Software Barrier Performance on Dual Quad-Core Opterons

Jie Chen and William Watson III
Scientific Computing Group
Jefferson National Lab
Newport News, Virginia 23606
USA
Email: {chen,watson}@jlab.org

Abstract

Multi-core processors based SMP servers have become building blocks for Linux clusters in recent years because they can deliver better performance for multi-threaded programs through on-chip multi-threading. However, a relative slow software barrier can hinder the performance of a data-parallel scientific application on a multi-core system. In this paper we study the performance of different software barrier algorithms on a server based on newly introduced AMD quad-core Opteron processors. We study how the memory architecture and the cache coherence protocol of the system influence the performance of barrier algorithms. We present an optimized barrier algorithm derived from the queue-based barrier algorithm. We find that the optimized barrier algorithm achieves speedup of 1.77 over the original queue-based algorithm. In addition, it has speedup of 2.39 over the software barrier generated by the Intel OpenMP compiler.

1. Introduction

Recently, multi-core processors based on the chip multi-threading/processing (CMT/CMP) [12] architecture, which uses multiple single-thread processor cores in a single CPU and executes multiple threads in parallel across the multiple cores, appear to dominate both the high-end and the mainstream computing markets. Because a multi-core processor offers better performance-to-cost ratios relative to a traditional multi-processor solution such as the Symmetric MultiProccssing (SMP) systems, computers based on multi-core processors, e.g., Intel quad-core Xeons [15] and AMD quad-core Opterons [6], become building blocks for high performance computing Linux clusters. For a system with a single multi-core processor, it is indeed a slim implementation of an SMP node on a chip. For a system with multiple multi-core processors organized in the SMP fashion, it behaves as a traditional SMP machine, where the number of processors is the number of cores.

Scientific applications can benefit from multi-core processors, where code can be executed in multiple threads each of which runs on a dedicated processing core. Especially applications of data parallelism, where multiple threads execute the same code on different sets of data, can improve their performance dramatically relative to their single threaded versions. Applications of this type usually utilize the fork-join programming model found in any OpenMP [10] application. However, the choice of a barrier algorithm is critical to the performance of any application of such type since each join action leads to executions of barrier synchronization by all threads. A barrier synchronization for a group of threads means that any thread must stop at this point and cannot proceed until all other threads reach this barrier.

Past research has focused on how to improve barrier performance by either reducing memory contention introduced by accessing shared flags within a barrier or by reducing the critical path of a barrier [3] [8]. However, there is a lack of studies on how the memory subsystem and the cache coherence protocol of a system influence the performance of barrier algorithms. In addition, most well known barrier algorithms target to large SMP machines. This paper studies the performance of a few known software barrier algorithms on an SMP server consisting of two AMD quad-core Opteron processors to shed some light on the above issues, and introduces an optimized barrier algorithm for the server.

The paper is organized as follows. Section 2 describes the software and hardware environment where our performance evaluations are carried out. Section 3 overviews the memory organization and the cache coherence protocol utilized by our test machine. Section 4 presents a few known barrier algorithms. Section 5 analyzes the performance of the algorithms based on memory/cache transactions and introduces an optimized barrier algorithm. Sec-

1-4244-0328-6/06/$20.00 ©2006 IEEE.

tion 6 compares the performance of the algorithms along with an OpenMP compiler generated barrier. Section 7 concludes.

2. Hardware and Software Environment

Our test machine is equipped with two AMD quad-core Opteron 2347 processors [1] running at 1.9 GHz. Each processor has its own memory controller shared by four cores which can access the memory and caches on the other processor through coherent HyperTransport [5] links. Table 1 lists some of the important information of the test machine.

Table 1: Information of the test machine

CPUs	L1	L2	L3	Memory
Two Quad-Core	64K Data 64K Instr	512 KB Private	2MB Shared	4x1GB

The test machine is running Fedora Core 7 Linux x86_64 distribution with a Linux kernel of 2.6.23. One compiler is utilized: the Intel icc (64bit) version 10.0.023 supporting OpenMP. The synchronization overhead introduced by the OpenMP directives is measured through the EPCC microbenchmark [2]. On the other hand, the synchronization overhead induced by software barriers are collected through a slightly modified EPCC microbenchmark program. All benchmark and test programs are compiled with -O3 optimization flag. During execution of any benchmark and test program, each thread is bound to a particular core to avoid the overhead of thread migration.

3. Memory Architecture

In a commodity multi-core small SMP system, the memory system usually is organized in one of the following two ways: a shared bus architecture where each core accesses the memory uniformly through a common bus; a cache coherent Non-Uniform Memory Architecture (ccNUMA) where each core has different access speeds to its local and remote memory through different paths and has channels to maintain cache coherence with the other cores. Systems with dual quad-core Opteron processors opt the ccNUMA architecture by deploying an integrated memory controller and local memory to each processor which is directly connected to the other processors by a Coherent HyperTransport link. Each core communicates with the other cores through a system request interface (SRQ) [1], which in turn talks to a non-blocking crossbar. The crossbar is then connected to the memory controller and to the various HyperTransport links. Fig. 1 illustrates the internal architecture of an Opteron processor.

In a multi-core SMP system such as our test machine, the memory system is organized in a hierarchical way including fast multi-level of caches and relatively slow memory. The quad-core Opteron processors use three levels of caches to accelerate data processing: private L1 cache, private L2 cache and shared L3 cache. The L1 cache is a write-allocate and writeback [14] cache and uses a least-recently-used replacement policy. The L1 cache behaves as a traditional lower level cache in the sense that L1 loads a cache block from the memory subsystem directly upon a read/write miss. The L2 cache is a private cache and is an exclusive cache architecture. The L2 cache only contains victim or copy-back cache blocks that are to be written to the memory subsystem as a result of a conflict miss. The L3 cache is a victim cache and is dynamically shared between all four cores to promote fast cache sharing among the cores.

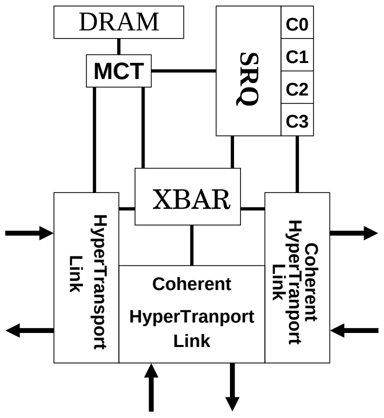

Fig. 1: Internal structure of an Opteron processor

A cache coherence protocol is a mechanism ensure that all cache copies remain consistent when the contents of that memory location are modified. The quad-core Opteron processors utilize a protocol called MOESI [13], named from five states of the protocol: Invalid, Exclusive, Shared, Modified and Owned. The MOESI protocol has one more state than another popular cache coherence protocol MESI [11] deployed by the Intel Xeon multi-core processors. The extra state, which is the Owned state, is to reduce the number of memory write backs during the MESI protocol operations. These memory write backs happen when a modified cache block has to be written back to the memory subsystem in order for other shared caches to load the correct value. Under the MOESI protocol, however, a cache block in the Owned state holds the most recent/correct copy of the data and the copy in the main memory can be stale. Only one cache can hold a block of data in the Owned state, all others must hold the data in the Shared state. The owner of a cache block is responsible to update other caches that try to read/write the block. This avoids the need to write a modified cache back to the main memory. Data flagged as in the Owned state in the cache of one core can be delivered directly to another core on the different CPU via a CPU-to-CPU HyperTransport link or to another core on the same CPU via the SRQ.

Traditionally, a simple barrier can be implemented by having each thread increment/decrement one or more barrier variables. Hence the performance of accessing these variables is crucial to the performance of a barrier. Table 2 shows the random access latency to each level of the memory system on the test machine using the LMbench [7] benchmark. It is clear that any barrier implementation should avoid cache conflicts that could evict shared barrier variables to the lower level caches.

Table 2: Random memory access latency

L1(ns)	L2(ns)	L3(ns)	Memory(ns)
1.57	8.03	26.07	107.3

Cache to cache transfer latency is also important to multi-threaded applications on a multi-core SMP machine. The AMD quad-core Opteron processors have two ways to achieve this: the SRQ channels for transferring data among caches on different cores within one CPU, and the HyperTransport channels for delivering cache data from one core to another across CPU boundaries. It is obvious that an SRQ channel transfers data faster than a HyperTransport channel. Executing a simple C program that uses two threads to send and receive cache data back and forth between two cores on the test machine shows an SRQ cache-to-cache latency of 105.4 ns and a HyperTransport cache-to-cache latency of 138.7 ns. That is the difference of 33.3 ns in cache transfer latency between an SRQ channel and a HyperTransport channel.

4. Barrier Algorithms

A software barrier synchronizes a number of cooperating threads that repeatedly perform some work and then wait until all threads are ready to move to the next computing phase. Fig. 2 illustrates the timing information for a single barrier operation/synchronization. The total elapsed time of a single barrier operation is the time difference between the time of the first thread arriving at the barrier and the time of the last thread leaving the barrier. The total time can be further divided into two phases: the gather phase is the time period during which each thread signals its arrival at the barrier; the release phase denotes the time interval during which each thread is notified the completion of the barrier operation and is allowed to resume execution. During a barrier operation, a thread can perform no other computation except signaling its arrival at the barrier and being signaled the end of the barrier operation. Therefore, to improve the performance of a barrier algorithm is to reduce the total time of the barrier operation.

There are a few popular barrier algorithms used over the years, such as the centralized barrier, the combining tree barrier, the tournament barrier [8] and so on. The centralized barrier works well for a small number of threads but does not scale well for a large number of threads because all threads contend for the same set of variables. The combining tree and the tournament barrier reduce the above contention and work best for a large SMP system but not particularly well for a small SMP system [4]. Recently, the queue-based barrier algorithm [3] has gained popularity because it reduces the contention, performs well for small and large SMP systems and is easy to implement.

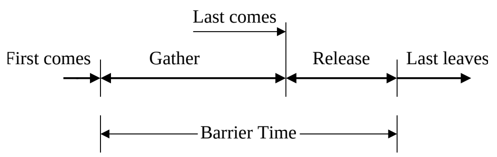

Fig. 2: Timing for a single barrier

The implementation of the centralized barrier algorithm (given below) uses one shared counter variable and one shared release flag. During the gather phase of a centralized barrier, each thread reduces the counter variable by one and waits on any change of the release variable. The barrier enters the release phase when the counter variable is reduced to zero by a thread. The thread increases the release variable by one, which signals the other waiting threads to proceed.

```
int flag=atomic_int_get(&release);
int count=atomic_int_dec(&counter);
if (count==0){
  atomic_int_set(&counter, num_threads);
  atomic_int_inc(&release);
}
else spin_until (flag ≠ release)
```

To avoid contention for the shared counter variable, the queue-based barrier algorithm (given below) picks one thread as the coordinating-thread or the master-thread and allocates a global array of flags. During the gather phase, each thread participating in the barrier operation signals its arrival at the barrier by increasing its flag variable by one in the flag array, and then spins on a shared release flag. It is the master-thread's responsibility to check the above signal flags to find out whether the other threads have arrived at the barrier. The barrier enters the release phase once all threads have arrived at the barrier. The master thread then adds one to the release flag for which non-master threads are waiting. Therefore the release phase of the queue-based barrier algorithm is identical to the centralized barrier algorithm.

```
typedef struct cflag{
  int volatile c_flag;
  /* each flag on a different cache line */
```

```
    int c_pad[CACHE_LINE_SIZE-1];
}cflag_t;
typedef struct qbarrier{
    int volatile release;
    char br_pad[CACHE_LINE_SIZE-1];
    cflag_t flags[1];
}qbarrier_t;
/* Master Thread */
int localflags[num_threads];
for(i=1;i<num_threads;i++){
    while (barrier→flags[i].cflag==localflag[i])
        ;
    localflags[i]=barrier→flags[i].cflag;
}
atomic_inc(&barrier→release);
/* Thread i */
int rkey=barrier→release;
++(barrier→flags[i].cflag);
while(rkey==barrier→release);
```

5. Analysis of Barrier Algorithms

To find out how the memory architecture of the quad-core Opteron processors effects the performance of a barrier algorithm, we analyze memory and cache transactions of two barrier algorithms under the MOESI cache coherence protocol. We then introduce a slightly modified queue-based algorithm that could perform better.

The performance of a barrier is clearly influenced by the number of accesses to the main memory or the number of cache-to-cache transfers and whether the memory/cache transactions can be carried out in parallel or can be pipelined. The difference between the centralized algorithm and the queue-based algorithm lies in the gather phase of each algorithm. The release phase is the same. For the simplicity of analyzing of these algorithms, let us assume that there are n processing cores. Furthermore, let us use RdX to denote the READ_EXCLUSIVE bus transaction generated by a cache write miss, and Rd to denote the READ bus transaction. Finally, we assume that a thread running on a core has the same id as the core id.

5.1. The Centralized Barrier Algorithm

Under the MOESI protocol deployed on the quad-core Opteron processors, memory transactions could be avoided if there are no cache eviction to the memory subsystem. Let us assume that one core initially has updated values of the counter variable and the release variable, and that the other caches are all invalidated. Fig. 3 demonstrates the bus and the cache transactions for thread 2 arriving at a centralized barrier, where the numbers inside the small circles denote the sequence of the transactions. Now thread 1 is the owner of the counter variable and it updates the cache of thread 2 which has generated the RdX bus transaction upon trying to write to the counter variable. Thread 1 invalidates its own cache block after the update is carried out via a cache-to-cache transfer channel. Thread 2 then becomes the new owner of the counter variable right after it decrements the variable. When thread 1 later checks the counter variable, thread 2 updates the cache of thread 1 immediately without any memory transaction. The above discussion applies to all the threads. There is no more cache-to-cache transaction when the last thread sets the counter variable to be the same as the number of threads because the last thread is already the owner of the counter variable. Hence there is a total of $2n$ cache-to-cache transactions during the gather phase of the barrier. It is easy to see that there are $n-1$ cache updates of the release flag from one core to the other cores in the release phase. Therefore, the total number of cache-to-cache transactions is $2n + n - 1 = 3n - 1$ during a centralized barrier operation. Worst of all, most of the above transactions can not be carried out in parallel because of the serialization of the RdX transactions on the same cache block of the counter variable. This leads to performance degradation of the barrier algorithm. The synchronization time of a centralized barrier for n threads is clearly is $O(n)$.

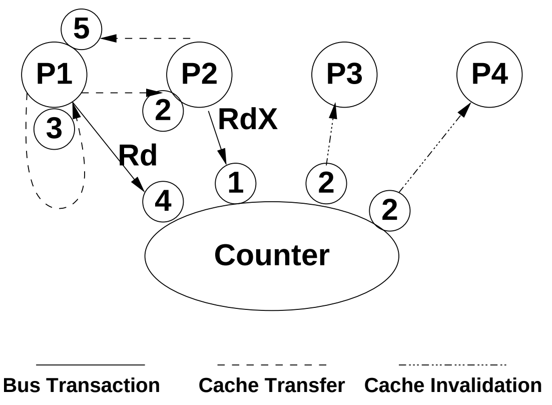

Fig. 3: Bus and cache transactions during the gather phase of a centralized barrier

5.2. The Queue-based Barrier Algorithm

Similar to the above discussions for the centralized barrier algorithm under MOESI protocol, there could be no memory transactions during the synchronization of a queue-based barrier. Initially, the master thread is the owner of all signal flags. Fig. 4 presents a snapshot for thread 2 arriving at the barrier. When thread 2 tries to update its signaling flag, the master thread updates the cache of thread 2 and invalidates its own copy. Therefore, thread 2 becomes the new owner of the cache block of its own signal flag right after it increases the value of the flag. Thread 2 later updates its signal flag in the global flag array when the master thread checks whether the other threads have arrived. Hence there are two cache transactions for each of the $n-1$ non-master threads. The total number of cache-to-cache transactions during the synchronization of a queue-based barrier is $2(n-1)+(n-1)=3n-3$, where the last $n-1$ is the contribution from the release phase of the barrier which behaves the same as a centralized barrier during the release phase.

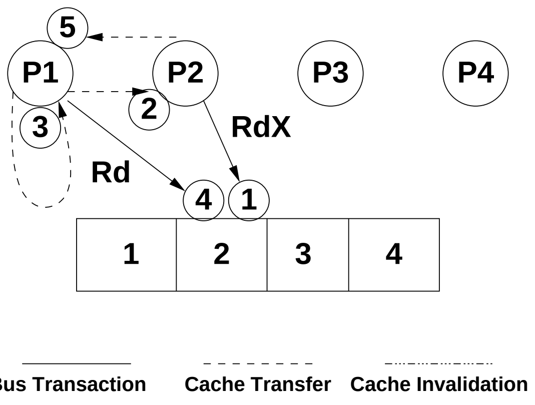

Fig. 4: Bus and cache transactions during the gather phase of a queue-based barrier

The number of cache transactions of a queue-based barrier is not much smaller than that of a centralized barrier. However, the $2(n-1)$ cache transactions during the gather phase of a queue-based barrier can be carried out effectively in parallel because each thread updates its flag variable independently. Therefore, the effective number of cache transactions during the gather phase for n threads is reduced from $O(n)$ to a constant, i.e., $O(1)$. During the release phase, however, the master thread has to send out $n-1$ cache invalidations to invalidate $n-1$ remote copies of the release flag, to process $n-1$ Rd bus requests from $n-1$ cores, and to transport $n-1$ cache updates to $n-1$ cores. It is difficult to pipeline these bus transactions and cache transfers because of the read contention for the release variable and the extreme short latencies of the SRQ/HyperTransport channels on the test machine. Therefore the total time complexity of the queue-base algorithm cannot be $O(1)$. Nonetheless, the queue-based barrier algorithm should perform much better than the centralized barrier algorithm does because it has $O(1)$ time complexity during gather phase and it performs the same as the centralized barrier algorithm does during the release phase.

5.3. Modified Queue-based Barrier Algorithm

To relieve the read contention for the shared release flag in the original queue-based algorithm, a modified algorithm based on the suggestion of using separate signal flags [9] is given below. The data structures and variables used in the following code segment are the same as in the original queue-based algorithm.

```
/* Master Thread */
for(i=1;i<num_threads;i++){
  while (barrier→flags[i].cflag==localflag[i])
     ;
}
for(i=1;i<num_threads;i++){
  ++(barrier→flags[i].cflag);
  localflags[i]=barrier→flags[i].cflag;
}
/* Thread i */
int key = barrier→flags[i].cflag + 1;
++(barrier→flags[i].cflag);
while (key == barrier→flags[i].cflag);
```

The modified algorithm uses the same array of flags to signal the arrival of each thread at a barrier and to release each thread from the barrier. This is different from the suggestion [3] of using a separate array of flags as the releasing flags. A separate release array could cause cache conflict misses and hence could degrade the performance. Fig. 5 demonstrates the bus and cache transactions during the release phase of the modified algorithm. When the master thread increases the value of the flag for thread 2, it invalidates the cache copy inside thread 2 and becomes the owner again of the flag. When thread 2 checks the value of its flag, the master thread updates the cache copy inside thread 2 upon receiving the corresponding Rd bus transaction. There is only one cache transaction for thread 2 during the release phase. The total number of cache transactions is $n-1$. A modified queue-based barrier generates the same number of cache transactions as an original queue-based barrier during the release phase.

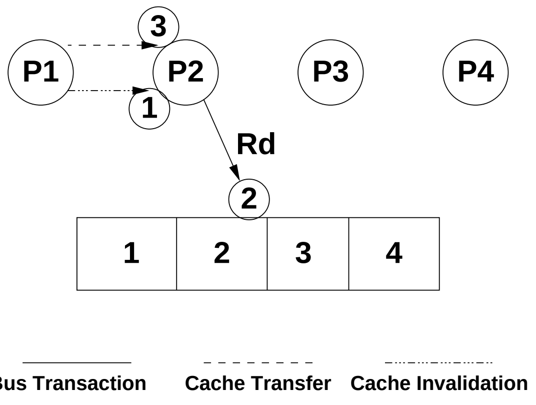

Fig. 5: Bus and cache transactions during the release phase of a modified queue-based barrier

These $n-1$ cache transactions/updates in the release phase could be pipelined because the updates are all independent. In addition, the quad-core Opteron processors utilize a dedicated long store queue called LS2 inside the Load-Store Unit (LSU) [1] to facilitate the write pipeline. Hence the performance of the modified queue-based barrier algorithm should be better than that of the original queue-based algorithm.

6. Performance of the Barrier Algorithms

To evaluate performance, we use the synchronization overhead introduced by a barrier algorithm as the metric of the barrier algorithm. The overhead values of these barrier algorithms are collected using the modified EPCC microbenchmark program. For the purpose of comparison, we use the EPCC microbenchmark program to obtain the overhead values of the barrier generated from the Intel OpenMP compiler. Fig. 6 shows the overhead values in terms of CPU cycles for three barrier algorithms and the OpenMP barrier.

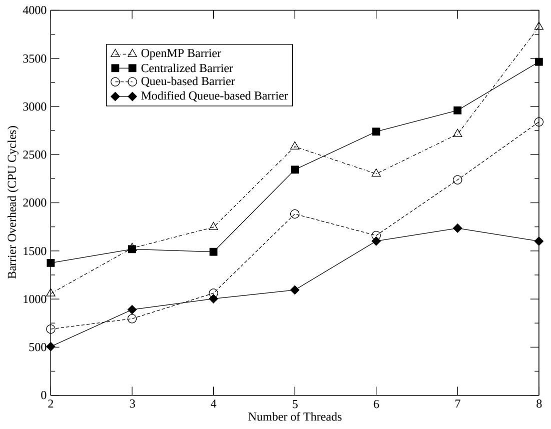

Fig. 6: Performance of barrier algorithms

Both queue-based barrier algorithms noticeably outperform the centralized barrier algorithm for two to eight threads. This confirms that the write memory contention for the shared counter variable inside a centralized barrier causes serialization of cache/bus transactions which lengthen the barrier synchronization. The modified queue-based algorithm performs almost the same as the original version for two to four threads because an original queue-based barrier has less severity of the read contention for the release flag for the small number of threads when fast cache-to-cache transfers happen among cores within a processor. However, the modified queue-based barrier algorithm performs noticeably better than the original version for five to eight threads due to its capability of pipelining cache/bus transactions related to the array of release flags in the algorithm. The obvious ridges in most of the plots in the figure from four to five threads stem from the longer latency of a cache-to-cache transfer through a HyperTransport channel than through an SRQ channel. Finally the speedups of all algorithms against the OpenMP barrier implementation are shown in Table 3, where C, Q and MQ stand for the centralized, the queue-based and the modified queue-based algorithms, respectively.

Table 3: Speedup over OpenMP barrier

	2	3	4	5	6	7	8
C	0.77	1.01	1.17	1.10	0.84	0.92	1.10
Q	1.54	1.92	1.65	1.37	1.39	1.21	1.34
MQ	2.08	1.72	1.74	2.36	1.44	1.56	2.39

It is easy to see that the two queue-based barrier algorithms have substantial speedups over the OpenMP barrier across the given numbers of threads. More importantly, our modified queue-based barrier algorithm has speedup of 1.77 over the original version in the case of eight threads.

7. Conclusions

Efficient barrier synchronization is critical to data parallel scientific calculations running on SMP systems using multi-core processors. This paper first analyzes two known barrier algorithms: the centralized algorithm and the queue-based algorithm, and identifies sources of performance overhead on a system equipped with two newly introduced AMD quad-core Opteron processors. The significant sources of the overhead come from read/write contention for shared variables so that bus/cache transactions either cannot be carried out in parallel or cannot be fully pipelined. To reduce the performance overhead, this paper introduces a modified algorithm derived from the queue-based algorithm to reduce the read contention during the release phase of a barrier operation.

On our test system, all barrier algorithms perform better than the barrier generated by the Intel OpenMP compiler. The centralized barrier algorithm is known to work well for a small number of processing cores but it contains both write and read contention for shared variables. Our studies show that the queue-based algorithm indeed outperforms the centralized algorithm for two to eight threads because it reduces the write memory contention found in the centralized algorithm. The modified queue-based algorithm performs the same as the original version for two to four threads, but it performs consistently better than the original version for five to eight threads. Especially, the modified algorithm achieves speedup of 1.77 over the original version in the case of eight threads. In addition, the modified algorithm has speedup of 2.39 over the barrier generated by the Intel OpenMP compiler.

As a part of an ongoing software development effort at Jefferson Lab to simplify multi-threading physics calculations on SMP systems based on multi-core processors, the C implementation of the barrier algorithms is freely available at ftp://ftp.jlab.org/hpc/qmt.tar.gz.

8. Acknowledgment

This work is supported by Jefferson Science Associates, LLC under U.S. DOE Contract DE-AC05-06OR23177.

References

[1] AMD: Software Optimization Guide for the AMD Family 10h Processors, 2007.

[2] J. M. Bull and D. O'Neill, A microbenchmark Suite for OpenMP 2.0, In *Proceedings of the European Workshop on OpenMP*, 2001.

[3] L. Cheng and J.B. Carter, Fast Barriers for Scalable ccNUMA Systems, In *Proceedings of the International Conference on Parallel Processing (ICPP'05)*, 241-250, 2005.

[4] D. E. Culler, J. P. Singh and A. Gupta, Parallel Computer Architecture: A Hardware and Software Approach, Morgan Kaufmann, 1999.

[5] HyperTransport Consortium: Low Latency Chip-to-Chip and beyond Interconnect, http://www.hypertransport.org/ .

[6] C. N. Keltcher, K. J. McGrath, A. Ahmed and P. Conway, The AMD Opteron Processor for Multiprocessor Servers, *IEEE Micro*, 23(2), 66-76.

[7] M. Larry and C. Staelin, lmbench: Portable Tools for Performance Analysis, In *Proceedings of the USENIX Technical Conference*, 1996.

[8] J. Mellor-Crummey and M. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Trans. on Computer Systems*, 21-65, 1991.

[9] J. Mellor-Crummey and M. Scott, Synchronization without Contention, In *Proceedings of the Symposium of Architectural Support for Programming Languages and Operating Systems*, 269-278, 1991.

[10] OpenMP Application Program Interface, version 2.5, public draft, 2004

[11] M. Papamarcos and J. Patel, A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories, In *Proceedings of the International Symposium on Computer Architecture*, 238-354, 1984.

[12] L. Spracklen, S. G. Abraham, Chip Multithreading: Opportunities and Challenges, In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 248-252, 2005.

[13] P. Sweazey and A. J. Smith, A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus, In *Proceedings of the International Symposium on Computer Architecture* , 414-423, 1986.

[14] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 1996.

[15] Dual-Core Intel Xeon Processor 5000 Sequence. http://www.intel.com/products/processor/xeon5000/documentation.htm