# Micro-benchmarks for Cluster OpenMP Implementations: Memory Consistency Costs

H.J. Wong, J. Cai, A.P. Rendell, and P. Strazdins

Australian National University,
Canberra ACT 0200, Australia
{jin.wong,jie.cai,alistair.rendell,peter.strazdins}@anu.edu.au

**Abstract.** The OpenMP memory model allows for a temporary view of shared memory that only needs to be made consistent when `barrier` or `flush` directives, including those that are implicit, are encountered. While this relaxed memory consistency model is key to developing cluster OpenMP implementations, it means that the memory performance of any given implementation is greatly affected by which memory is used, when it is used, and by which threads. In this work we propose a micro-benchmark that can be used to measure memory consistency costs and present results for its application to two contrasting cluster OpenMP implementations, as well as comparing these results with data obtained from a hardware supported OpenMP environment.

## 1 Introduction

Micro-benchmarks are synthetic programs designed to stress and measure the overheads associated with specific aspects of a hardware and/or software system. The information provided by micro-benchmarks is frequently used to, for example, improve the design of the system, compare the performance of different systems, or provide input to more complex models that attempt to rationalize the runtime behaviour of a complex application that uses the system.

In the context of the OpenMP (OMP) programming paradigm, significant effort has been devoted to developing various micro-benchmark suites. For instance, shortly after its introduction Bull [3,4] proposed a suite of benchmarks designed to measure the overheads associated with the various synchronization, scheduling and data environment preparation OMP directives. Other notable OMP related work includes that of Sato et al. [16] and Müller [12].

All existing OMP micro-benchmarks have been developed within the context of an underlying hardware shared memory system. This is understandable given that the vast majority of OMP applications are currently run on hardware shared memory systems, but it is now timely to reassess the applicability of these micro-benchmarks to other OMP implementations. Specifically, for many years there has been interest in running OMP applications on distributed memory hardware such as clusters [14,8,9,2,7]. Most of these implementations have been experimental and of a research nature, but recently Intel released a commercial product that supports OMP over a cluster – Cluster OpenMP (CLOMP) [6]. This

R. Eigenmann and B.R. de Supinski (Eds.): IWOMP 2008, LNCS 5004, pp. 60–70, 2008.

interest, plus the advent of new network technologies that offer exceptional performance and advanced features like Remote Direct Memory Access (RDMA), stands to make software OMP implementations considerably more common in the future. Also it is likely that the division between hardware and software will become increasingly blurred. For example, in recent work Zeffer and Hagersten [19] have proposed a set of simplified hardware primitives for multi-core chips that can be used to support software implemented inter-node coherence.

This paper addresses the issue of developing a benchmark to measure the cost of memory consistency in cluster OMP implementations. In the OMP standard, individual threads are allowed to maintain a temporary view of memory that may not be globally consistent [13]. Rather, global consistency is enforced either at synchronization points (OMP `barrier` operations) or via the use of the OMP `flush` directive[1]. On a hardware shared memory system both operations are likely to involve hardware atomic and memory control instructions such as the `fence`, `sync`, or `membar` operations supported on x86, POWER, and SPARC architectures respectively. These operations can be used to ensure that the underlying hardware does not a) move any of the load/store operations that occurred before the barrier/flush to after or vice versa, and b) ensure that all load/store operations that originated before the barrier/flush are fully completed (e.g. not waiting on a store queue).

It is important to note that memory consistency and cache coherency are different concepts. Cache coherency policies ensure that there is a single well defined value associated with any particular memory address. Memory consistency on the other hand determines when a store instruction executed by one thread is made visible to a load operation in another thread and what that implies about other load and store operations in both threads [1].

A fundamental difference between hardware and software supported OMP is that much of the work associated with enforcing the OMP memory consistency model occurs naturally in the background on a hardware shared memory system, but this is not the case for most software supported OMP systems. That is on a typical hardware shared memory system the result of a store operation becomes globally visible as soon as it is propagated to a level where the cache coherency protocol knows about it (e.g. when it has been retired from the store queue). Thus if an OMP program has one thread executing on one processor that is periodically changing the value of a shared variable, while another thread is executing on another processor and is periodically reading the same variable, the reading thread is likely to see at least some of the different values written by the other thread, and this will be true regardless of whether there are any intervening OMP `barrier` or `flush` directives. This is not, however, required by the OMP memory consistency model (and must not be relied upon by any valid OMP program). From the OMP perspective it can be viewed as if the cache coherency hardware is speculatively updating global shared memory on

[1] Note that OpenMP `flush` operations may be implied. For instance, they are implied before reading from and after writing to `volatile` variables. Thus, the temporary and global views of these variables are kept consistent automatically.

the expectation that the current change will not be overwritten before the next `barrier` or `flush` directive is encountered; and if the change is overwritten, propagating it will have been a waste of time and bandwidth!

For software supported OMP systems, propagating changes to global memory generally implies significant communication cost. As a consequence global updates are generally stalled as long as possible, which for OMP means waiting until a thread encounters a `flush` or `barrier` directive. What happens then varies greatly between different cluster OMP implementations. The aim of this paper is to outline a micro-benchmark that can be used to quantify these differences, and then to illustrate its use on two software enabled OMP systems, comparing the results with those obtained from a hardware supported OMP system.

The following section outlines the memory consistency benchmark, while Section 3 details the two software supported OMP systems used in this work. Section 4 contains the results obtained for the two software OMP systems and the contrasting hardware supported OMP system. Finally Section 5 contains our conclusions and comments for future work.

## 2 MCBENCH: A Memory Consistency Benchmark

The goal of the Memory Consistency Benchmark (MCBENCH)[2] is to measure the overhead that can be attributed to maintaining memory consistency for an OMP program. To do this, memory consistency work is created by *first* having one OMP thread make a change to shared data and then flush that change to the globally visible shared memory; and *then* having one or more other OMP threads flush their temporary views so that the changes made to the shared data are visible to them.

As noted above it is important that the readers' flushes occur *after* the writer's flush, otherwise OMP does not require the change to have been propagated. Both these requirements are met by the OMP `barrier` directive since this contains both synchronization and implicit flushes [13]. Accordingly, the general structure used by MCBENCH is a series of change and read phases that are punctuated by OMP `barrier` directives (where implicit flushes and synchronization occurs) to give rise to memory consistency work.

Since the above includes other costs that are not related to the memory consistency overhead, it is necessary to determine a reference time. This is done by performing the exact same set of operations but using private instead of shared data. The difference between the two elapsed times is then the time associated with the memory consistency overhead.

To ensure that the same memory operations are performed on both the private and shared data, the MCBENCH kernel is implemented as a routine that accepts the address of an arbitrary array. Figure 1 shows that this array of $a$ bytes is divided into chunks of fixed size $c$ which are then assigned to threads in a round-robin fashion. In the Change phase, each thread changes the bytes in

[2] MCBENCH is available for download at `http://ccnuma.anu.edu.au/dsm/mcbench`.

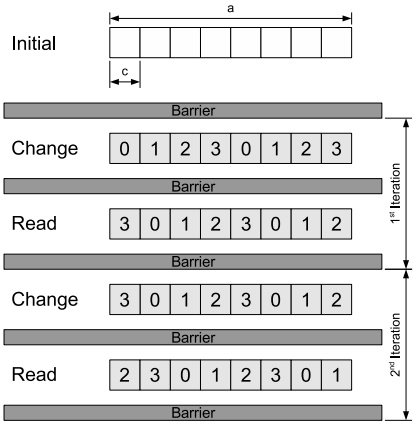


**Fig. 1.** MCBENCH – An array of size $a$-bytes is divided into chunks of $c$-bytes. The benchmark consists of Change and Read phases that can be repeated for multiple iterations. Entering the Change phase of the first iteration, the chunks are distributed to the available threads (four in this case) in a round-robin fashion. In the Read phase after the barrier, each thread reads from the chunk that its neighbour had written to. This is followed by a barrier which ends the first iteration. For the subsequent iteration, the chunks to Change are the same as in the previous Read phase. That is, the shifting of the chunk distribution only takes place when moving from the Change to Read phases.

their respective chunks. This is followed by a barrier, and the Read phase where the round-robin distribution used in the Change phase is shifted such that, had the array been a shared one, each thread will now read the chunks previously changed by their neighbours. The size of the shared array is the total number of bytes that was modified during the Change phase, and this is also the total number of modified bytes that must be consistently observed in the subsequent Read phase. Thus, this number represents the memory consistency workload in bytes that the underlying memory system must handle.

## 3 Two Software Supported OMP Implementations

Two software supported OMP implementations have been used in this work; both layer OMP over a page-based software Distributed Shared Memory (DSM) system [15,14,8,9,7]. The first is Intel's Cluster OpenMP product (CLOMP) [6] that uses a modified form of the TreadMarks DSM [10] for its underlying shared memory. The second is based on the Omni OpenMP Compiler [11] and uses the SCLIB DSM that has been developed by one of us [18]. The latter stands for "SCash LIBrary" and is so named because the consistency model used is the same as SCASH [5], but in contrast to SCASH it uses MPI for its communication requirements as opposed to the specialized PM communication library [17].

Two key activities that DSMs need to perform are detecting access to shared data and determining what changes, if any, have been made to local copies of shared data so that those changes can be made globally visible:

- **Detecting Shared Data Accesses** – At the granularity of a page, a subset of the virtual address space associated with each user process is designated as globally shared. Access to these shared pages is detected by using `mprotect` to modify the page's memory protection and implementing a handler for the associated `SIGSEGV` signal. To catch both read and write accesses, the `PROT_NONE` setting can be used, while to catch only writes `PROT_READ` is used.
- **Determining Changes to Shared Data** – When a write is attempted on a read-only page, a copy of the page is made before modifying the memory protection to allow the write to proceed. This process is called *twinning*. The presence of the twin then makes it possible to determine if changes have been made to any data on that page at a later stage. This process is called *diffing*. Using the *twinning-and-diffing* strategy makes it possible to have multiple threads modifying the same page concurrently.[3]

TreadMarks and SCLIB differ in that SCLIB is a home-based DSM while TreadMarks is not (or homeless). Home-based means that for each page of globally shared memory there is a uniquely defined node that is responsible for maintaining the currently valid version of that page, i.e. the *master copy*. An advantage of the home-based approach is that a thread only needs to communicate with one node in order to obtain a fresh copy of any given shared page or to send *diffs* relating to that page. In contrast, TreadMarks does not maintain any master copy of pages. Rather, *diffs* are held by their respective writers and only requested for when required. The advantage of this approach is that if a page is only accessed by one thread, there would be no communication of *diffs* pertaining to that page. This makes the DSM naturally adaptive to data access phase changes within the user application.

The difference between the two approaches is succinctly captured by what happens in a barrier. In TreadMarks, *diffs* are created for all modified pages which are then stored in a repository. Write notices are communicated amongst the threads, detailing which pages have been modified and by which threads. Pages that have been modified are also invalidated by changing the page protection to `PROT_NONE`. Post barrier, any access made to a modified page will invoke the `SIGSEGV` handler that fetches diffs from the various writer threads and applies them to the local page, thereby updating the page. At this point the page can be made read-only.

SCLIB is similar to TreadMarks in that *diffs* for all the modified pages are made during the barrier. However, rather than storing these in repositories, the *diffs* are communicated immediately to the page-home and applied directly onto the *master copy* of the page. Thus, at the end of the barrier, all master copies of

[3] Although multiple writers can modify the same page concurrently, these should be on separate portions of that page, otherwise the result from merging the diffs will be non-deterministic.

**Table 1.** Details of the experimental environments used in this work

| Item | Detail | GigE Cluster | InfiniBand Cluster | Hardware (Sun V1280) |
|---|---|---|---|---|
| CPU | Manufacturer | AMD | Intel | Sun |
| | Type | Athlon 64 X2 4200 | Xeon 5150 | UltraSPARC |
| | Clock | 2.2GHz | 2.66GHz | 900MHz |
| | Cores | 2 | 2 | 1 |
| Nodes | Count | 8 | 20 | 3 (boards) |
| | CPU Sockets | 1 | 2 | 4 (per board) |
| Network | Type | Gigabit Ethernet | InfiniBand (4x DDR) | - |
| | Latency | 60 usec | 3 usec | - |
| | Bandwidth | 98 MB/s | 1221 MB/s | - |
| OpenMP | Impl1 | ICC-10.0.026, CLOMP-1.3, -O | ICC-10.0.026, CLOMP-1.3, -O | - |
| | Impl2 | Omni/SCLIB, OpenMPI-1.1.4 (GCC-3.3.5), -O | Omni/SCLIB, OpenMPI-1.2.5 (GCC-4.1.2), -O | - |
| | Impl3 | - | - | Sun Studio 11, -xopenmp=noopt |

pages will be up-to-date. Post barrier, a page is read-only for the home thread and either read-only or invalid for all other threads, depending on whether there were changes[4]. Subsequent page faults cause the contents of a page to be fetched from the page-home, updating the temporary view of the faulting thread.

In summary, TreadMarks only sends the locally made changes to other threads when and if those other threads request them [5]; this restricts communication within the barrier to relatively short messages that just notify the other nodes that something has changed in a given page. SCLIB, on the other hand, sends the actual changes to a page to the home thread where the master copy is updated during the barrier event. In principle the time spent within the TreadMarks barrier is shorter than for SCLIB, but this comes at the expense of possibly having to retrieve *diffs* from multiple threads after the barrier and apply them. *A priori* it is not easy to judge which scheme is best.

## 4 Experimental Results

MCBENCH was used to measure the memory consistency overheads of the two software OMP implementations introduced in the previous section on the two different cluster environments detailed in Table 1. Some results were also obtained using the Sun Studio 11 OMP compiler on a Sun V1280 hardware shared memory machine.

[4] In a single-writer situation, the writer, if not the home of the page, will not receive an invalidation notice because its copy is as up-to-date as the master copy.

[5] Or when the repository of diffs become too full.

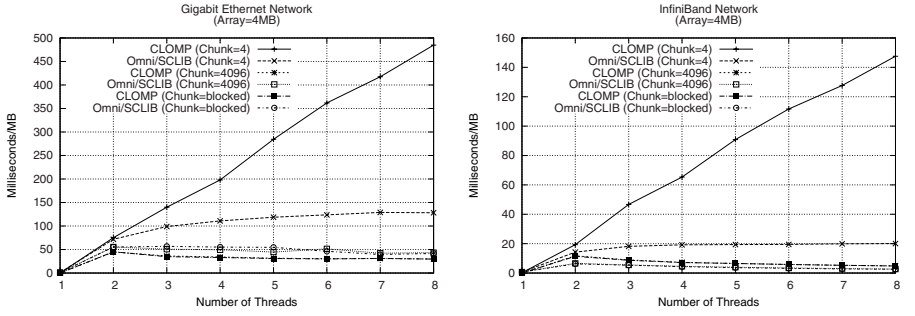


**Fig. 2.** MCBENCH results using an array of 4MB on the Gigabit Ethernet and InfiniBand clusters. The chunk size is 4 bytes, 4096 bytes, and blocked (4MB/$n$).

There are three parameters that can be adjusted when running MCBENCH. These are the size of the array ($a$), the chunk size ($c$), and the number of threads ($n$). The first parameter, $a$, represents the OMP memory consistency workload at each iteration in terms of bytes. This array is then divided into chunks of $c$ bytes, which the available $n$ threads then modify and read using the round-robin distribution illustrated in Figure 1. Thus, varying the chunk size modifies the memory access patterns of the threads.

### 4.1 Analysis: Software OMP Implementations

In studying the software OMP implementations, we observe how the overheads of the implementations scale with the number of threads. Scalability of this nature is important for clusters because the primary means of increasing computational power in a cluster is to add more nodes.

The chunk sizes that were used are 4 bytes, 4096 bytes, and *blocked* ($\lfloor \frac{a}{n} \rfloor$ bytes). At 4 bytes, the memory access pattern represents the worst case for the two software OMP implementations because they are layered on page-based DSMs whose pages are many times larger than $4n$ bytes. The result of such an access pattern is that each thread has to perform memory consistency over the whole array even though it only modifies/reads $\frac{a}{n}$ bytes in a phase. The second chunk size is 4096 bytes and is the same as the page size of the DSMs used by both software OMP implementations. By further aligning the shared array to the page line, a dramatic difference can be observed in the overheads. Lastly, the *blocked* chunk size is a simple block distribution of the shared array between the available threads. This represents the case of accessing large contiguous regions of shared memory.

Figure 2 plots the overhead in terms of "Milliseconds/MB" for both software OMP implementations while using a 4MB shared array together with the three different chunk sizes. As expected, both implementations give their worst performances when the chunk size is 4 bytes. However, these scale very differently. The SCLIB DSM is home-based and so maintains a master copy of each page.

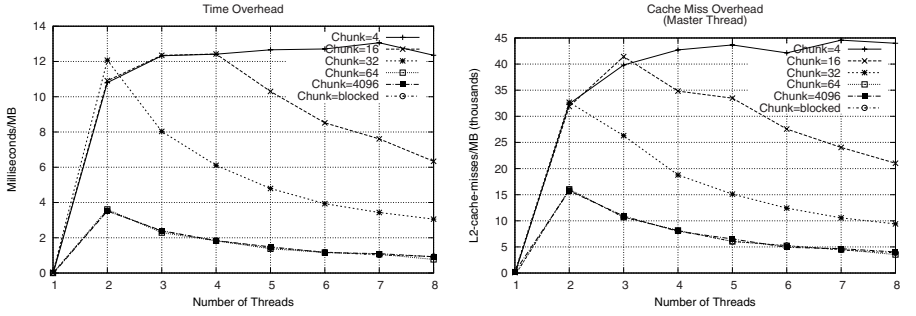


**Fig. 3.** MCBENCH results on the Sun V1280 SMP. A 4MB array is used with chunk sizes varied from 4 bytes to blocked (4MB/$n$). The left shows the overhead in terms of Microseconds/MB and the right reports the L2-cache-misses/MB observed by the master thread.

During each iteration of the worst case access pattern, each thread will fetch, twin, diff and later send diffs for $\frac{n-1}{n}$ of the pages in the shared array. This explains the $O(\frac{n-1}{n})$ scaling observed in Figure 2 for both clusters.

In contrast, CLOMP is not home-based. Instead of sending modifications to a central master copy, modifications are maintained by the various writers and must be retrieved from each of these if a thread encounters a subsequent page fault for this page. Since every page has been modified by every thread in the worst case scenario, every page fault during the Read Phase of the benchmark inevitably results in communication with all the other threads. This results in the observed $O(n)$ complexity.

For the 4096-byte and *blocked* chunk sizes, the number of pages accessed for modification or reading can be approximated as $\frac{1}{n} \times \frac{a}{pagesize}$. Thus $O(\frac{1}{n})$ complexity is observed for the CLOMP implementation. For Omni/SCLIB, only $\frac{n-1}{n}$ of the pages accessed are *remote* (i.e. the master copy is at another thread). Due to this, only $\frac{n-1}{n} \times \frac{1}{n}$ of the pages need to be fetched. Therefore the implementation scales at $O(\frac{n-1}{n^2})$. Although, for large enough $n$ this complexity approaches $O(\frac{1}{n})$.

### 4.2 Analysis: Hardware OMP Implementation

The L1 data cache on the Sun V1280 has a line size of 32 bytes, with 16-byte sub-blocks. The L2 cache is of size 8MB, with 512-byte lines with 64-byte sub-blocks (these form the unit of cache coherency), and is 2-way associative with a random replacement policy.

Figure 3 shows the effect of varying the chunk size $c$ for $a = 4$MB, i.e. over data that can fit entirely in the L2 cache. In addition to the time overhead metric, MCBENCH was instrumented to count the L2 cache misses observed by the master thread. As expected, that there is a reasonably high correlation between the time-based and counter-based overheads. Again, $c = 4$ represents a worst-case

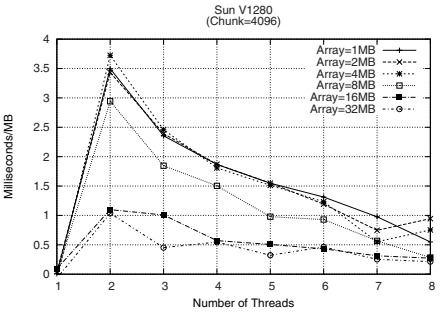


**Fig. 4.** MCBENCH results on the Sun V1280 SMP. The effect of varying the array size for 4096-byte chunks is shown.

scenario, showing roughly $O(\frac{n-1}{n})$ scalability due to cache-line transfers (including a L1 cache invalidation) that could possibly occur for every element updated. The observed results suggest that the coherency-related hardware has been saturated in this manner. The performance is markedly improved and the scaling becomes roughly $O(\frac{1}{n})$ when the chunk size, $c$, is increased to 16 and 32 bytes. Finally, when $c$ reaches or exceeds the size of the unit of cache coherency, transfers occur once for every 64-byte region accessed that needs to be made consistent. The backplane fat-tree topology hardware is able to parallelize the coherency traffic effectively at this point, resulting in greatly reduced overhead and a clear $O(\frac{1}{n})$ scaling.

Figure 4 shows the effect of increasing the array size from 1 MB to 32 MB. The results indicate that the consistency overhead is apparently reduced when the array size exceeds the L2 cache capacity. While this may seem at first counter-intuitive, at this point we see the distinction between memory and cache consistency overheads emerging, and it must be noted that MCBENCH results are the difference between the access patterns for shared and private arrays. For the latter, a simple cache simulation program showed that for this L2 cache, there is a cache hit rate of only 15% ($a = 16$ MB) and 1% ($a = 32$ MB) upon the first byte in each line that is accessed. Hence, the effect is due to the introduction of memory access overheads for the private array, which, while not as great as the cache coherency overheads for the shared array, reduce some of the difference between the two cases.

## 5 Conclusions

The overhead reported by MCBENCH is determined by the difference in performance between memory operations on shared and private data. This is achieved by using an access pattern that is designed to force the memory subsystem to perform memory consistency work when shared data is used.

Although MCBENCH was designed in the context of cluster OpenMP implementations, it has been demonstrated that the benchmark can be used on both

software and hardware OMP implementations. The results show that the magnitude of the memory consistency overhead is greatly influenced by the structure of the underlying memory subsystem. In particular, varying the chunk size shows that the granularity of coherency is able to sway results from one end of the scale to the other. This is true for both software and hardware based OMP implementations. Beyond the coherency granularity, the same scaling is observed for all three implementations.

Finally, it is important to remember that micro-benchmarks measure specific system overheads by using small well defined operations to exercise that aspect of interest. Thus, it is rare for a single micro-benchmark to explain the performance of a full application completely. Rather, they give an appreciation of the overheads of specific conditions that may manifest during the execution of the application of interest.

## Acknowledgments

We are grateful to the APAC National Facility for access to the InfiniBand cluster and to R. Humble and J. Antony for their assistance. This work is part funded by Australian Research Council Grant LP0669726 and Intel Corporation, and was supported by the APAC National Facility at the ANU.

## References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
2. Basumallik, A., Eigenmann, R.: Towards automatic translation of OpenMP to MPI. In: Arvind, Rudolph, L. (eds.) Proceedings of the 19th Annual International Conference on Supercomputing (19th ICS 2005), Cambridge, Massachusetts, USA, jun 2005, pp. 189–198. ACM Press, New York (2005)
3. Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proc. of the European Workshop on OpenMP (EWOMP 1999) (1999), `citeseer.ist.psu.edu/bull99measuring.html`
4. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. ACM SIGARCH Computer Architecture News 29(5), 41–48 (2001)
5. Harada, H., Tezuka, H., Hori, A., Sumimoto, S., Takahashi, T., Ishikawa, Y.: SCASH: Software DSM using high performance network on commodity hardware and software. In: Eighth Workshop on Scalable Shared-memory Multiprocessors, May 1999, pp. 26–27. ACM Press, New York (1999)
6. Hoeflinger, J.P.: Extending OpenMP to clusters. White Paper Intel Corporation (2006)
7. Huang, L., Chapman, B.M., Liu, Z.: Towards a more efficient implementation of OpenMP for clusters via translation to global arrays. Parallel Computing 31(10-12), 1114–1139 (2005)
8. Karlsson, S., Lee, S.-W., Brorsson, M.: A fully compliant OpenMP implementation on software distributed shared memory. In: Sahni, S.K., Prasanna, V.K., Shukla, U. (eds.) HiPC 2002. LNCS, vol. 2552, pp. 195–208. Springer, Heidelberg (2002)

9. Kee, Y.-S., Kim, J.-S., Ha, S.: ParADE: An OpenMP programming environment for SMP cluster systems. In: Supercomputing 2003, p. 6. ACM Press, New York (2003)
10. Keleher, P., Cox, A., Dwarkadas, S., Zwaenepoel, W.: TreadMarks: Distributed memory on standard workstations and operating systems. In: Proceedings of the 1994 Winter Usenix Conference, pp. 115–131 (1994)
11. Kusano, K., Satoh, S., Sato, M.: Performance Evaluation of the Omni OpenMP Compiler. In: Valero, M., Joe, K., Kitsuregawa, M., Tanaka, H. (eds.) ISHPC 2000. LNCS, vol. 1940, pp. 403–414. Springer, Heidelberg (2000)
12. Müller, M.S.: A Shared Memory Benchmark in OpenMP. In: Zima, H.P., Joe, K., Sato, M., Seo, Y., Shimasaki, M. (eds.) ISHPC 2002. LNCS, vol. 2327, pp. 380–389. Springer, Heidelberg (2002)
13. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 2.5 (May 2005)
14. Sato, M., Harada, H., Hasegawa, A.: Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. Scientific Programming 9(2-3), 123–130 (2001)
15. Sato, M., Harada, H., Ishikawa, Y.: OpenMP compiler for software distributed shared memory system SCASH. In: Workshop on Workshop on OpenMP Applications and Tool (WOMPAT 2000), San Diego (2000)
16. Sato, M., Kusano, K., Satoh, S.: OpenMP benchmark using PARKBENCH. In: Proc. of $2^{nd}$ European Workshop on OpenMP, Edinburgh, U.K (September 2000), `citeseer.ist.psu.edu/article/sato00openmp.html`
17. Tezuka, H., Hori, A., Ishikawa, Y.: PM: A high-performance communication library for multi-user parallel environments. Tech. Rpt. TR-96015, RWC (November 1996)
18. Wong, H.J., Rendell, A.P.: The design of MPI based distributed shared memory systems to support OpenMP on clusters. In: Proceedings of IEEE Cluster 2007 (September 2007)
19. Zeffer, H., Hagersten, E.: A case for low-complexity MP architectures. In: Proceedings of Supercomputing 2007 (2007)