Optimizing Blocking and Nonblocking Reduction Operations for Multicore Systems: Hierarchical Design and Implementation

Manjunath Gorentla Venkata, Pavel Shamis, Rahul Sampath
Oak Ridge National Laboratory
Oak Ridge, TN, USA
manjugv,shamisp,sampathrs@ornl.gov

Richard L. Graham, Joshua S. Ladd
Mellanox Technologies
San Jose, CA, USA
joshual@mellanox.com

***Abstract*—**

Many scientific simulations, using the *Message Passing Interface* (MPI) programming model, are sensitive to the performance and scalability of reduction collective operations such as MPI_Allreduce and MPI_Reduce. These operations are the most widely used abstractions to perform mathematical operations over all processes that are part of the simulation. In this work, we propose a hierarchical design to implement the reduction operations on multicore systems. This design aims to improve the efficiency of reductions by 1) tailoring the algorithms and customizing the implementations for various communication mechanisms in the system 2) providing the ability to configure the depth of hierarchy to match the system architecture, and 3) providing the ability to independently progress each of this hierarchy. Using this design, we implement MPI_Allreduce and MPI_Reduce operations (and its nonblocking variants MPI_Iallreduce and MPI_Ireduce) for all message sizes, and evaluate on multiple architectures including InfiniBand and Cray XT5. We leverage and enhance our existing infrastructure, Cheetah, which is a framework for implementing hierarchical collective operations to implement these reductions.

The experimental results show that the Cheetah reduction operations outperform the production-grade MPI implementations such as Open MPI default, Cray MPI, and MVAPICH2, demonstrating its efficiency, flexibility and portability. On InfiniBand systems, with a microbenchmark, a 512-process Cheetah nonblocking *Allreduce* and *Reduce* achieves a speedup of 23x and 10x, respectively, compared to the default Open MPI reductions. The blocking variants of the reduction operations also show similar performance benefits. A 512-process nonblocking Cheetah *Allreduce* achieves a speedup of 3x, compared to the default MVAPICH2 *Allreduce* implementation. On a Cray XT5 system, a 6144-process Cheetah *Allreduce* outperforms the Cray MPI by 145%. The evaluation with an application kernel, *Conjugate Gradient* solver, shows that the Cheetah reductions speeds up total time to solution by 195%, demonstrating the potential benefits for scientific simulations.

Keywords*—*MPI, Collectives, HPC, Allreduce, Reduce

I. Introduction

MPI_Allreduce operation, as defined in the MPI standard, combines the data from all participants with an operation, and distributes the results of the operation to all participants. The size and type of data from all the participants is the same, and all participants receive the same results. In an MPI_Reduce operation, the root process collects the data from all processes and performs an operation. Only the root process receives the result. Hereafter, we refer to MPI_Allreduce and MPI_Reduce operations as *Allreduce* and *Reduce*, respectively.

Reduction operations, Allreduce and Reduce, are the commonly used collective operations from among the MPI collective operations, by scientific simulations. Other parallel programming models such as SHMEM and UPC, also provide similar collective operations. Research shows that over 40% of the time spent in MPI operations is spent in the reduction operations [10] [9]. Scientific applications and benchmarks such as LAMMPS, Fluent, OpenFOAM, and Paratec spend more than 50% of its collectives time in the MPI_Allreduce operation. Iterative solvers such as *Conjugate Gradient*, GMRES, and Newton, which are important components of many scientific simulations, use reductions one or more times in each iteration to compute dot products and/or norms.

Currently used algorithms and implementations for *Allreduce* and *Reduce* suffer from several performance drawbacks on multicore systems. These systems typically consist of tens of *Central Processing Unit* (CPU) cores on a node, network interface with bandwidth of tens of Giga bytes per second and latency of a few microseconds, and have multiple communication mechanisms - multiple cache levels, intra-node communication buses, and network interfaces - with varying performance characteristics. The multicore system architecture is ubiquitous in extreme scale systems. Also, these systems are widely used by scientific community for executing the scientific simulations [1]. Most existing *Allreduce* and *Reduce* implementations do not consider these performance variations in communication mechanisms in modern systems, and typically have a single implementation for all these different communication mechanisms resulting in far from optimal performance. In this paper, we optimize *Reduction* operations for these multicore systems by using a hierarchical design.

The hierarchical design and implementations aim to perform and scale better by tailoring the algorithm and customizing the implementation to various communication mechanisms in the system. Consider a two-level hierarchical *Allreduce*, for example, to understand the design. The *Allreduce* operation is implemented by combining *Reduction* primitive, *Allreduce* primitive and a *Broadcast* primitive (We use the term

978-1-4799-0898-1/13/$31.00 © 2013 IEEE

primitive for collective operation, when it is not necessarily MPI complaint). Figure 1 shows collective primitives in a two-level and 8-process *Allreduce* on two nodes. With the *Reduce* and *Broadcast* primitives optimized for intra-node, and an *Allreduce* optimized for inter-node, we expect this would translate to have better performance compared to a single implementation that is optimized for all communication mechanisms. *Reduce* similarly is implemented by combining various implementations of *Reduce* primitives. The hierarchical *Allreduce* and *Reduce* operations are implemented by enhancing our previously introduced Cheetah framework. The Cheetah reductions implemented in this paper, as we are aware, is a first implementation of *Reduction* operations that supports n-levels of hierarchy.

Besides hierarchical design, we extend commonly used algorithms, support nonblocking semantics and independent progress of hierarchies to take advantage of modern hardware characteristics, to arrive at an optimized implementation. Toward this end, we introduce *Recursive K-ing* algorithm for *Allreduce*, which has the flexibility to vary the tree width, or number of processes reducing the data, concurrently. For *Reduce* operation, we use a *N-ary* algorithm.

In summary, we make the following contributions in the paper:

- We provide multiple n-level hierarchical *Allreduce* and *Reduce* implementations optimized for multicore systems. It supports all message sizes and multiple architectures including *Shared-Memory* (SM), InfiniBand, and SeaStar.
- We provide both blocking and nonblocking implementation of the reductions.
- We demonstrate the use of *Recursive K-ing* algorithm for *Allreduce* operation. The algorithm is known to provide better support for high-radix systems.
- We enhance the Cheetah infrastructure to support more variety of collective operations including the reduction operations. Thus providing infrastructure to build n-level hierarchical collectives. All implementations and infrastructure is integrated into Open MPI.
- We systematically evaluate both *Allreduce* and *Reduce* on multiple architectures, and at a higher scale. The experimental results show that the Cheetah reduction operations outperform the state-of-the-art implementations by more a than order-of-magnitude.

The rest of the paper is organized as follows: Section II compares the approach in the paper with the previous approaches. Section III provides the details of the Cheetah infrastructure. Sections IV, V, and VI provide details of algorithms and implementations of the reduction operations. Section VII provides the evaluation details of these implementations with the microbenchmarks and an application kernel, *Conjugate Gradient* solver. Section IX concludes the paper.

II. Related Work

Given that the reduction operations are the most important class of collective operations, there is a huge body of research in this area. Traditionally, reduction operations were optimized

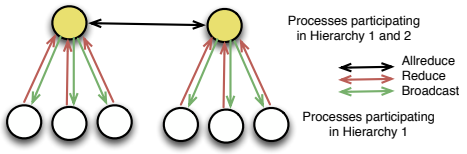

Fig. 1. Interaction between various primitives in a two-level *Allreduce*

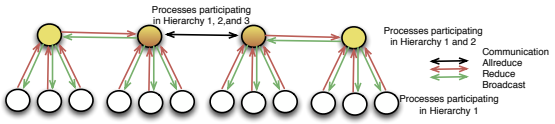

Fig. 2. Interaction between various primitives in a three-level *Allreduce*

for the topology of the systems. Open MPI's Tuned module [2] and MPICH2 are examples of this approach. They were designed for single core systems, and, as a consequence, result in sub-optimal performance on multicore systems. There have been several efforts to implement collectives as hierarchical collectives [12] [5] [6]. However, they treat the multicore node more as a SMP node, and typically support one-level of hierarchy. To achieve hierarchical implementation they use the approach, where a single process combines the data from all processes on a node, conducts the inter-node communication and broadcasts results to all local process. For systems with multiple CPU sockets, which is common in current and emerging extreme-scale systems, this approach will be very limiting. It does not accommodate systems with multiple intra-node data transfer paths with varying performance characteristics, resulting in increasing inter CPU socket communication, cache pollution, and, hence, lost performance.

Our work differs from other work in multiple ways. In this work, we provide a very generic way to implement hierarchical reductions that can be scaled to implement n-level hierarchical reductions. We also support configuring reductions with varying number of hierarchies to suit the system architecture and performance requirements. Each primitive associated with the hierarchy, at any level, can be progressed independently while the semantics permit. The mechanisms are generic enough to implement both blocking and nonblocking semantics, root-based (*Reduce*) and non-root based reductions (*Allreduce*), unlike this research [8]. Compared to [5], the concepts and mechanisms are portable across different architectures.

III. Hierarchical Design and Implementation

To arrive at the hierarchical implementation of reduction operations, we enhance our previously implemented Cheetah framework. An early version of this infrastructure was described in our previous paper [3].

A Cheetah collective operation is hierarchical, and it is composed of multiple independently progressing collective primitives, each optimized for a particular communication mechanism. Figure 1 shows a 8-process two-level hierarchical *Allreduce* distributed across two nodes. The processes on the same node are grouped into a group called *hierarchy 1*. A process in the group is selected as a leader, and is responsible for inter-hierarchy communication. The leader processes of each group is combined into a group called *hierarchy 2*. In this example, the communication among the processes in the *hierarchy 1* is intra-node, and the communication between the processes in *hierarchy 2* is inter-node. To realize an *Allreduce* operation, the processes in *hierarchy 1* execute Reduce primitive with the leader process as the root. Then, leaders with

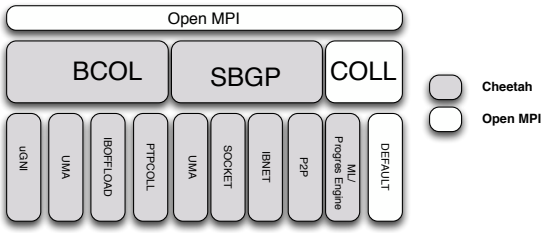

Fig. 3. Cheetah Frameworks and components within Open MPI.

reduction result (all processes in *hierarchy 2*) participate in an *Allreduce* primitive operation, and the results of *Allreduce* is broadcast to all processes in *hierarchy 1*. Extending this approach to three-level hierarchy is shown in Figure 2. A n-level hierarchical *Allreduce* is achieved with $n-1$ *Reduce* and *Broadcast* primitives, and one *Allreduce* primitive.

Figure 3 shows Cheetah's frameworks and its components to implement hierarchical reductions. It includes components to create hierarchies (or called as subgroups - SBGP), provide optimized primitives (BCOL), and *Messaging Layer* (ML), which acts as a controller. The ML provides mapping between primitives and MPI semantics, with the help of enhancements, *DAG Engine* and *Schedule*, it controls and progresses the collectives. The rest of the section describes these components in detail.

A. Subgrouping Component

The subgrouping component groups the processes in the job into multiple subgroups based on the communication mechanism shared among them; the hierarchy in Figure 1 is referred as subgroups here, and they are used interchangeably. Currently we support four subgrouping methods - *UMA*, *Socket*, *IBNET*, and *Point-to-Point* (P2P). *UMA* and *Socket* represents intra-node subgrouping methods, while *IBNET* and P2P represents the inter-node subgrouping methods. A group of processes sharing the CPU socket is grouped into *Socket* subgroup, and processes sharing the CPU node is grouped into *UMA* subgroup. A group of processes connected through Mellanox's InfiniBand HCA with *Core-Direct* is grouped into *IBNET* subgroup. The processes connected through none of these communication mechanisms, but other interconnects such as Gemini, SeaStar, Ethernet or any thing else is grouped into a P2P subgroup.

B. Basic Collective *(BCOL) Primitives*

The BCOL primitives provide a basic implementation of collective operations, which are not necessarily complaint to the MPI standard. The primitives are optimized implementations for a particular communication mechanism. We currently have optimized primitives for Gemini, *Core-Direct*, intra- and inter-CPU socket, and Ethernet.

C. Directed Acyclic Graph *(DAG) Collectives Launcher*

The DAG collectives launcher provides an DAG abstraction called *Schedule* to express the collective operations, and a mechanism, *DAG Engine* to control the order and execution of primitives. The *DAG Engine* provides the mechanisms that support nonblocking semantics and concurrent progress of multiple outstanding collectives.

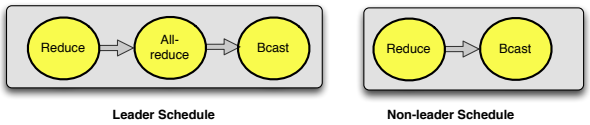

Fig. 4. *Schedule* for a two-level *Allreduce*

Schedule: The *Schedule* provides an abstraction to express combination of collective primitives, and its execution order. For example, the DAG in Figure 4 shows a *Schedule* for *Allreduce* in Figure 4. Each node in the graph represents a collective primitive. The incoming edges describes the list of primitives that have been completed before the collective primitive represented by the node is executed. The outgoing edges imposes a constraint on the target node. In the figure, the *Broadcast* node is required to wait until the *Allreduce* node is completed. Each process in the collective operation has its own *Schedule*. The figure shows the *Schedule* for a leader and non-leader process.

Three types of *Schedule*'s can express all collective operations in the MPI standard : Fixed, Sequential, and General.

Sequential: In this type of *Schedule*, the order of the primitives executed is typically fixed at the initialization of library, and not changed between invocations. The collective operations such as *Allreduce* require this *Schedule*. This will be explained in detail in IV.

Fixed: The order of the primitives are fixed during invocation for this type of *Schedule*, but they can vary between invocations. This schedule type is used to express the collective operations that is dependent on the root of the collective operation to determine the order. For example, the *Broadcast* collective operation requires the subgroup containing root to be executed first, and the *Reduce* operation requires the subgroup containing the root to execute last.

General: The order of primitives executed is not fixed in this type of *Schedule*. As a consequence, any primitive that can be progressed is executed. This is used by collective operations such as Barrier, as it does not impose dependencies between the primitives.

***DAG Engine*:** It controls the order and execution of collective primitives expressed by the *Schedule*; it acts as an interpreter of the *Schedule*. *DAG Engine* makes use of two list structures to progress and complete the primitives. When a collective operation is invoked, the *DAG Engine* adds each primitive in the *Schedule* to the progress list, progresses the primitive, and when the primitive is waiting for resources, it removes the collective primitive and adds it to the pending list. A collective primitive could be waiting on network resources, memory resources, or other processes to progress. The *DAG Engine* then progresses the primitive that can be progressed. Thus supporting the nonblocking semantics for the collective operation, and also enabling multiple outstanding primitives.

IV. Allreduce Algorithm and Implementation

This section provides details of algorithms used in *Allreduce*, and the hierarchical *Allreduce* implementation. Particularly, the implementation used for small messages.

Recursive K-ing Allreduce Algorithm

The *Recursive K-ing Allreduce* algorithm is similar to a commonly used *Reduction* algorithm, except that instead of two

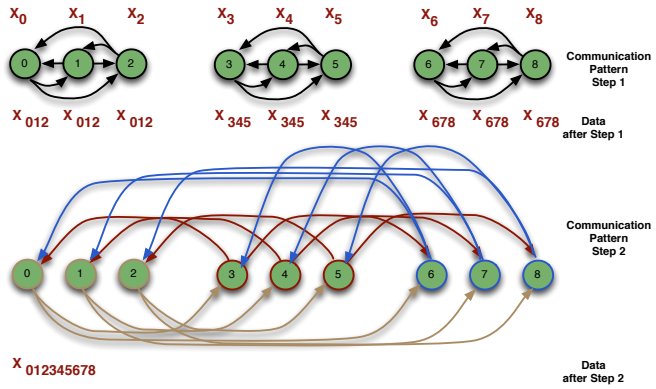

Fig. 5. Small Message Recursive K-ing for power k processes

process participating in a reduction in every step, k processes participate in the reduction operation and for $log_k(n)$ steps, instead of $log_2(n)$ step. The communication pattern of this algorithm is shown in Figure 5.

The figures shows an example of *Allreduce* with nine processes (ranks), and with the graph of degree three ($k = 3$). In the first step, rank X combines data with rank $(X + 1 \bmod 9)$ and rank $(X + 2 \bmod 9)$. In the second step, it combines data with rank $(X + 3 \bmod 9)$ and rank $(X + 2 * 3 \bmod 9)$.

If the number of processes in the *Allreduce* are not a power of k, then all processes above highest power of k (called extra process) is assigned an proxy process that is ranked within the power of k. The extra process reduces its data with the proxy process, and then reduces the data with other process similar to the Figure 5. Then, the proxy processes send the results back to the extra process.

Hierarchical *Allreduce* Implementation

A hierarchical *Allreduce* is implemented combining *Reduce*, *Allreduce*, and *Broadcast* primitives. How these primitives interact was shown in Figure 1. It is easy to extrapolate this to n level *Allreduce*. The n-level hierarchical *Allreduce* is a combination of $n - 1$ *Reduce* and *Broadcast* primitives, and one *Allreduce* primitive.

The execution of an hierarchical *Allreduce* involves building a *Schedule* (required to construct on first invocation or initialization of library), and executing the schedule. The schedule needs to be built only on the first invocation. It uses sequential *Schedule* as the *Reduce*, *Allreduce*, and *Broadcast* primitives have to executed in this order for *Allreduce* collective operation. For example shown in Figure 1, the leader process *Schedule* executes all three primitives (*Allreduce*, *Reduce*, and *Broadcast*) and non-leader process *Schedule* executes only two primitives (*Reduce* and *Broadcast*); the schedules are shown in Figure 4. For this architecture and configuration, *Reduce* and *Broadcast* are associated with *UMA* subgroup and use SM implementations, and the *Allreduce* primitive is associated with P2P subgroup and use inter-node implementation.

The inter-node *Allreduce* primitive is implemented using *Recursive K-ing* algorithm described in the previous section, the SM *Reduce* primitive is implemented using *N-ary* algorithm described in the next section, and the SM *Broadcast* primitive is implemented using a binary tree algorithm. Each of this primitive can independently configured (in this case the degree of the tree) to match communication mechanism's characteristics.

V. Reduce Algorithm and Implementation

This section provides details of *N-ary* algorithm used in *Reduce*, and the the hierarchical *Reduce* implementation used for small messages.

N-ary Algorithm

In the *N-ary* algorithm, a process is either a root, leaf, or internal node. The root waits for data from all its children, reduces the data and completes. The leaf sends the data to its parent, and completes the reduction operation. The internal node waits for the data from its children, reduces the data, and sends the result to its parent. The degree (number of children) of the *N-ary* tree is configurable, which is denoted by n.

Hierarchical implementation of Reduce

The n-level hierarchical *Reduce* is a combination of n *Reduce* primitives. Each reduce primitive is expressed over a subgroup. We currently have two different *Reduce* implementations, one implementation optimized for SM (*UMA* or *Socket*) and other optimized for inter-node (P2P subgroup).

The SM *Reduce* implementation uses a binary tree algorithm, and inter-node *Reduce* operation uses a *N-ary* algorithm. These, two reduce primitives suffices to express the reduction operation on most modern systems. For example, a system where each node has one CPU socket, and many CPU cores could use a configuration with one SM and one inter-node primitive. However a system where each system node has many CPU sockets, and CPU cores, (example *Smoky* system architecture is described in VII section) would use multiple SM primitives (and multiple *Socket* subgroups and one *UMA* subgroup) and one inter-node primitive (one P2P subgroup).

The hierarchical *Reduce* implementation uses a static *Schedule* to describe the collective operation. The subgroup which includes the root process is the last node in the *Schedule*.

VI. Large Message *Allreduce* and *Reduce* Algorithms and Implementations

Based on the size of the message, we used two different implementations. For small messages, we used the latency driven implementations described in Section IV and V. For large messages, we used the implementations described in this section. The message threshold where we make the switch is a runtime decision.

When a reduction operation either *Allreduce* and *Reduce* is invoked with large message, the large message is fragmented into multiple message fragments, and each fragmented message executes the small message reduction collective operation. Once, all fragments of the message invokes the small message reduction operation and completes, the large message reduction operation is completed and returned. The large message is fragmented respecting the datatype boundary to ensure the correct reduction result. To maximize bandwidth, we have multiple outstanding operations active at any given instance. The size of the fragment and the number of active invocations are controlled by a configurable runtime parameters.

Like small message implementations, the large message implementations are also hierarchical. The large message implementations support n levels of hierarchy. The large message

hierarchical *Allreduce* uses *Reduce*, *Allreduce*, and *Broadcast* primitives, while the large message hierarchical *Reduce* uses multiple *Reduce* primitives.

VII. Evaluation

In this section, we describe the evaluation of Cheetah reductions' performance characteristics using microbenchmarks and an application kernel, *Conjugate Gradient* solver. The Cheetah reductions are compared with Open MPI's default blocking and nonblocking [4] collectives, Cray MPI (vendor provided MPI implementation on the Cray systems), and MVAPICH2.

A. Experimental Setup

1) Hardware: We ran all experiments on the Oak Ridge Leadership Computing Facility's Jaguar, a Cray XT5 system, and Smoky. For each of these experiments, the MPI job was configured with one MPI process per CPU core. For all experiments, we used the *Smoky* system, except for comparison with the Cray MPI, for which, we used the Jaguar system.

Jaguar has 18,688 compute nodes, each one containing 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. Each AMD Opteron processor has six computing cores and three levels of cache memory. The compute nodes run Compute Node Linux micro-kernel.

Smoky is an 80 node test and development cluster. Each node contains four 2.2 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 *Megabyte* (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.

2) Benchmarks and Application Kernels: The microbenchmark used to measure the latency, simply invoked the reduction operations (either *Allreduce* or *Reduce*) in a tight loop, measured the overall time, and reported the average completion time of the reduction operation.

The application kernel used to evaluate the reduction operations solved the weak-scaling 1D Poision problem using the *Conjugate Gradient* algorithm, and reported the total time for completion. The 1D Poision problem was discretized using second-order finite differencing on a uniform grid.

B. Performance characteristics of Allreduce and Reduce implementations

In this experiment, we evaluated the latency characteristics of the blocking and nonblocking reductions, as the number of processes participating in the reduction operation increased from 16 to 512. For each problem size, we measured the completion time of the reduction for 10000 iterations and reported the average time. For all of these measurements, the Cheetah *Reduce* was configured with two-level hierarchy (UMA, P2P). The Cheetah reductions were compared with the default Open MPI reductions.

1) **Evaluation of Blocking and Nonblocking *Allreduce*:** The latency characteristics for a 8 byte blocking *Allreduce* is shown in Figure 6, and a nonblocking *Allreduce* is show in Figure 8. The 8 byte *Allreduce* operation uses the implementation described in Section IV. The latency characteristics of 1 MB blocking *Allreduce* is shown in Figure 7 and nonblocking *Allreduce* is shown in Figure 9. We present here only the latency characteristics of 8 byte and 1 MB reduction operations, as they were representative for all other small and large message sizes, respectively.

Latency of Blocking *Allreduce*: Figure 6 and 7 shows that the blocking Cheetah *Allreduce* operation outperforms the Open MPI's default *Allreduce*, both for small and large messages. The latency of a 512-process and 8 byte Cheetah *Allreduce* is 99.67 *usecs*, and the latency of a 512-process and 1 MB *Allreduce* operation is 43.2 *msec*. The Cheetah *Allreduce* outperforms the Open MPI operation by 23 times and 5 times, respectively.

Latency of Nonblocking *Allreduce*: Figure 8 and 9 shows that similar to the blocking *Allreduce*, the nonblocking Cheetah *Allreduce* outperforms the Open MPI's default, both for small and large messages. The latency of 512-process and 8 byte Cheetah *Allreduce* operation is 38.64 *usecs*, and the latency of 512-process and 1 MB *Allreduce* operations is 21.4 *msec*. It outperforms the Open MPI nonblocking *Allreduce* operation by 20 times and 12 times, respectively.

2) **Evaluation of Blocking and Nonblocking *Reduce*:** Figure 10 and 12 shows that the latency of 128 byte blocking and nonblocking *Reduce* respectively; the 8 byte *Reduce* operation used the implementation described in Section V. Figure 11 and 13 shows the latency of blocking and nonblocking 1 MB *Reduce* respectively; the 1 MB *Reduce* operation uses the implementation described in Section VI. We present here only the latency characteristics of 128 byte and 1 MB reduction

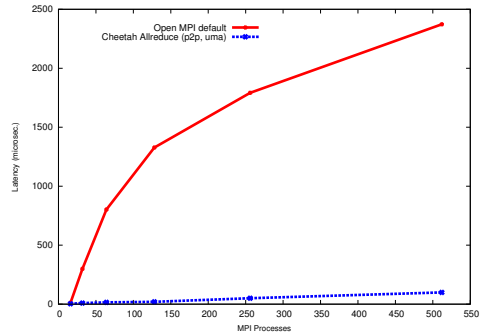

Fig. 6. Latency graph for a 8 byte blocking *Allreduce* operation

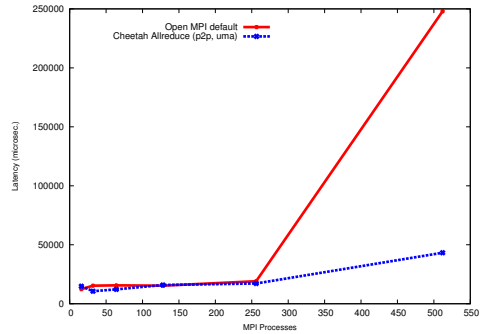

Fig. 7. Latency graph for a 1 MB blocking *Allreduce* operation

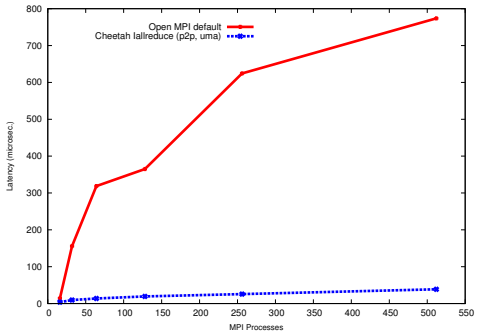

Fig. 8. Latency graph for a 8 byte nonblocking *Allreduce* operation

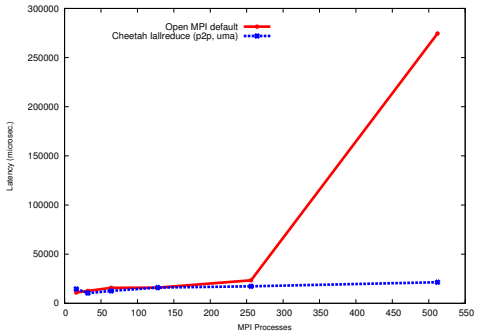

Fig. 9. Latency graph for a 1 MB nonblocking *Allreduce* operation

operations, as they were representative for all other small and large message sizes respectively.

Latency of Blocking Reduce: Figure 10 and 11 shows that the latency of blocking Cheetah *Reduce* operation outperforms the Open MPI's default, both for small and large messages. The latency of a 512-process and 128 byte blocking Cheetah *Reduce* is 8.86 *usecs* and the latency of 512-process and 1 MB Cheetah *Reduce* is 12.5 *msec*, outperforming the Open MPI operations by 10 times and 13 times, respectively.

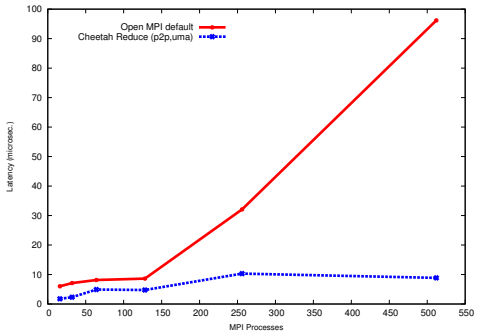

Fig. 10. Latency graph for a 128 byte blocking *Reduce* operation

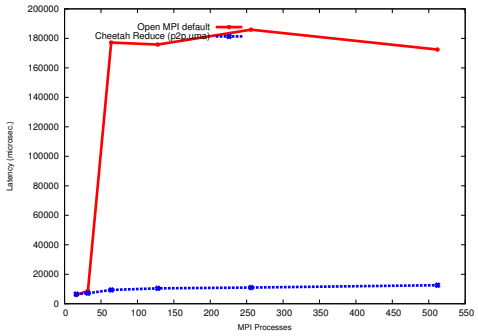

Fig. 11. Latency graph for a 1 MB blocking *Reduce* operation

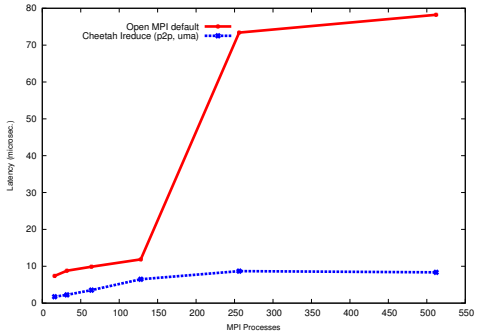

Fig. 12. Latency graph for a 128 byte nonblocking *Reduce* operation

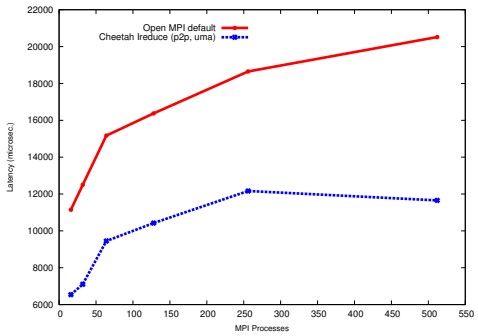

Fig. 13. Latency graph for a 128 byte nonblocking *Reduce* operation

Latency of Nonblocking Reduce: Figure 12 and 13 shows that the latency of nonblocking Cheetah *Reduce* operation for small and large message is lesser than the Open MPI's default. The latency of a 512-process and 128 byte Cheetah *Reduce* operation is 8.36 *usecs*, and 1 MB Cheetah *Reduce* is 11.6 *msec*. The Cheetah *Reduce* operations achieved a speedup of 9x and 70%, respectively, compared to the Open MPI's default.

C. Comparing Cheetah Allreduce *with Cray MPI and MAVPICH2*

We compared the performance of Cheetah *Allreduce* with Cray MPI on Cray XT5, and MVAPICH2 on *Smoky* to put the performance characteristics of Cheetah reductions in context with other state-of-the-art MPI implementations. The Cheetah reductions were configured with two-levels of hierarchy (UMA, P2P), and for Cray MPI and MVAPICH2 we used thier default configurations.

Figure 14 shows the latency of 6144 process Cheetah

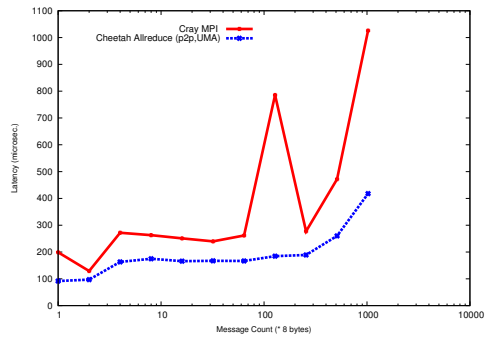

Fig. 14. Latency of 6144 process blocking *Allreduce* operation as message size is increased

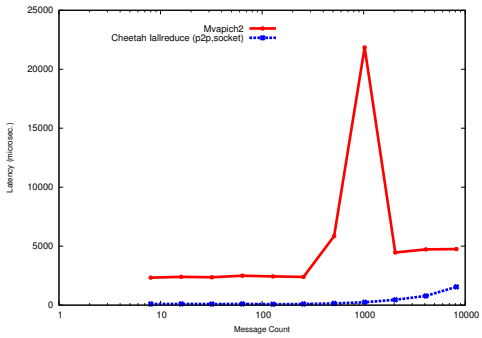

Fig. 15. Latency of 512 process nonblocking *Allreduce* operation as message size is increased

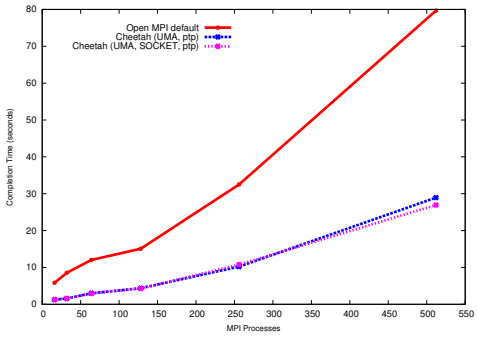

Fig. 16. Runtime of the *Conjugate Gradient* solver

Allreduce compared to Cray MPI on a Cray XT5 machine as we increase the message size. We observe that the latency of Cheetah *Allreduce* is consistently less than that of Cray MPI for all message sizes. For 8192 byte and 6144 process *Allreduce*, Cheetah *Allreduce* achieves a speedup of about 145% compared the default Cray MPI. A similar experiment with MAVPICH2 shows that for a 512-process 64 KB nonblocking *Allreduce*, Cheetah shows up to 3x speedup. The spike in the latency graph in Figure 15 is a result of switching of *Allreduce* implementations in the default MVAPICH2.

D. *Performance of a* Conjugate Gradient *Solver*

In this experiment, we measured the performance of the *Conjugate Gradient* solver. We ran the solver with Open MPI default and with Cheetah reductions, and measured the time to completion.

Figure 16 shows the completion time for up to 512 process problem. The results shows that *Conjugate Gradient* solver solves the problem two times faster while using Cheetah reductions compared to using Open MPI default reductions. The only difference in this experiment between the Cheetah run and the Open MPI default run is the reduction implementations employed. In the Cheetah run, the solver uses two-level Cheetah *Allreduce* and *Reduce*, and the rest of MPI operations are Open MPI default. In the Open MPI default run, the solver uses Open MPI's default reduction and MPI operations.

VIII. Analysis

To summarize the performance results, the Cheetah reductions, both *Allreduce* and *Reduce*, outperformed the default Open MPI reductions. The Cheetah *Allreduce* outperformed

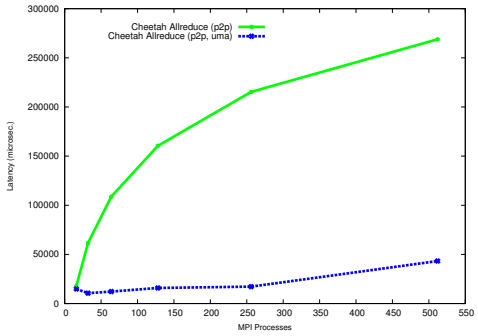

Fig. 17. Latency of one-level *Allreduce* compared to two-level *Allreduce*

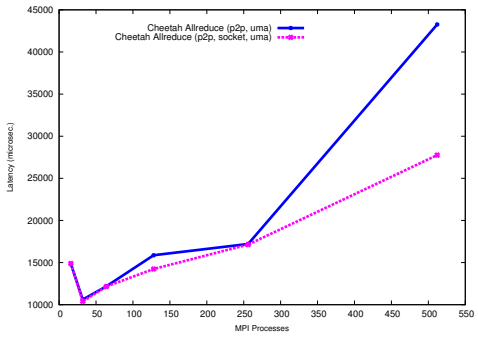

Fig. 18. Latency of a 1 MB blocking *Allreduce* operation as number of processes in the *Allreduce* is increased

the Cray MPI and MAVPICH2. [1] Small and large message blocking Cheetah *Allreduce* outperformed the Open MPI implementation, by 23 times and 5 times, respectively. Small and large message blocking Cheetah *Reduce* outperformed the Open MPI implementations by 10 times and 13 times, respectively. Similar performance gains are seen for the nonblocking reductions. The Cheetah *Allreduce* outperformed the Cray MPI for all message sizes for a 6144 processes problem size. For a 512-process problem, the Cheetah *Allreduce* achieved a speed up of 3x, when compared to the MVAPICH2. We observed similar performance gains for smaller problem size. In addition, we observed that Cheetah reductions improved the completion time of *Conjugate Gradient* solver, demonstrating the potential gains for many scientific simulations.

The performance gains can be contributed to hierarchical design and implementation. Hierarchical design achieves better performance as it tailors the algorithms for each communication hierarchy i.e., it has a different implementation of collective primitive for each communication hierarchy. In addition, it supports independent progress of these primitives. To further corroborate that hierarchical design contributes to the performance gains, we conducted an experiment where we changed the Cheetah *Allreduce* to have no hierarchy, and compared with Cheetah *Allreduce* with hierarchy.

Figure 17 shows the results of this experiment. We can observe that Cheetah with no hierarchy (only P2P) performs worse than Cheetah with hierarchy (*UMA*,P2P). We observed similar performance behavior for other reductions and configurations i.e., blocking and nonblocking *Reduce*, nonblocking *Allreduce*, and for both small and large message sizes.

We conducted another experiment on *Smoky* system, where

[1] The Cheetah *Reduce* also outperformed MAVPICH2, but for a lack of space, we did not include them.

we increased the number of hierarchies to three-levels (*Socket*, *UMA*, P2P) for a *Allreduce* operation and compared its performance to two-level (*UMA*, P2P) *Allreduce*. Figure 18 shows the latency of each configuration. We observe that the three-level configuration performs better than the two-level configuration. Also, we observe similar performance characteristics for *Conjugate Gradient* solver. However, we see negligible performance gains for some reductions configurations with three-level hierarchy setup, particularly small message *Reduce*. The performance gains can be explained as follows:

- When moving from one-level to two-level configuration, for this experimental setup, we are maximizing intra-node data transfers and minimizing inter-node data transfers. Moving from two-level to three-level, in addition to reducing inter-node transfers, we also minimize inter-CPU socket transfers maximizing intra-CPU socket transfers. This contributes to the performance gains.
- When moving from three-level to two-level configuration. The latency difference between a intra-socket and inter-socket data transfer is lesser compared to the latency difference between a intra-node and inter-node data transfer. This while contributes to overall better performance gains, it results in a lesser gain, as the overhead to manage the third hierarchy is counter acting.
- In addition to better managing the data transfers, the Cheetah's hierarchical reductions uses the network interface resources more efficiently. For example, it uses less connection resources on the InfiniBand HCA, which is known to have performance drawbacks [11] [7]. This can be observed as an exponential performance degradation in the Open MPI's 256-process *Reduce* operations (seen in Figure 7 and 9), and 512-process *Allreduce* operations (seen in Figure 7 and 9).

In conclusion, we see significant performance gain for going from one-level to two-level configuration, and additional smaller gain from going from two-level to three-level configuration. We anticipate that as the system architecture moves towards having multiple CPU sockets and many CPU cores on a node, building collectives with many hierarchies are advantageous. Also, it becomes essential to support varying number of hierarchies to accommodate the diverse architectures of multicore systems.

IX. Conclusion

In this work, we proposed a hierarchical design for implementing the MPI reduction operations on multicore systems, which is a commonly used architecture for current and emerging extreme-scale systems. We provided implementations of MPI_Allreduce and MPI_Reduce (and its nonblocking variants), called the Cheetah reductions, that are based on this approach. To quantify the performance benefits, we systematically evaluated the Cheetah reduction operations on Jaguar (Cray XT5) and Smoky (InfiniBand) for various problem sizes and message sizes. The empirical results demonstrated that the Cheetah reductions performs and scales better than the state-of-the-art MPI implementations - Open MPI, Cray MPI, and MVAPICH2. Furthermore, the evaluation with an application kernel also demonstrated significant improvements, demonstrating the potential benefits for scientific simulations.

Acknowledgment

This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Top 500 list. 1993-2012. http://www.top500.org.

[2] G. E. Fagg, J. Pjesivac-grbovic, G. Bosilca, J. J. Dongarra, and E. Jeannot. Flexible collective communication tuning architecture applied to open mpi. In *In 2006 Euro PVM/MPI*, 2006.

[3] R. Graham, M. G. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer. Cheetah: A Framework for Scalable Hierarchical Collective Operations. *CCGRID 2011*, 2011.

[4] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[5] K. C. Kandalla, U. M. Yang, J. Keasler, T. V. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. R. de Supinski, and D. K. Panda. Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers. In *IPDPS*, 2012.

[6] N. Karonis, B. R. D. Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. pages 377–384, 2000.

[7] M. J. Koop, T. Jones, and D. K. Panda. MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand. In *IPDPS*, pages 1–12, 2008.

[8] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra. HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *IPDPS*, 2012.

[9] R. Rabenseifner. Automatic Profiling of MPI Applications with Hardware Performance Counters. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 35–42, London, UK, UK, 1999. Springer-Verlag.

[10] R. Rabenseifner. Optimization of Collective Reduction Operations. In *International Conference on Computational Science 2004, Springer-Verlag LNCS 3036*, pages 1–9, 2004.

[11] G. M. Shipman, S. W. Poole, P. Shamis, and I. Rabinovitz. X-srq-improving scalability and performance of multi-core infiniband clusters. In *PVM/MPI*, pages 33–42, 2008.

[12] H. Zhu, D. Goodell, W. Gropp, and R. Thakur. Hierarchical collectives in mpich2. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 325–326, 2009.