Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem*

Darius Buntinas, Guillaume Mercier, and William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
{buntinas, mercierg, gropp}@mcs.anl.gov

Abstract. This paper presents the implementation of MPICH2 over the Nemesis communication subsystem and the evaluation of its shared-memory performance. We describe design issues as well as some of the optimization techniques we employed. We conducted a performance evaluation over shared memory using microbenchmarks as well as application benchmarks. The evaluation shows that MPICH2 Nemesis has very low communication overhead, making it suitable for smaller-grained applications.

1 Introduction

The Message Passing Interface (MPI) standard has been designed to enhance portability in parallel applications, as well as to bridge the gap between the performance offered by a parallel architecture and the actual performance delivered to the application. The level of achievable performance depends, however, on the implementation. Two critical areas determine the overall performance level of an MPI implementation. The first area is the low-level communication layer that the upper layers of an MPI implementation can use as foundations. The second area covers the communication progress and management. We designed an efficient communication subsystem, called Nemesis, that features very low overhead and is therefore suitable to serve as a basis for the MPICH2 software [1], an open source implementation of MPI.

The design and implementation of the Nemesis communication subsystem has been previously presented in [2]. In this paper, we describe how we ported MPICH2 over Nemesis and show the performance benefits of MPICH2 Nemesis. We also explain the improvements that have been made in the MPICH2 communication progress engine. The resulting MPICH2 software stack yields a very low latency and high bandwidth and compares favorably with competing software.

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38, and by a grant of computing time from the Ohio Supercomputer Center.

The implementation also allows us to better assess both the overhead and the performance of MPI.

Section 2 gives an overview of the Nemesis communication subsystem. Section 3 describes how this communication subsystem has been integrated in MPICH2 as a new CH3 channel. We detail how we implemented several important features of the MPI2 standard. The various optimizations that MPICH2 gained are also explained. Section 4 presents a performance evaluation using shared-memory communication; in particular, we compare our implementation with the MPICH2 shm channel and other MPI implementations. Section 5 concludes this paper and discusses future work.

2 Overview of the Nemesis Communication Subsystem

In this section, we briefly describe the Nemesis communication subsystem. See [2] for a complete description of the design and implementation.

The Nemesis communication subsystem was designed to be a scalable, high-performance, shared-memory, multinetwork communication subsystem for MPICH2. The goals for our design, in order of priority, were scalability, high-performance intranode communication, high-performance internode communication, and multi-network internode communication. The implication of ranking the goals this way is that we strive to minimize the overhead for intranode communication, even if this comes at some penalty for internode communication.

To achieve the goals of high scalability and low intranode overhead, we designed Nemesis using lock-free queues in shared memory. Thus, each process needs only one receive queue, onto which other processes on the same node can enqueue messages without the overhead of acquiring a lock. Alternative designs would be to use a pair of receive queues per pair of processes or to use a single queue with a lock. On a large SMP, neither would be scalable, because of the $\mathcal{O}(N^2)$ number of queues needed or the contention on the lock, nor would they be efficient, because of the overhead of polling multiple queues or the overhead of acquiring and releasing a lock.

For internode communication, when a message is received from the network, a polling function for that *network module* enqueues the message onto the process's receive queue. A network module has a send queue onto which messages to be sent are enqueued. The send queue is analogous to a process's lock-free receive queue in that, when a process sends a message, it will enqueue the message onto the appropriate queue, whether it is a queue for another process on the same node, or a send queue for a network module. This simplifies the critical path when sending a message: No special action is taken when sending a message to a process on a remote node versus a process on the local node. Multiple networks can be supported by implementing additional network modules. Our current implementation supports internode communication over sockets and Myricom's GM message-passing system [3].

After analyzing our initial design, we applied several optimizations. To reduce latency, we optimized the placement of the receive queue *head* and *tail* pointers and added a *shadow head* pointer to reduce L2 cache misses. We also gathered variables that are often used together in the same cache line to reduce the number

of L1 cache misses in the critical path. For small SMP nodes, we used a *fastbox* mechanism to bypass the queues. A pair of buffers is allocated between each pair of processes. When sending a message, a process can bypass the queue by copying the message into the fastbox, if it is free, and setting a flag indicating a message is waiting. The receiving process then copies the message out of the fastbox and resets the flag. If the fastbox is full when a process is sending a message, the regular queue mechanism is used. This mechanism would not scale well for large SMPs and is used only for SMPs with a small number of processors. To improve bandwidth, we implemented architecture-specific memory copy functions. For ia32 and x86_64 architectures the memory copy function uses nontemporal store operations that bypass the cache. More details on these optimizations can be found in [2].

3 Integration into MPICH2

The communication portion of MPICH2 is implemented in several layers, as shown in Figure 1, and provides two ways to port MPICH2 to a communication subsystem. The ADI3 layer presents the MPI interface to the application layer above it, and the ADI3 interface to the device layer below it. MPICH2 can be ported to a new communication subsystem by implementing a device.

The figure shows the device for the Quadrics network. The figure also shows the CH3 device. The CH3 device presents the CH3 interface to the layer below it, and provides another way for MPICH2 to be ported to a new communication subsystem: by implementing a *channel*. This interface has fewer functions than the ADI3 interface, making it significantly simpler to implement. Because the interface is simpler, however, it may not be able to take advantage of certain features provided by the communication subsystem, such as RDMA or collective operations.

We chose to port MPICH2 over Nemesis by implementing a CH3 channel. While our intent is to eventually implement an ADI3 device for Nemesis, implementing a CH3 channel allowed us to rapidly create a prototype and evaluate the implementation of Nemesis. We did, however, modify the CH3 layer in order to allow certain optimizations of the Nemesis channel. In the rest of this section we describe the basic design of the Nemesis channel and key optimizations.

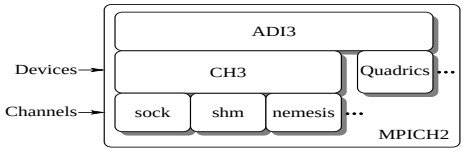


Fig. 1. Software layers of MPICH2

3.1 Basic Design of the Nemesis Channel

To send a message, the CH3 layer calls a send function implemented by the channel, passing in a pointer to the message header a description of the data

to be sent and a pointer to an MPI *request* object. The description of the data consists of an array of pointers and lengths (i.e., an *IOV*) that can be used to describe noncontiguous data. The Nemesis channel copies the header and data into a Nemesis receive queue element, called a *cell*, and fills in a short Nemesis header, then enqueues it on the appropriate receive queue or fastbox, or sends it over the network to the appropriate remote node. If the CH3 message is larger than a cell, multiple cells can be used, since the cells are delivered in FIFO order.

If not enough free cells are available to send an entire message, the IOV describing the unsent data is saved in the request, which is then enqueued onto a pending-send queue. When free cells are available, the messages on the pending-send queue are sent out. When all the data described by the IOV has been sent, the channel makes an up-call to CH3 to see whether there is more data to be sent. If there is, the IOV is reloaded; otherwise the request is marked as complete.

To receive a message, the Nemesis channel polls the receive queue and fastboxes. In order to reduce the overhead of unnecessarily polling too many fastboxes, the Nemesis channel polls only *active* fastboxes, which are the fastboxes of processes for which this process has posted a receive. Because fastboxes introduce a second path for messages between two processes, sequence numbers are used to maintain the order of messages.

When a cell is found, either in the receive queue or the fastbox, and there are no pending receives for that source process, the channel makes an up-call to CH3 with a pointer to the message header. If there is data to receive, CH3 will return an IOV along with a pointer to a request. The channel then copies the data from the cell to the user buffer described by the IOV. If the IOV describes more data than is contained in the cell, the IOV for the unreceived data is saved in the request, and the request is saved as a pending-receive corresponding to the process that sent the message. When the next cell from that process is received, the channel gets the saved request, and the new data is copied from the cell to the user buffer described by the IOV in the request. When all of the data described by the IOV has been received, the channel makes an up-call to CH3 to either reload the IOV, if there is more data to receive, or to mark the request as complete.

Because cells are allocated in shared memory, they are a limited resource. Hence, it is important to process a cell and copy out its data as soon as possible, so that it can be freed. This means that an unexpected message should be copied out of its cells and into a temporary buffer, as opposed to leaving the data in the cells until the receive has been posted. Unexpected messages are handled by the CH3 layer in just this way. If an unexpected message is received, CH3 creates a new request and passes back an IOV pointing to a newly allocated temporary buffer. So, the channel takes the same action whether the received message is unexpected or not. The message is copied out of the temporary buffer into the user buffer once a receive matching the message has been posted.

3.2 Large Message Transfer Using Rendezvous

While the shared-memory queue is very efficient for transferring small- to medium-sized messages, transferring large amounts of data through the queue may not

be the most efficient method. High-performance networks have RDMA capabilities where data can be transferred directly from the user's source buffer on one node to the user's destination buffer on another node, avoiding the data copies associated with using the queue. Some shared-memory machines, such as the SGI Altix, have similar mechanisms for processes on the same node. Even without special mechanisms, using a queue may not be the most efficient method of transferring large amounts of data between processes on the same node [4].

To support various mechanisms for transferring large messages, we defined the *Large Message Transfer* (LMT) interface and added it to CH3. Avoiding the queue can not only improve the bandwidth of the transfer but also reduce the impact on the application's data in the cache [4].

CH3 uses a rendezvous protocol when sending large messages, which ensures that a matching receive has been posted before the message data is sent. The rendezvous protocol is used primarily to avoid having to buffer the message if a matching receive has not been posted. We designed the LMT interface to be used together with the rendezvous protocol; the interface allows the channel to piggyback information on the CH3 rendezvous messages. The channel implements seven LMT functions, which are called by CH3.

For shared-memory communication, using the LMT interface, a shared-memory region is allocated and attached to by the sending and receiving processes. Then, using a double-buffering mechanism, the sending process copies the data into the shared-memory region while the receiving process copies it out. Because we used a memory copy function that uses nontemporal store operations, not only does this result in a high bandwidth transfer, but it has a very low impact on the application's data in the cache of the receiving process. The LMT optimization improves bandwidth for intranode communication by about 130 MiBps for large messages.

We have also used the LMT interface for the GM network module, which allows the use of RDMA operations. In the socket network module, we also used the LMT interface so that `read()` and `write()` operations can be issued to directly access the application's buffers, rather than copying the data through a cell.

3.3 Bypassing the Posted Receive Queue

We performed another optimization to improve the latency of small messages by bypassing the CH3 posted receive queue in certain cases. In the current implementation of CH3 when a receive is posted by the application, CH3 first searches the unexpected message queue to see whether it has already received a matching message. If a matching message is not found, the request is posted on the posted receive queue. CH3 then calls the progress engine to check for incoming messages. When a new message is received, CH3 looks for a matching receive request by searching the posted receive queue and enqueues the message in the unexpected queue if the message is not found.

Notice that if a receive is posted for which there is no matching message in the unexpected message queue, and the matching message is waiting to be received on the Nemesis receive queue or network, the receive request is queued on the posted receive queue, only to be matched and dequeued in the next step when the

progress engine is called and the matching message received. Our optimization implements a new function to call the progress engine with a receive request. As messages are received from the Nemesis receive queue they are checked for a match with the receive request. Only when no matching messages are found on the receive queue, is the request enqueued onto the posted receive queue. Note that if there already is a request on the posted receive queue that can possibly match the same message as the new receive request, we cannot use the optimization and, instead, add the new request to the receive queue as in the original implementation. This optimization reduced latency by about 18%, or 62 ns.

4 Performance Evaluation of MPICH2 over Nemesis

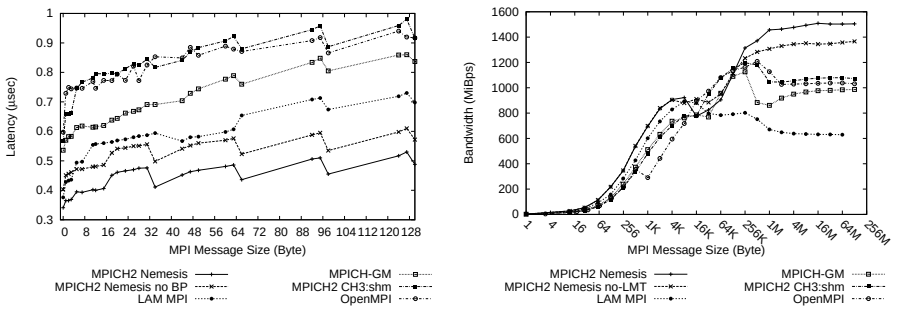
In this section we evaluate the shared-memory performance of our implementation of MPICH2 over the Nemesis communication subsystem. First we present a microbenchmark evaluation on a 2 GHz dual-processor dual-core Opteron 280 machine with 2 GiB of memory. Then we present application benchmarks on an SGI Altix 350 machine with 16 1.4 GHz Itanium 2 processors and 32 GiB of memory. We configured MPICH2 with the `--enable-fast` option that disables error checking and configured OpenMPI to disable error checking and support for heterogeneous clusters, which should improve the performance for those implementations. All implementations were compiled using `-O3` optimization.

4.1 Latency and Bandwidth

We compare our implementation to LAM/MPI [5] version 7.1.2, OpenMPI [6] version 1.1, MPICH-GM [7] version 1.2.6..14b, and MPICH2 version 1.0.3 configured with the CH3 *shm* channel that communicates by using shared memory. All these MPI implementation use shared-memory intranode communication. Except where noted, the results for MPICH2 Nemesis have both the LMT and posted receive queue bypass optimizations applied. We measured latency and bandwidth using Netpipe [8]. Figure 2 shows these results.

The latency graph in Figure 2(a) shows two data series for MPICH2 Nemesis. The results shown by the data series labeled "MPICH2 Nemesis no BP" were taken without the posted receive queue bypass optimization. This optimization improves latency by about 62 ns, resulting in a zero-byte latency of 341 ns. With the optimizations applied, MPICH2 Nemesis has lower latency than the other MPI implementations. Even up to 128 bytes, the MPICH2 Nemesis latency is just over 500 ns.

Figure 2(b) shows the bandwidth comparison. Nemesis uses an optimized memory copy routine that uses nontemporal store operations. Using the nontemporal copy routine results in dramatically higher bandwidth for MPICH2 Nemesis compared to the other MPI implementations. The results shown by the data series labeled "MPICH2 Nemesis no-LMT" were taken without applying the LMT optimization to MPICH2 Nemesis. The LMT optimization improves bandwidth by about 130 MiBps for large messages, resulting in a peak bandwidth of over 1,500 MiBps. Notice that for MPICH2 Nemesis, at 16 KiB the bandwidth of the non-LMT implementation is a little higher than the implementation with LMT. The reason is that at 16 KiB, the communication protocol



(a) Latency (b) Bandwidth

Fig. 2. Shared-memory performance of MPI implementations

switches from eager to rendezvous and additional setup is performed for LMT. The figure shows that MPICH2 Nemesis has higher bandwidth than the other MPI implementations except for messages between about 16KiB and 256KiB. We intend to perform additional tuning to improve the medium-sized message bandwidth and find the optimal message size for the crossover from eager to rendezvous protocol.

4.2 Instruction Count

One of the goals of our implementation is to streamline the critical path. One way of measuring our success is by counting the number of instructions required to send or receive a message. Using the PAPI[9] performance counter interface, we measured the instruction count for send and receive eight-byte messages. When measuring the instruction count for the receive operations, we wanted to avoid counting instructions performed polling while waiting for the message to arrive because the waiting time can vary quite a bit. To do this we added a delay equal to the round trip time before starting to count instructions and performing the receive. This ensured that the incoming message had arrived and was waiting at the receive queue when `MPI_Recv` was called. The table in Figure 3 shows these results. All MPI implementations were compiled with the -O3 optimization level, except for MPICH-GM, where the unoptimized code had fewer instructions.

The row labeled "MPICH2 Nemesis no BP" shows the instruction counts when the posted receive queue bypass optimization was not applied. The results show that this optimization reduces the combined send and receive instruction count by almost half. With the optimization, the combined instruction count for MPICH2 Nemesis is less than 22% that of OpenMPI, less than 50% that of MPICH2 CH3:shm and MPICH-GM, and 55% that of LAM MPI. The instruction counts show that the critical path in our implementation is already quite efficient, however, we believe that we still can further streamline the critical path and improve cache utilization which will reduce overall latency for small messages.

4.3 The Halo Benchmark

One of the benchmarks we used to predict the application performance of MPICH2 Nemesis was the Halo benchmark [10]. This benchmark simulates a nearest

MPI Implementation	`MPI_Send`	`MPI_Recv`	Total
OpenMPI	550	1,745	2,295
MPICH-GM	455	617	1,072
LAM MPI	436	472	908
MPICH2 CH3:shm	311	748	1,059
MPICH2 Nemesis no BP	241	712	952
MPICH2 Nemesis	**241**	**259**	**500**

Fig. 3. Instruction count for sending and receiving a eight-byte message.

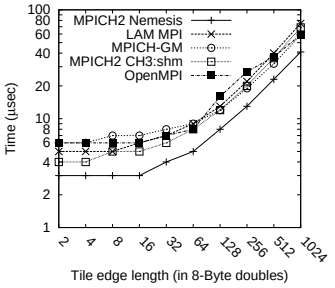


Fig. 4. Results of the Halo benchmark using four processes

neighbor exchange of a 1 to 2 row and column "halo" from a 2D array. The authors of the Halo benchmark state that performance of the Halo benchmark correlates well with the performance of their layered ocean model application. We ran the benchmark on the Opteron machine using four processes.

The Halo benchmark performs the halo exchanges by using several different algorithms. The results in Figure 4 show the results for the algorithm that performed best for each MPI implementation. The algorithm which used `MPI_SendRecv()` performed best in MPICH2 Nemesis, MPICH-GM and OpenMPI. In MPICH2 CH3:shm, the algorithm using `MPI_ISend()` and `MPI_IRecv()` performed best. In LAM MPI, the best performance was seen when using the algorithm that used persistent sends and receives, where the receives are posted before the send operations are called. In the figure, we see that MPICH2 Nemesis performs considerably better than the other implementations for all tile sizes. Of the others, MPICH2 CH3:shm performs better than LAM MPI, MPICH-GM, and OpenMPI for small tile sizes. For larger tile sizes MPICH-GM performs better than MPICH2 CH3:shm, LAM MPI, and OpenMPI. The performance of this benchmark is dominated by latency for small tile sizes and by bandwidth for large tile sizes. The factor of improvement for MPICH2 Nemesis over the other implementations ranges from 1.5 to 2.6. This suggests that MPICH2 Nemesis should perform well on applications that are sensitive to latency or need high bandwidth.

4.4 The NAS Benchmarks

We evaluated the application-level performance of MPICH2 Nemesis using the NAS benchmarks [11]. We wanted to evaluate how the low latency and high bandwidth of MPICH2 Nemesis can benefit the parallel speedup of applications. To emphasize the communication cost over the computation time, we used smaller problem sizes, specifically, the class A problem size with the CG, MG, FT, SP, BT, and LU benchmarks and the class B problem size for the IS benchmark. For the IS benchmark the class A problem size was too small for 8 and 16 processes and resulted in too much variation in the results. We decided not to use the EP benchmark because there was very little communication.

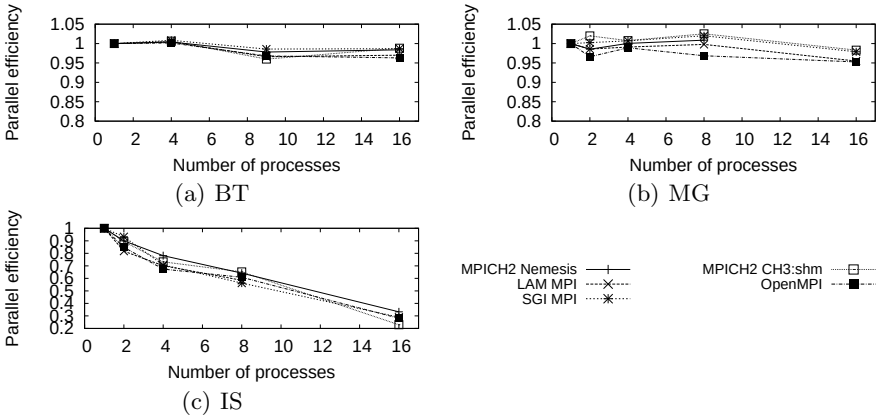


Fig. 5. Parallel efficiency for the NAS benchmarks

To get results for a larger number of processes, we ran the benchmarks on a 16-processor SGI Altix at the Ohio Supercomputer Center (OSC). On that machine MPICH-GM was not available; instead, we evaluated the SGI MPI implementation. The Altix machine has features to allow one process to directly access another process's address space, which can allow for very efficient large message transfers. However, these features were not enabled on the OSC machine. It is not clear how much of an impact the lack of these features has on the performance of SGI MPI.

In our evaluation, all of the MPI implementations performed similarly. Figure 5 shows the parallel efficiency for the class A BT and MG and class B is benchmarks. We omit the graphs for the other results because of space limitations. We see that for the BT and MG benchmarks the parallel efficiency for all implementations is better than 0.95. For the IS benchmark, which has a higher communication to computation ratio than the other benchmarks [12], we see that the parallel efficiency decreases considerably with the number of processes. Here too, we see that all of the MPI implementations perform similarly. The parallel efficiency for any individual implementation differs less than 10% from the average for up to 8 processes, and less than 20% from the average for 16 processes.

5 Discussion and Future Work

In this paper we have presented our new implementation of MPICH2 over the Nemesis communication subsystem. We evaluated the shared-memory communication of our implementation on a 4-core Opteron machine using microbenchmarks. Our implementation achieved a zero-byte latency of 341 ns and a 128-byte latency of just over 500 ns. The peak bandwidth of our implementation was over 1,500 MiBps. We also measured the number of instructions required to send and receive MPI messages. MPICH2 Nemesis uses only 500 instructions to send and receive an eight-byte messages. To evaluate application-level performance, we used the Halo benchmark, which favors low-latency and high-bandwidth MPI

implementations, and saw a factor of improvement from 1.5 to 2.6 compared to the other implementations we evaluated. Our evaluation using the NAS benchmarks on a 16-processor Altix machine did not show large differences in parallel efficiency between the different MPI implementations. These results show that MPICH2 Nemesis has an efficient implementation of shared-memory communication, which achieves low latency and high bandwidth. Moreover, the results indicate that MPICH2 Nemesis would be especially suitable for smaller-grained applications which are sensitive to latency and bandwidth.

Future work on MPICH2 Nemesis is to implement Nemesis as a full ADI3 device, which should further improve performance. We also intend to implement optimized collective communication operations that take advantage of shared memory, as well as collective operation primitives provided by network interfaces.

References

1. Argonne National Laboratory: MPICH2. (`http://www.mcs.anl.gov/mpi/mpich2`)
2. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem. In: Proceedings of International Symposium on Cluster Computing and the Grid 2006 (CCGRID '06). (2006)
3. Brown, G.: The GM message-passing system. `http://www.myri.com/news/02512/slides/Brown_gm.pdf` (2002) Presented at the Myrinet User's Group Conference (MUG-2002).
4. Buntinas, D., Mercier, G., Gropp, W.: Data transfers between processes in an SMP system: Performance study and application to MPI. In: Proceedings of the 35th International Conference on Parallel Processing (ICPP 2006). (2006) To appear. Available at `http://www-unix.mcs.anl.gov/~buntinas/papers/icpp06-smp-xfer.pdf`.
5. Burns, G., Daoud, R., Vaigl, J.: LAM: An open cluster environment for MPI. In: Proceedings of Supercomputing Symposium. (1994) 379–386
6. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004) 97–104
7. Myricom: (MPICH-GM) `http://www.myri.com/scs/`.
8. Snell, Q.O., Mikler, A.R., Gustafson, J.L.: Netpipe: A network protocol independent performace evaluator. In: Proceedings of Internation Conference on Intelligent InformationManagement and Systems. (1996)
9. Browne, S., Deane, C., Ho, G., Mucci, P.: PAPI: A portable interface to hardware performance counters. In: Proceedings of Department of Defense HPCMP Users Group Conference, Monterey, California. (1999)
10. Wallcraft, A.J.: The Halo benchmark. `http://www.sdsc.edu/SciComp/PAA/Benchmarks/Portal/Halo/halo.html` (1998)
11. Bailey, D.H., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center (1994)
12. Wong, F.C., Martin, R.P., Arpaci-Dusseau, R.H., Culler, D.E.: Architectural requirements and scalability of the NAS parallel benchmarks. In: Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), New York, ACM Press (1999) 41