized support for specific network capabilities, such as the collective offload capabilities provided by new InfiniBand hardware. In addition to providing this support for blocking MPI collectives, support for nonblocking collectives, which has been voted into the MPI-3 draft standard, is being developed.

Acknowledgments

This work was done at Oak Ridge National Laboratory which is managed for the U.S. Department of Energy by UT-Battelle, LLC under contract No. DE-AC05-00OR22725.

References

[1] E. Garbriel, G. Fagg, G. Bosilica, T. Angskun, J. J. D. J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[2] R. L. Graham and G. Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130 – 140, 2008.

[3] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. pages 377 – 384. International Parallel and Distributed Processing Symposium (IPDPS), May 2000.

[4] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network performance-aware collective communication for clustered widearea systems. *Parallel Computing*, 27(11):1431 – 1456, 2001.

[5] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *SIGPLAN Not.*, 34(8):131 – 140, 1999.

[6] P. Sanders and J. L. Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, pages 799 – 803, 2002.

[7] P. Sanders and J. L. Träff. Parallel prefix (scan) algorithms for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 49 – 57. Springer, 2006.

[8] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMP's. In *Supercomputing, SC '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (cd-rom)*, page 23, New York, New York, USA, 1999. ACM.

[9] L. A. Steffenel and G. Mounie. A framework for adaptive collective communications for heterogeneous hierarchical computing systems. *J. Comput. Syst. Sci.*, 74(6):1082–1093, January 2008.

[10] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 381 – 392, New York, NY, USA, 2001. ACM.

[11] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. International Parallel and Distributed Processing Symposium (IPDPS), April 2003.

[12] J. L. Träff. Improved MPI all-to-all communication on a giganet SMP cluster. In *Recent Advances in Parallel Virtual*

Machine and Message Passing Interface, pages 392 – 400. Springer, 2002.

[13] M.-S. Wu, R. A. Kendall, and K. Wright. Optimizing collective communications on SMP clusters. pages 399 – 407. International Conference on Parallel Processing, 2005.

[14] H. Zhu, D. Goodell, W. Groop, and R. Thakur. Hierarchical collectives in mpich2. In *Lecture Notes in Computer Science*, pages 325–326. Springer Berlin, 2009.