

MPI-2 One-sided Usage and Implementation for Read Modify Write Operations: A Case Study with HPCC *

Gopalakrishnan Santhanaraman, Sundeep Narravula, Amith. R. Mamidala, and
Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210
{santhana,narravul,mamidala,panda}@cse.ohio-state.edu

Abstract. MPI-2's One-sided communication interface is being explored in scientific applications. One of the important operations in a one sided model is *read-modify-write*. MPI-2 semantics provide MPI_Put, MPI_Get and MPI_Accumulate operations which can be used to implement *read-modify-write* functionality. The different strategies yield varying performance benefits depending on the underlying one-sided implementation. We use HPCC Random Access benchmark which primarily uses *read-modify-write* operations as a case study for evaluating the different implementation strategies in this paper. Currently this benchmark is implemented based on MPI two-sided semantics. In this work we design and evaluate MPI-2 versions of the HPCC Random Access benchmark using one-sided operations. To improve the performance, we explore two different optimizations: (i) software based aggregation and (ii) hardware-based atomic operations. We implement aggregation techniques using MPI Accumulate with datatypes to improve the performance of one sided implementation. In order to study the impact of hardware capabilities provided by modern interconnects, we implement a prototype of Accumulate for MPI_Sum (Direct Accumulate) using InfiniBand's atomic fetch and add operation. We evaluate our different approaches on an InfiniBand cluster. The software based aggregation outperforms the basic one sided scheme without aggregation by a factor of 4.38. The hardware based scheme shows an improvement by a factor of 2.62 as compared to the basic one sided scheme. Our study shows that the software based aggregation performs the best. We also demonstrate the potential and scalability of the hardware based approach.

keywords: MPI-2, One-sided, HPCC , Accumulate, InfiniBand

1 Introduction

In the last decade MPI (Message Passing interface) [8] has evolved as the *de facto* parallel programming model in high performance computing scenarios. The MPI-2 standard provides a one-sided communication interface which is starting to be explored in scientific applications. One of the important operations in a one sided model is *read-modify-write*. Applications like [14] which is based on MPI-2 one-sided, predominantly use this operation. MPI-2 semantics provide MPI_Put, MPI_Get and MPI_Accumulate operations that can be used to implement the *read-modify-write* operations.

HPCC Benchmark suite is a set of tests that examine the performance of HPC architectures that stress different aspects of HPC systems involving memory and network in addition to computation [15]. HPCC Random Access benchmark is one of the benchmarks in this suite which

* This research is supported in part by Department of Energy's grant #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation's grants #CNS-0403342 and #CCF-0702675; grants from Intel, Sun Microsystems, Cisco Systems, and Linux Networx; Equipment donations from Intel, Mellanox, AMD, Apple, IBM, Microway, PathScale, Silverstorm and Sun Microsystems.

measures the rate of random updates to remote memory locations. Currently this benchmark is implemented based on MPI two-sided semantics. In this work we design different MPI-2 versions of the Random Access benchmark using the MPI-2 one-sided alternatives. We use the one-sided versions of the Random Access benchmark as a case study for studying different implementations of the *read-modify-write* operations and provide optimizations to improve the performance. The modified one-sided HPCC Random Access benchmarks are available on line for reference [3].

In this work we use two different techniques: (i) software-based aggregation and (ii) hardware-based atomic operations provided by InfiniBand to improve the performance. We evaluate and analyze the benefits of using software aggregation using datatypes with one-sided operations as well as the hardware based direct accumulate approach.

The rest of the paper is organized as follows. In Section 2, we provide an overview of InfiniBand, MVAPICH2 and HPCC Random Access benchmark. In Section 3, we describe different strategies for implementing one-sided versions of the HPCC benchmark. In Section 4, we discuss the two optimization techniques that we propose. In Section 5, we show performance evaluations of the various schemes. We present some discussion in Section 6. In section 7, we present related work. Conclusions and future work are presented in Section 8.

2 Background

In this section, we provide a brief background on InfiniBand, MVAPICH2 and the HPCC Random Access benchmark.

InfiniBand: The InfiniBand Architecture (IBA) [9] is an industry standard. It defines a switched network fabric for interconnecting processing nodes and I/O nodes. IBA supports both channel semantics and memory semantics. In channel semantics, send/receive operations are used for communication. In memory semantics, InfiniBand provides Remote Direct Memory Access (RDMA) operations, including RDMA write and RDMA read. InfiniBand also provides RDMA atomic operations namely `fetch_and_add` and `compare_and_swap`. The network interface card on the remote node guarantees the atomicity of these operations. The operations are performed on 64 bit values. In atomic `fetch_and_add` operation, the issuing process specifies the value that needs to be added and the address of the 64 bit location to which this value is to be added. Leveraging this atomic `fetch_and_add` mechanism is one of the focus of this paper.

MVAPICH2: MVAPICH2 is a popular implementation of MPI-2 over InfiniBand[1]. The implementation is based on MPICH2. MPICH2[2] supports MPI-1 as well as MPI-2 extensions including one-sided communication. In MVAPICH2 the one sided implementation of `MPI_Put` and `MPI_Get` uses the InfiniBand `RDMA_Write` and `RDMA_Read` services to provide high performance and scalability to the applications. The `MPI_Accumulate` is currently two sided based in the sense that the remote node is involved in performing the accumulate operation on that node. In this work we also provide a prototype of truly one-sided implementation of the `MPI_Accumulate` operation using the InfiniBand atomic `fetch_and_add` operation.

HPCC: The HPC Challenge (HPCC) benchmark suite has been funded by the DARPA High Productivity Computing Systems (HPCS) program to help define the performance boundaries of future Petascale computing systems [6]. HPCC is a suite of tests that examine the performance of high-end architectures using kernels with memory access patterns more challenging than those of the High Performance LINPACK (HPL) benchmark used in the Top500 list. The Random Access benchmark measures the rate of integer updates to random memory locations (GUPs). It uses xor operation to perform the updates on the remote node. The verification procedure allows 1% incorrect or skipped updates which allows loose concurrent memory update semantics on shared memory architecture. It allows optimization in terms of aggregating up to 1024 updates to improve the performance.

3 One sided HPCC Random Access benchmark: Design Alternatives

In this section we describe the different approaches taken to implement the one sided version of the HPCC Random Access benchmark. As described earlier, the random access benchmark measures the GUPs rating. The term randomly means that there is little relationship between one address to be updated and the next. An update is a read modify write operation on a table of 64-bit words. An address is generated, the value at that address read from memory, modified by an xor operation with a literal value and that new value is written back to memory. Currently the MPI version of the benchmark is based on two sided version. In this version the random address and value is generated and is sent to the remote node. The remote node receives this data and appropriately updates the memory location.

3.1 Design Issues

In this section we first describe the semantics and mechanisms offered by MPI-2 for designing one-sided applications. In a one-sided model, the sender can access the remote address space directly without an explicit receive posted by the remote node. The memory area on the target process that can be accessed by the origin process is called a Window. In this model we have the communication operations `MPI_Put`, `MPI_Get` and `MPI_Accumulate` and the synchronization calls to make sure that the issued one sided operations are complete. There are two types of synchronization: a) active in which the remote node is involved and b) passive in which the remote node is not involved in the synchronization. The active synchronization calls are collective on the entire group in case of `MPI_Fence` or a smaller group in case of `Start_Complete` and `Post_Wait` model. This could lead to some limitations when the number of synchronizations needed per process are different for different nodes. In passive synchronization the origin process issues `MPI_Lock` and `MPI_Unlock` call to indicate the beginning and end of the access epoch. Next we describe our approach taken in designing the one-sided versions of the HPCC Random Access benchmark. We map the table memory to the Window so that the one-sided versions can read and write directly to this memory.

3.2 HPCC Get-Modify-Put (HPCC_GMP)

In the first approach we call `MPI_Get` to get the data, perform the modification, then use `MPI_Put` to put the updated data to the remote location. As compared to the two sided versions there are no receive calls made on the remote node. Also the active synchronization model cannot be used since we cannot match the number of synchronization calls across all nodes. This is because the number of remote updates as well as the location of the remote updates for each node can vary randomly. Hence we use passive synchronization `MPI_Lock` and `MPI_Unlock` calls in this scheme. Further we need one set of Lock and Unlock calls to fetch the data, perform the modification, then another set of Lock and Unlock operations to put the data. The reason for this is the flexibility of MPI-2 semantics which allows `MPI_Get` to fetch the data in `Unlock`. Also the `MPI_Get` and `MPI_Put` can be reordered within an access epoch. We describe this approach in Fig. 1a and will henceforth refer to it as *HPCC_GMP*. This approach leads to a lot of network operations resulting in lower performance. Further the possibility of incorrect updates increases. This is due to the coherency issues that might arise because of parallel updates occurring simultaneously. To make sure that there are no incorrect updates, mutual exclusion (atomicity) has to be implemented on top of the existing approach which could lead to further degradation in performance.

3.3 HPCC Accumulate (HPCC_ACC)

Our next approach uses the `MPI_Accumulate` operation provided by MPI-2. MPI-2 semantics provide `MPI_Accumulate` which are basically atomic reductions. This non collective one-sided operation combines communication and computation in a single interface. It allows the

programmer to update atomically remote locations by combining the content of the local buffer with the remote memory buffer. This implementation calls `MPI_Accumulate` between `MPI_Lock` and `MPI_Unlock` synchronization calls. Using this approach shown in Fig. 1b, we do not have the issue of incorrect updates. Also as compared to our *HPCC_GMP*, the number of network operations is significantly reduced. Another approach is to use Accumulate with Active synchronization model using `Win_Fence`. This could be done by calling `Win_Fence` at the very beginning, performing all the updates using `MPI_Accumulate` and then call one `Win_Fence` at the very end. All the processes need to call two `Win_Fence` calls, one at the beginning and one at the end. However since MPI-2 semantics allows the actual data transfer to occur inside the synchronization call that closes the exposure epoch, all the accumulates could happen during the second `Win_Fence` call. Many MPI implementations actually make use of this flexibility. This violates the random benchmark rule that you could store only 1024 updates at the maximum before sending them. Hence we did not consider this approach.

4 Optimizations

In this section we describe two optimizations we propose in this paper to improve the performance of the one-sided version of HPCC Random Access benchmark.

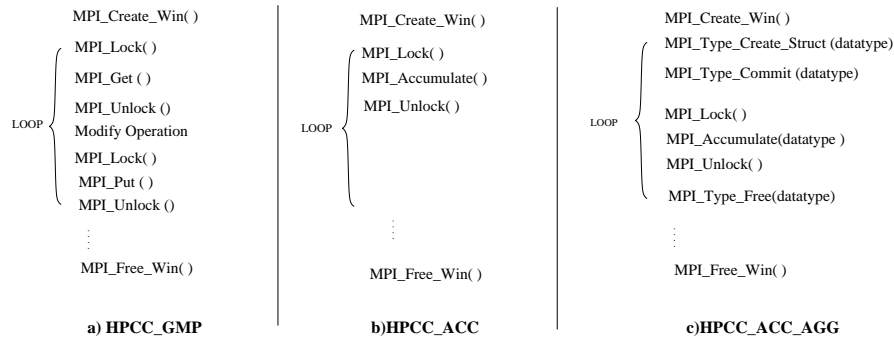


Fig. 1. Code snippets of one-sided versions of HPCC Random Access benchmark

4.1 HPCC Accumulate with Software Aggregation (HPCC_ACC_AGG)

In this technique we want to aggregate or pack a number of update operations together so that the overhead of sending as well as synchronization operations can be reduced. Using this approach, we aggregate a bunch of update operations before sending them as a single communication operation. The HPCC random access benchmark allows each processor to store up to 1024 updates before sending them out. The MPI-2 semantics provides *datatypes* feature that can be leveraged to achieve aggregation. For one-sided operations both the sender and destination datatypes need to be created. We create `MPI_Type_struct` sender and receiver datatypes to represent a bunch of updates in the following manner. The count holds the number of updates to be aggregated, the `block_lengths` are all one, the `displacement` array holds the remote address or local address respectively of each update and the MPI datatype of each entry is 64 bit unsigned integer. We then use the created datatypes to issue a single communication call as shown in Fig. 1c. Using this approach we expect to improve the performance since the number of network operations are minimized.

4.2 Hardware based Direct Accumulate (HPCC_DIRECT_ACC)

InfiniBand provides hardware atomic fetch and add operation that can be leveraged to optimize `MPI_Accumulate` operation for `MPI_SUM`. The Accumulate operations use the hardware fetch

and add operation that can provide good latency and scalability. One of the limitations of this approach is that we can only do single 64 bit accumulates with each fetch and add operation, i.e. aggregation is not possible. A benefit of using this approach is that since it is truly one-sided in nature, it provides more scope for overlap that can lead to improved performance. It is to be noted that this optimization is implemented in the underlying MVAPICH2 MPI library as a prototype and is transparent to the application writer.

5 Performance Evaluation

In this section, we evaluate the performance of the one-sided version of the HPCC benchmark for the different schemes. We present some micro-benchmark results to give the basic performance of different one-sided operations and show the potential of our proposed optimizations. The experimental testbed is x86 64 node cluster with 32 Opteron nodes and 32 Intel nodes. Each node has 4GB memory and equipped with PCI-Express interface and InfiniBand DDR network adapters (Mellanox InfiniHost III Ex HCA).

5.1 Basic performance of one-sided operations

In this section we show the performance of the basic one-sided operations MPI.Put, MPI.Get and MPI.Accumulate. Fig. 2a shows the small message latency for these operations. The latency for 8bytes for put and get are 5.68 and 11.03 usecs, respectively, whereas the accumulate latency is 7.06 usecs. Since get_modify_put implementation needs both get and put in addition to modify and synchronization operation, we expect this performance to be lower compared to the accumulate based approach.

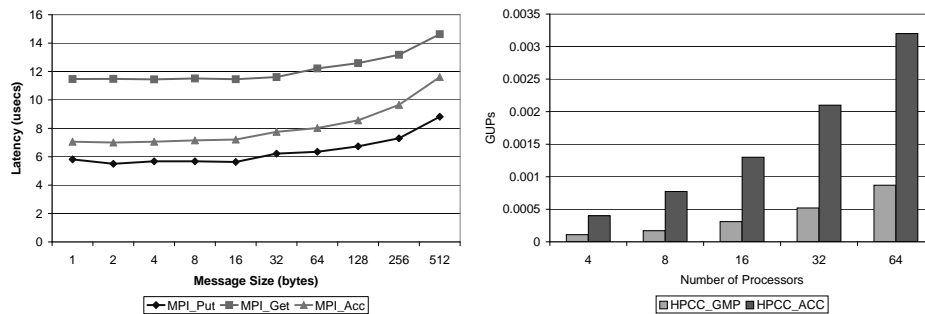


Fig. 2. Basic Performance (a) Micro-benchmarks and (b) Basic HPCC GUPs

5.2 HPCC one-sided benchmark performance with different schemes

In this section we evaluate the performance of the two different versions of the benchmark *HPCC_GMP* and *HPCC_ACC*. The results are shown in Fig. 2b. As expected the *HPCC_ACC* performs better than the *HPCC_GMP* because of the number of synchronization and communication operations in *HPCC_GMP*. The overhead of these additional network operations leads to lower performance of *HPCC_GMP*. This performance gap increases with increasing number of processors since the synchronization cost increases further for larger number of nodes. Hence we choose *HPCC_ACC* as our base case for further optimizations and evaluations.

5.3 Aggregation Benefits

To improve the performance of the Accumulate operation, we proposed aggregation using Accumulate with datatype. In this section we evaluate the performance benefits of using datatype at micro-benchmark level. In the basic version we do multiple accumulates corresponding

to the number of updates. In the aggregated version we create a datatype corresponding to the number of updates and perform a single accumulate operation with that datatype. Fig. 3a shows the results of our study. With increasing amounts of aggregation, the Accumulate with datatype outperforms the multiple accumulate schemes. With aggregation the cost of sending overhead and the synchronization overheads are limited to the number of aggregated operations. Next we compare the performance of *HPCC_ACC_AGG* with *HPCC_ACC* for 512 and 1024 aggregations. The results are shown in Fig. 3b. We observe a similar trend with the optimized *HPCC_ACC_AGG* performing better than the *HPCC_ACC* scheme. This result demonstrates the benefits that aggregation can provide.

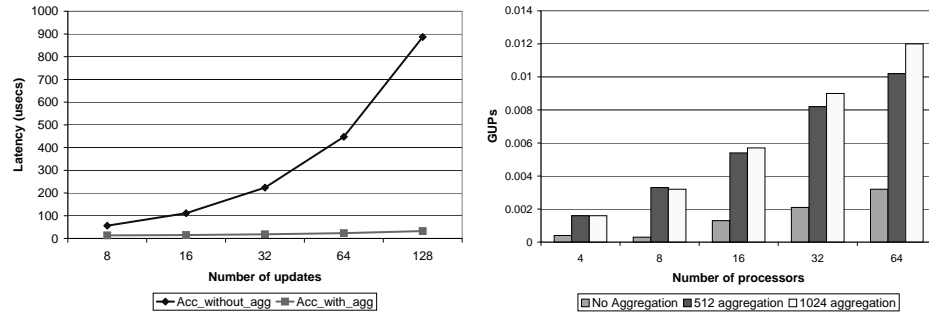


Fig. 3. Aggregation Performance Benefits (a) Basic Aggregation Micro-benchmarks and (b) HPCC with Aggregation

5.4 Hardware based Direct Accumulate

In this section we first study the benefits that could be achieved using the hardware based fetch and add operation to implement a read modify write operation at microbenchmark level (*DIRECT_ACC*). We compare its performance with the the schemes that uses Get Modify Put (*GMP*) approach and MPI Accumulate (*ACC*) approach. The MPI implementation allows optimizations that delays the actual lock and data transfer operation to happen during unlock. In this case measuring just the lock and unlock cost does not provide any additional insight. Hence we measure the latency that includes both data transfer and lock/unlock synchronization operation. Fig. 4a compares the basic performance of *GMP*, *ACC* and *DIRECT_ACC*. We note that for single updates of 64bit integer, the (*DIRECT_ACC*) scheme provides the lowest latency. This is because the existing MPI Accumulate implementation is inherently two sided whereas the Direct Accumulate implementation makes use of the truly one-sided hardware feature.

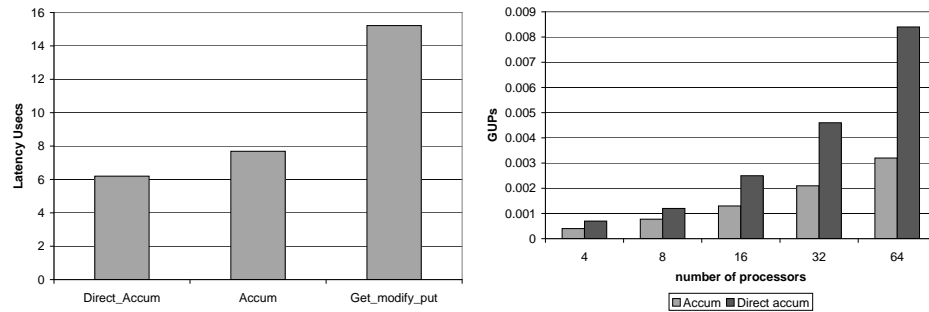


Fig. 4. Direct Accumulate Performance Benefits (a) Micro-benchmarks and (b) HPCC with Direct Accumulate

Next we try to understand the benefits that a hardware based Accumulate operation can provide to an application. To evaluate this we modify the *HPCC_ACC* benchmark to use the *MPI_SUM* operation instead of the *MPI_BXOR* operation and call this as *HPCC_ACC_MOD*. The verification phase is correspondingly modified. We then compare the *HPCC_ACC* which uses the existing *MPI_Accumulate* implementation in the *MVAPICH2* library with the modified *HPCC_ACC_MOD* which uses our Direct Accumulate prototype implementation. The results are shown in Fig. 4b. We observe that the Direct accumulate performs significantly better than the basic accumulate. Also the Direct Accumulate seems to scale very well with increasing number of processors. The reason for this is two-fold: 1) low software overhead and 2) true one-sided nature of the hardware based Direct Accumulate.

Finally we compare our two proposed techniques Direct Accumulate and software aggregation (Accumulate with datatype). The results are shown in Fig. 5. The software aggregation scheme beats the hardware based direct accumulate approach since currently the hardware fetch and add operation does not support aggregation. Also the gap between the two schemes seem to be narrowing with increasing nodes. This demonstrates the scalability of the hardware based operations and suggests the benefits of having aggregation in hardware as well.

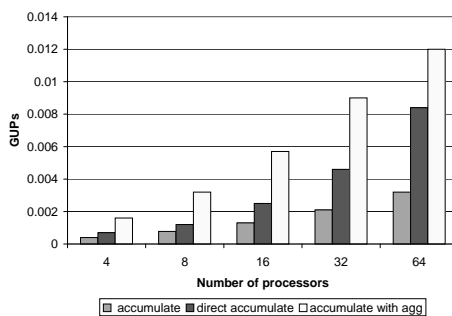


Fig. 5. Software Aggregation vs Hardware Direct Accumulate benefits

6 Discussion

Current implementations for HPCC Random Access benchmark are based on two-sided communication primitives. While the main objective of this paper is not to compare the designs based on one-sided and two-sided semantics, it is also important in this context to note that the current one-sided implementations are largely based on two-sided primitives in the MPI libraries and hence, such an evaluation is not as informative. InfiniBand's hardware fetch and add operation provides a design opportunity for a Direct Accumulate for *MPI_Sum* operation for a single 64 bit field. While we have demonstrated that both aggregation and direct hardware based accumulation has benefits, an aggregated direct accumulate is likely to yield much higher performance benefit. However it is clearly not possible to implement such a design with current InfiniBand's hardware. Also, it is to be noted that the hardware fetch and add operation currently only allows the implementation of accumulation of *MPI_Sum* for 64 bit fields and other operations need additional hardware support.

7 Related Work

There are several studies regarding implementing one sided communication in MPI-2. Some of the MPI-2 implementations which implement one sided communication are MPICH2 [2], WMPI [13], SUN-MPI [4], In [12, 10], the authors have used InfiniBand hardware features to optimize the performance of MPI-2 one sided operations. Other researchers [11] study the different approaches for implementing the one sided atomic reduction. The authors in [5] have

looked at utilizing the hardware atomic operations in Myrinet/GM to implement efficient synchronization operations. Recently several researchers have been looking at providing optimizations to the HPCC benchmark. In [7] the authors have suggested techniques for optimizing the Random access benchmark for Blue Gene clusters. In [16] the authors have evaluated UPC programming model on Cray machines using the HPCC benchmark suite.

8 Conclusions and Future work

In this paper we designed MPI-2 one sided versions of HPCC random access benchmark using `get_modify_put` and `MPIAccumulate` operations. We evaluated these two different approaches on a 64 node cluster. To improve the performance we explored two different techniques: a) software based aggregation and b) utilizing hardware atomic operations. We analyze the benefits and trade-offs of these two approaches. Our studies show that the software based aggregation performs the best. We also demonstrate the potential and scalability of the hardware based approach. For future work, we would like to explore optimizations and techniques that match application requirements especially in the context of more capable hardware. We also plan to contribute our one-sided versions of the benchmark to the HPCC benchmarking group. Further we would also like to evaluate the potential benefits with one sided applications which use these operations.

References

1. Network Based Computing Laboratory, MVAPICH2. <http://mvapich.cse.ohio-state.edu/>.
2. Argonne National Laboratory. MPICH2. <http://www-unix.mcs.anl.gov/mpi/mpich2/>.
3. MPI-2 One-Sided based HPCC Random Access benchmarks. <http://nowlab.cse.ohio-state.edu/projects/hpcc-one-sided/>.
4. Stephen Booth and Fernando Elson Mourao. Single Sided MPI Implementations for SUN MPI. In *Supercomputing*, 2000.
5. D. Buntinas, D. K. Panda, and W. Gropp. NIC-Based Atomic Remote Memory Operations in Myrinet/GM. Workshop on Novel Uses of System Area Networks (SAN-1), February 2002.
6. Jack Dongarra and Piotr Luszczek. overview of the hpc challenge benchmark suite. SPEC Benchmark Workshop, 2006.
7. Rahul Garg and Yogish Sabharwal. Optimizing the HPCC randomaccess benchmark on blue Gene/L Supercomputer. ACM SIGMETRICS Performance Evaluation Review, June 2006.
8. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.
9. InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.
10. W. Jiang, J.Liu, H. W. Jin, D. K. Panda, D. Buntinas, R.Thakur, and W.Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. EuroPVM/MPI, September 2004.
11. J.Nieplocha, V.Tipparaju, and E.Apra. An evaluation of two implementation strategies for optimizing one-sided atomic reduction. International Parallel and Distributed Processing Symposium, 2005.
12. J. Liu, W. Jiang, Hyun-Wook Jin, D. K. Panda, W. Gropp, and Rajeev Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. International Symposium on Cluster Computing and the Grid (CCGrid 04), April 2004.
13. Fernando Elson Mourao and J Gabriel Silva. Implementing MPI's One-Sided Communications for WMPI. In *EuroPVM/MPI*, September 1999.
14. R.J.Thacker, G.Pringle, H.M.P Couchman, and S.Booth. Hydra-mpi: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures. High Performance Computing Systems and Applications, 2003.
15. HPCC Benchmark Suite. <http://icl.cs.utk.edu/hpcc>.
16. T.El-Ghazawi, F.Cantonnet, Y.Yao, and J.Vetter. Evaluation of UPC on the Cray X1. Cray User Group meeting, 2006.