Multi-Threading Performance on Commodity Multi-core Processors

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

Weizhen Mao
Department of Computer Science
College of William and Mary
Williamsburg, VA 23185
Email: wm@cs.wm.edu

Abstract— **Multi-core processors based commodity servers recently become building blocks for high performance computing Linux clusters. The multi-core processors deliver better performance-to-cost ratios relative to their single-core predecessors through on-chip multi-threading. However, they present challenges in developing high performance multi-threaded code. In this paper we study the performance of different software barrier algorithms on Intel Xeon and AMD Opteron multi-core processor based servers. Especially, we explore how different memory subsystems, such as shared bus or ccNUMA, and their cache coherence protocols effect the performance of barrier algorithms. In addition, we compare multi-threading software overhead between OpenMP directives and a locally developed threading library that utilizes optimized barrier algorithms along with low overhead locking primitives. We find that OpenMP implementations provide high performance run-time libraries coupled with excellent compiler directives with overhead slightly more than the carefully optimized library.**

1. Introduction

Recently, multi-core processors based on the chip multi-threading/processing (CMT/CMP) [16] architecture, which uses multiple single-thread processor core 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 dual-core Xeons [20] and AMD dual-core Opterons [10], 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 running 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. To take advantage of multi-core architecture, applications need to be either rewritten or converted into multi-threaded ones. Currently there are two commonly used strategies to achieve this: OpenMP [14] and Posix Thread (Pthread). OpenMP is a specification designed to support portable implementation of multi-threaded programs for SMP machines. It contains a set of compiler directives and a small set of callable run-time library routines that extend to Fortran, C and C++ to express parallelism in a fork-join programming model. The performance of an OpenMP application is thus compiler dependent. The Pthread library is a portable C library that offers various programming models, but has a large set of APIs and is difficult to program. However, a carefully crafted architectural optimal run-time library with a simple set of APIs that support the fork-join programming model could offer more flexibility, enhance performance of applications and be easy to use.

It is a well-known fact that the choice of a barrier algorithm is critical to the performance of any library that supports the fork-join programming model since each join action leads to executions of the barrier synchronization among all threads. A barrier for a group of threads means that any thread must stop at this point and cannot proceed until all other threads reach this barrier. The performance of a barrier is heavily influenced by the memory subsystem of an SMP machine. Currently there are two commonly used memory architectures in commodity multi-core SMP systems: the shared bus architecture, where each core accesses the memory uniformly through a common bus; and the 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. Even though a threaded application only deals with a uniform and global shared memory address space on an SMP system, the memory subsystem takes care of communications among caches/memories of multiple cores to ensure cache coherence and cache to memory coherence. The memory subsystem usually minimizes unnecessary accesses to the relatively slow main memory by using fast caches in order to improve overall performance. However, reading from and writing to main memory are inevitable because of the limited size of caches and maintenance of cache coherence among multiple cores. The cache coherence is achieved by bus-snooping governed by a cache coherence protocol [4], which influences how the main memory is accessed.

Past research has focused on how to improve barrier performance by either reducing memory contentions introduced by accessing shared flags within a barrier or by reducing the critical path of a barrier [3] [12] [13]. There is a lack of studies on how the memory subsystem, particularly its cache coherence protocol, influences the performance of barrier algorithms. In addition, most well known barrier algorithms target to large SMP machines. This paper compares the performance of a few known software barrier algorithms on small commodity multi-core processor based SMP systems (using Intel Xeons and AMD Opterons) to shed light on the above issues.

Past research has also focused on performance evaluation of OpenMP on different types of processors such as multi-core processors and processors with HyperThreading capability [5]. There have been no detailed comparison studies on the performance between the OpenMP directives and related routines from Pthread based multi-threading libraries. This paper compares software synchronization overhead between the OpenMP directives and the corresponding routines of a hand crafted threading library, called QCD Multi-Thread (QMT) for the multi-threaded LQCD (Lattice Quantum Chromodynamics) applications, which utilizes an optimized barrier algorithm and low overhead locking primitives to address issues such as different ways to multi-thread scientific applications. We show that some OpenMP implementation provide a high performance run-time library coupled with excellent compiler directives with overhead only slightly more than the overhead introduced by the QMT library routines.

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 organizations and cache coherence protocols deployed by the commodity multi-core SMP systems. Section 4 presents two barrier algorithms. Section 5 analyzes the memory/cache transactions of the algorithms under different memory architectures utilized by our test machines. Section 6 illustrates how memory subsystems and their cache coherence protocols effect the performance of the barrier algorithms. Section 7 compares synchronization overhead between the OpenMP directives and the corresponding QMT library routines. Section 8 concludes.

2. Hardware and Software Environment

We have chosen the Dell PowerEdge 1850 with two Intel Xeon dual-core processors and the Dell PowerEdge SC1435 with two AMD Opteron dual-core processors as test beds. In this section, we describe these two machines, benchmarks that we ran, and how to execute the benchmarks.

The Dell PowerEdge 1850 has two Intel Xeon 5150 [20] 2.66 GHz dual-core processors.The memory system is organized as a shared bus system, where each core accesses to the FB-DIMM [6] memory through a common memory controller. The Dell PowerEdge SC1435 hosts two AMD Opteron 2220SE 2.8 GHz dual-core processors. Each processor has its own memory controller that creates a path to its own locally attached memory. A processor can access to the memory or the L2 cache of the other processor through an interconnect called HyperTransport [8]. Thus, the system is a ccNUMA architecture because of the non-uniform memory access times for local and remote memory. From here on, we refer to the two test machines as Intel and AMD, respectively. Table 1 summarizes the configurations of the two systems in detail.

Table 1: Configuration of Test Machines

	CPUs	L1	L2	Memory
Intel	Two 2.66GHz Dual-Core	32K Data 32K Instr	4 MB Shared	4GB Shared Bus
AMD	Two 2.8 GHz Dual-Core	64K Data 64K Instr	1 MB Private	4GB ccNUMA

QMT is a light weight library developed at Jefferson Lab to ease the effort of multi-threading LQCD applications running on multi-core processors. It offers a very simple set of APIs that support the fork-join OpenMP programming model, and provides some of the OpenMP capabilities such as barrier synchronization and global reduction. It utilizes an optimal barrier algorithm for either Intel or AMD machine and is portable on any platform where Pthread is available.

Both test machines are running Fedora Core 5 Linux x86_64 distribution with a Linux kernel of 2.6.17. Two compilers are utilized: the gcc version 4.1 and the Intel icc version 9.1. The synchronization overhead introduced by the OpenMP directives are measured through the EPCC microbenchmark [2]. On the other hand, the synchronization overhead induced by software barriers and other QMT library routines are collected through a slightly modified EPCC microbenchmark program. In addition, the Linux kernel is patched to support Performance API (PAPI) [1] version 3.5, which is used to collect performance events through the event counters on the processors of the test machines.

3. Memory and Cache Coherence Protocols

In a multi-core SMP system, the memory system is organized in a hierarchical way including fast multi-level of caches and relatively slow memory. Each core usually has its own private L1 cache, but it can either has a private L2 cache such as on the AMD test machine or has a shared L2 cache with the other cores within a single CPU in the case of the Intel test machine. Multiple caches at each level are allowed to have copies simultaneously of a given memory location. A cache coherence protocol is a mechanism to ensure that all copies remain consistent when the contents of that memory location are modified. On the Intel test machine with the shared bus memory architecture, cache coherence is maintained by having all cache controllers “snoop” on the memory bus and monitor the transactions. A snooping cache controller may take an appropriate action according to the coherence protocol if a bus transaction is relevant to it. On the AMD test machine with the ccNUMA architecture, a bus snooping is simulated by explicit broadcasts of memory requests to all cores [10].

One type of popular cache coherence protocols for small SMP systems with write-back caches is the invalidation-based

protocol: a snooping cache invalidates its cached copy on a relevant write by another core. The cache coherence protocol deployed by the Intel Xeons is the MESI protocol [15], named from the four states of the protocol: Modified, Exclusive, Shared, Invalid. For an application using multiple cores across multiple CPUs, writing to an invalid cache block or writing to a shared cache block (write miss) results in a read-exclusive bus transaction, which causes the other copies of the block to be invalidated. If a cache block is in the Modified state, the block has to be written back to the main memory so that another core can load correct value from the memory upon accessing to its own invalid cache block. This certainly introduces extra overhead because of the relatively high latency of accessing main memory. However, for an application only involving cores that share a single cache, a modified cache block may not be written back to the memory. Hence, the shared large L2 cache for the dual-core Intel test machine enables fast cache-to-cache communication for applications using only two threads within a single CPU.

To address the problem of writing a modified cache block to the main memory to maintain cache coherence, the modified cache coherence protocol called MOESI [18], which adds an Owner state as the fifth state, is used by the AMD Opterons. A cached block in the Owner state holds the most recent correct copy of the data. Unlike the Shared state, the copy in the main memory can be stale. Only one cache can hold a block of data in the Owner 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 the block. This avoids the need to write a modified cache back to the main memory. Data flagged as in the Owner state in an Opteron cache can be delivered directly from the cache of one CPU either to another CPU via a CPU-to-CPU HyperTransport link or to other caches on the same CPU via the system request interface (SRI) [17].

Traditionally, a simple barrier can be implemented by having each thread increment/decrement one or more barrier count variables. The barrier completion is signaled by a release flag that each thread checks in a spin loop until it shows that all threads have arrived at the barrier. Hence the performance of accessing to 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 machines using the LMbench [11] benchmark.

Table 2: Random Memory Access Latency

	L1(ns)	L2(ns)	Memory(ns)
Intel	1.13	5.29	150.3
AMD	1.07	4.3	173

Cache to cache transfer latency is also important to multi-threaded applications on a multi-core SMP machine. An AMD Opteron based SMP machine has two ways to achieve this: the SRI channels for transferring data among L2 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 SRI 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 AMD test machine shows an SRI cache-to-cache latency of 83.5 ns and a HyperTransport cache-to-cache latency of 119 ns. On the other hand, running the same program on the Intel test machine shows the cache-to-cache latency of 55 ns for two cores within one CPU and 187 ns for two cores between two CPUs. This is due to the shared L2 cache for the two cores within a CPU and the cache coherence dictated by MESI protocol through the system memory for cores across multiple CPUs.

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. 1 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.

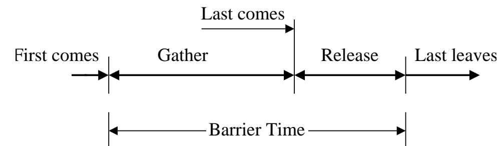

Fig. 1: Timing for a Single Barrier

There are a few popular barrier algorithms used over the years, such as the centralized barrier, the combining tree barrier, the tournament barrier [12] 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.

The implementation of the centralized barrier (given below) uses one shared counter variable and one hared release flag.

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

To avoid contention to 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. Each thread participating in the barrier operation signals its arrival at the barrier by writing to its flag variable in the flag array, and then spins on the separate 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 and to update the release flag for which non-master threads are waiting.

```
typedef struct qmt_cflag {
   int volatile c_flag;
   /* each flag on different cache line */
   int c_pad[CACHE_LINE_SIZE - 1];
}qmt_flag_t;
typedef struct qmt_barrier {
   int volatile release;
   char br_pad[B_CACHE_SIZE - 1];
   qmt_flag_t flags[1];
}qmt_barrier_t;
/* Master Thread */
for (i = 1; i < num_threads; i++) {
   while (barrier→flags[i].cflag == 0);
   barrier→flags[i].cflag = 0;}
atomic_inc (barrier→release);
/* Thread i */
int rkey = barrier→release;
barrier→flags[i].cflag = -1;
while (rkey == barrier→release);
```

5. Analysis of Barrier Algorithms

To find out how the memory architecture of an SMP system effects the performance of a barrier algorithm, we analyze memory and cache transactions of two barrier algorithms under the MESI and the MOESI cache coherence protocols.

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 depending on the cache coherence protocol of a system. The difference between the centralized algorithm and the queue-based algorithm lies in the gather phase of each algorithm. The release phase is rather similar. 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. MESI Protocol on the Intel Test Machine

A centralized barrier employs one counter variable and one release variable. Initially, the counter variable is the same as the number of cores and the release variable is set to be zero. Let us assume that one core initially holds valid cached values of the counter variable and the release variable, and that the other caches are all invalidated. Furthermore, all threads are assumed to arrive at the barrier at about the same time. Therefore, all threads contend for the counter variable because each thread attempts to perform the atomic_dec() instruction on the counter variable when it arrives at the barrier. Updating the counter variable inside an invalid cache block causes an RdX bus transaction. Since only one RdX bus transaction on the same cache line can be granted, one core will be allowed to update the counter variable. The left part of Fig. 2 is a snapshot of the bus and the memory transactions for thread 2 arriving at the barrier, where the numbers inside the small circles denote the sequence of the transactions. Thread 2 is being granted the bus while thread 1 is holding the updated/valid cache value of the counter variable. When thread 2 generates an RdX bus transaction, thread 1 has to flush out the counter variable in its cache to the main memory and changes the cache block into an invalid state. Thread 2 then has to load the updated counter variable from the main memory, to decrease the value by one and to put the cache block in a modified state. A Rd bus transaction is generated when thread 1 compares the counter variable to zero, which leads thread 2 to write the modified value of the variable in its cache to the system memory. Finally thread 1 reads the updated counter variable from the memory. There is a total of 4 memory reads/writes and two bus transactions for thread 2 to complete signaling the arrival at the barrier. Similar analysis leads to the same conclusion for each of the other threads. At the end of the gather phase of the barrier, there are no memory reads/writes when the last thread is setting the counter variable to be the same as the number of cores because the thread is holding the updated value of the counter variable and the other threads are not reading the variable. In the release phase of the barrier, there are $n-1$ memory reads and one single memory write because one thread writes the modified cache value of the release flag to the main memory while the other threads read the value from the memory. Hence the total number of memory accesses during a centralized barrier operation is $4n+(n-1)+1=5n$. 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.

Unlike the centralized barrier algorithm, the queue-based algorithm does not use a shared counter variable. Instead, it designates one thread as the *master* and allocates an array of flags, one per participating thread. Each thread signals its arrival at a barrier by updating its flag in the array. The right part of Fig. 2 illustrates the memory and the bus transactions for one particular thread, e.g. thread 2, arriving at the barrier. Initially, the master thread holds all flags exclusively. When thread 2 updates its flag, an RdX bus transaction is generated. This bus transaction prompts the master thread to flush the cached copy of the flag to the memory and thread 2 reads the updated value of the flag from the memory, which leads to two memory transactions. When the master thread is checking the flag, it generates an Rd bus transaction which requests thread 2 to flush the modified value of the flag to the memory. This yields two more memory transactions. This analysis applies to the other non-master threads as well. There are no memory

writes at the end of the gather phase of the barrier when the master thread resets all flags because the master thread has the updated flag values. Similar to the centralized algorithm, the number of memory accesses during the release phase of the barrier is n. Hence the total number of memory accesses during a queue-based barrier operation is $4(n-1)+n = 5n-4$. It is clear that a queue-based barrier operation generates fewer memory transactions than a centralized barrier does. More importantly, updating each flag in the global flag array of a queue-based barrier can be carried out in parallel since each flag resides on a different cache line. These two factors lead to a better performance for the queue-based algorithm.

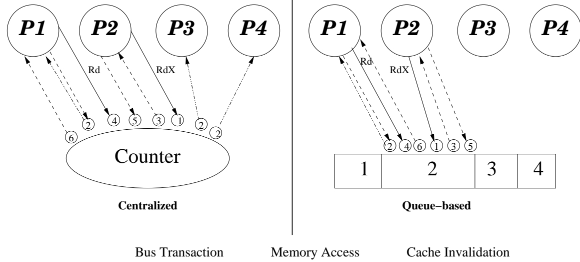

Fig. 2: Bus/Memory Transactions

Finally, two threads running on two cores within a single CPU exchange data without going through the main memory since the Intel Xeon utilizes a 4MB shared L2 cache for the two cores. Hence the above analysis of the algorithms can only be applied to the cases of more than two threads. The two algorithms perform similarly when two threads are involved. Table 3 summarizes the number of memory transactions of the algorithms and the ratio of memory transactions between the centralized algorithm and the queue-based algorithm.

Table 3: Memory Transactions on Intel

Threads	Centralized	Queuebased	Ratio
2	N/A	N/A	1.00
3	15	11	1.36
4	20	16	1.25

5.2. MOESI Protocol on the AMD Test Machine

The centralized barrier algorithm under the MOESI protocol behaves similar to what it does under the MESI protocol except that there could be no main memory transactions, which are substituted by cache-to-cache transactions, if cache evictions can be avoided. Once again 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. The left part of Fig. 3 demonstrates the bus and the cache transactions for thread 2 arriving at a centralized barrier. Under the MESI protocol on the Intel test machine, thread 1 has to flush its cached counter variable value to the main memory in order for thread 2 to load the correct value. 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. Finally, the number of cache invalidations in the gather phase can be divided into two parts: n owner cache invalidations, each from a thread invalidating its own copy of the counter variable after updating another thread; and a single invalidation on the counter variable of each one of the $n-1$ threads when the last thread changes the counter variable. The number of cache invalidations in the release phase is simply $n-1$ because $n-1$ cached values of the release flag are invalidated when one of the threads modifies the release flag. Hence the total number of cache invalidations during the synchronization of the barrier is $3n-2$.

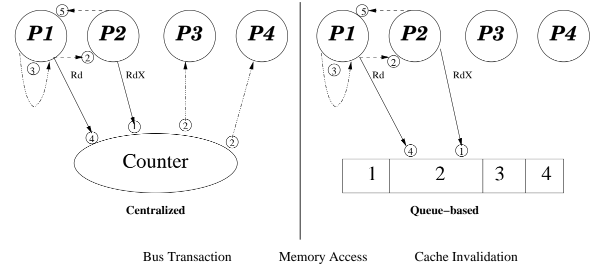

Fig. 3: Bus/Cache Transactions

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. The right part of Fig. 3 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 changes the value of the flag to be -1. 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. At the end of the gather phase, the master thread resets each signal flag to be zero, and becomes the owner of each signal flag again without introducing any cache transaction because it already has updated values of the signal flags. 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. Finally, the number of cache invalidations

is $2(n-1)$ during the gather phase of the barrier because the master thread invalidates its own $n-1$ signal flags first and then invalidates signal flag in the other $n-1$ threads. Using the same discussion as in the previous paragraph, the number of cache invalidations is $n-1$ during the release phase of the barrier. Hence the number of cache invalidations during the synchronization of the barrier is $3n-3$.

Even though the memory transactions could be avoided for both the centralized and the queue-based algorithms under the MOESI protocol on the AMD test machine, the RdX bus transactions always exist. They are difficult to be carried out in parallel for the centralized barrier algorithm because of the shared counter variable, but the queue-based barrier algorithm has the advantage in this respect. Nonetheless, the lack of access to the main memory in either type of barrier and a small difference in the number of cache transactions between these two types of barriers suggest that the centralized barrier algorithm may perform reasonably well for the small number of threads. Table 4 summarizes the number of cache transactions of the algorithms and the ratio of cache transactions between the centralized algorithm and the queue-based algorithm.

Table 4: Cache Transactions on AMD

Threads	Centralized	Queuebased	Ratio
2	5	3	1.67
3	8	6	1.33
4	11	9	1.22

Finally, there are two different cache-to-cache transactions during a barrier operation for each of the two algorithms. One is the cache-to-cache transactions between the two cores within a single CPU via the SRI channels and the other is the cache-to-cache transactions across physical boundary of CPUs through the HyperTransport channels. A HyperTransport transaction clearly takes longer time than an SRI transaction. For instance, a HyperTransport channel takes 35.5 more nanoseconds than an SRI channel to transfer a single integer on the AMD test machine. Therefore the performance of the algorithms can be effected not only by the number of cache transactions but also by the type of transactions.

6. Performance of the Barrier Algorithms

In this section, we use the software overhead of a barrier algorithm as the performance metric of the barrier algorithm. The software overhead values of these two barrier algorithms are collected using the modified EPCC microbenchmark program. To understand the performance of the barrier algorithms, the performance event counters on the Xeon and the Opteron processors are utilized through the PAPI library routines to quantify the relation between the number the memory/cache transactions and the performance of a barrier. Fig. 4 shows the overhead values in terms of CPU cycles for the centralized and the queue-based algorithms on each of the test machines.

On the Intel test machine, the queue-based algorithm performs essentially the same as the centralized algorithm for two threads because there are no memory transactions due to the shared L2 cache for the two cores within a single CPU. However, the queue-based algorithm performs much better as expected than the centralized algorithm does for three and four threads because a queue-based barrier has fewer memory transactions thus reduces memory contentions. To verify the above observations, the number of memory transactions is collected during many consecutive loops for each of the barrier algorithms using the PAPI routines with the native event of BUS_TRANS_MEM. The left part of Fig. 5 presents two ratios between a centralized barrier and a queue-based barrier for two to four threads. They are the software overhead values and the number of memory transactions collected by the PAPI event. The centralized barrier clearly has more memory transactions than the queue-based barrier, which agrees with our analysis. The software overhead directly relates to the the number of memory transactions. However, the number of memory transactions does not reveal the complete picture of the barrier overhead. The serialization of the bus transactions due to the memory contention on the same shared cache block of the counter variable in the centralized barrier algorithm contributes to the additional overhead of the algorithm. The smaller ratio of the number of memory transactions relative to the ratio of the overhead values for four threads reveals the contribution from the memory contention to the centralized barrier overhead.

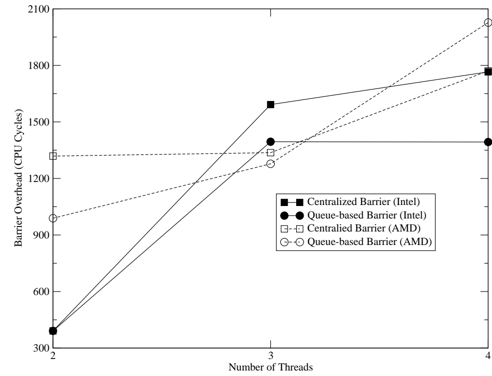

Fig. 4: Performance of Barrier Algorithms

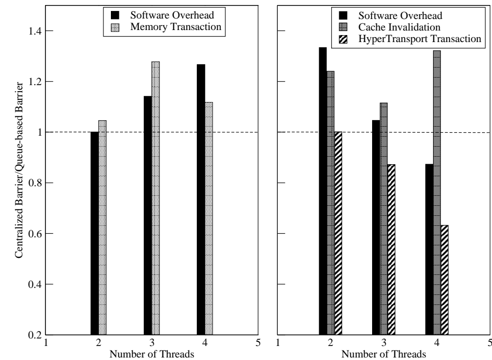

Fig. 5: Centralized vs. Queue-based Barriers

On the AMD machine where a barrier utilizes fast cache-to-cache transactions, the queue-based algorithm performs differently from the centralized algorithm for two threads due to the private L2 cache for each of the two cores within a single CPU. The queue-based algorithm performs better

than the centralized algorithm for two and three threads because it has fewer cache-to-cache transactions thus reduces the contention. But the queue-based barrier actually produces more overhead than the centralized barrier for four threads even though the queue-based barrier generates fewer cache transactions. Hence the number of cache transactions is not the determining factor for the performance of the barriers. It is entirely possible for the queue-based barrier to have more HyperTransport cache-to-cache transactions than the centralized barrier when four threads are involved. In order to substantiate the hypothesis, the number of HyperTransport transactions is collected for many consecutive barrier synchronization loops for each of the barrier algorithms through the PAPI event of NB_HT_BUS(x)_DATA where x stands for the transport buses 0, 1 and 2. In addition, the number of cache invalidations is collected during the same loops because there is no direct performance event monitoring the number of cache transactions. The ratio of the cache invalidations between a centralized barrier and a queue-based barrier is $\frac{3n-2}{3n-3}$ which approximately equals to the ratio of the cache transactions between the two barriers, $\frac{3n-1}{3n-3}$. Hence the PAPI routines with the native events called DC_COPYBACK_I, which track the number cache invalidations, are utilized. The right part of Fig. 5 shows various ratios between a centralized barrier and a queue-based barrier. They are the software overhead values, the number of cache invalidations, and the number of cache-to-cache transfers through the HyperTransport channels. For two and three threads, the software overhead ratio directly relates to the cache transaction ratio, which indicates that the cache transaction is the dominating factor of the performance of the barriers. For four threads, however, the software overhead of the centralized barrier is smaller than that of the queue-based barrier along with a higher number of cache transactions but a smaller number of HyperTransport transactions. This suggests that the number of HyperTransport transactions becomes the prominent factor in determining the performance of the barrier algorithms on the AMD machine. In short, the performance of a barrier algorithm is determined by two competing factors: the number of cache/bus transactions and the percentage of cache transactions that are the HyperTransport transactions.

Finally, the same barrier algorithm has different performance characteristics on different machines because of the memory architectures and the cache coherence protocols. The queue-based barrier algorithm performs consistently better than the centralized algorithm for two to four threads on the Intel machine, because the queue-based algorithm removes the memory contention on the shared cache block and has a smaller number of memory transactions. On the AMD machine, the queue-based algorithm performs better than the centralized algorithm for two and three threads because the queue-based algorithm removes the contention on the same cache block and has a fewer cache transactions. When four threads are involved, the queue-based algorithm actually performs a little worse than the centralized barrier algorithm, because it introduces more HyperTransport cache transactions. In the case of two threads, the two algorithms all perform better on the Intel machine than they do on the AMD machine because of the large shared L2 cache on the Intel machine.

7. Overheads in OpenMP and QMT

In OpenMP, most of the synchronizations including barrier operations are realized by the compiler directives and their overhead depends on the implementation in the vendor supplied OpenMP run-time library. In the QMT library we developed, however, the overhead can be minimized by optimal implementation tailored to a specific architecture. In particular, an optimized barrier algorithm is chosen at run-time according to the study in the previous sections. To compare the overhead values of the OpenMP directives from different OpenMP compilers, the EPCC microbenchmark code is compiled on each of the test machines using the Intel icc and the GNU gcc compilers. The benchmark program is executed on each of the test machines using one to four threads. The overhead values of the OpenMP directives from the Intel icc are much less than that of the OpenMP directives from the GNU gcc for all synchronization mechanisms. To highlight the above observation, Table 5 shows the overhead values in terms of CPU cycles for the OpenMP directive of the lock/unlock overhead on the test machines in the case of 4 threads. The large lock/unlock overhead difference between gcc and icc is not a surprise because gcc implements the OpenMP lock using the Linux futex [7] system call, which incurs a lot of overhead, in comparison to the user level lock deployed by the icc compiler [19]. From now on, the gcc OpenMP is no longer under consideration.

Table 5: Lock/Unlock Overhead CPU Cycles

	Intel	AMD
icc	551	979
gcc	4503	4608

To measure the synchronization overhead induced by the QMT library routines, the two compilers are used to compile a benchmark code slightly modified from the EPCC code. The new benchmark values are collected for one to four threads. Especially, the $taskset$ command is used to bind two threads to the two cores on the same CPU when the benchmark program is executed using two threads. The left part of Fig. 6 shows the results of executing the two benchmarks on the Intel test machine. The overhead value of every OpenMP directive is slightly larger than that of the corresponding QMT routine. Every overhead scales close to linearly from one thread to four threads. Moreover, the overhead values from either the OpenMP directives or from the QMT routines show some sensitivity to the change from running two to three threads because of the shared L2 cache of the two cores within a single CPU. Both the icc OpenMP implementation and the QMT library offer very low lock/unlock overhead because of the light weight user level lock mechanism.

The right part of Fig. 6 presents the results of the two benchmarks on the AMD test machine. The overhead of each of the icc OpenMP directives exhibits similar behavior as in the previous figure even though it grows slightly larger

than the corresponding value for the Intel machine. This result comes at no surprise because the Intel icc is optimized for Intel architecture after all. However, the results of the overhead for the QMT routines bear remarkable similarity to the corresponding results collected on the Intel machine because the QMT library is implemented optimally for either the Intel or the AMD architecture.

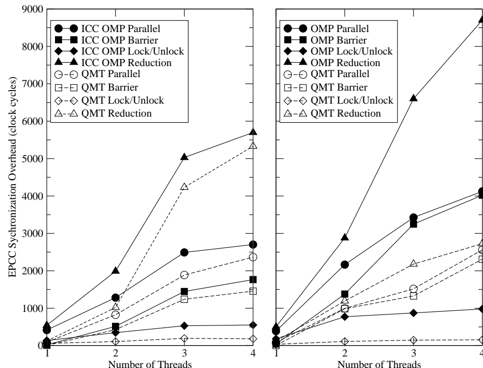

Fig. 6: OpenMP and QMT Overheads

Our studies indicate that the synchronization overhead for the icc OpenMP directives scales almost linearly in spite of its slightly larger value than the overhead value of the corresponding QMT routine. Therefore applications that seek portability over absolute performance will benefit from a well implemented OpenMP compiler and its runtime library. On the other hand, applications that look for performance and flexibility will profit from a hand craft Pthread library such as the QMT library.

8. CONCLUSIONS

This paper studies the performance of two barrier algorithms on commodity multi-core based SMP machines. The centralized barrier algorithm is known to work well for a small number of processing cores, and the queue-based barrier algorithm tries to reduce memory contention introduced by the centralized algorithm. Our study shows that the queue-based algorithm indeed outperforms the centralized algorithm on the Intel shared bus memory architecture for threads ranging from two to four simply because it has little memory contention and fewer accesses to the main memory dictated by the MESI cache coherence protocol. On the AMD platform, with the ccNUMA architecture and the MOESI cache coherence protocol, the queue-based algorithm performs better than the centralized algorithm for two and three threads, but it performs no better for four threads. The behavior is caused by the MOESI protocol, which maintains the cache coherence through the cache-to-cache transactions via either the SRI channels or the slower HyperTransport channels instead of accessing the main memory. In comparison to the centralized algorithm, the queue-based algorithm has fewer cache-to-cache transactions on average for threads from two to four, but it has more HyperTransport cache-to-cache transactions. Therefore the performance of a barrier algorithm can be influenced not only by the memory organization but also by the cache coherence protocol of an SMP system.

This paper also compares the software overhead of the OpenMP directives from the Intel icc compiler against a locally developed Pthread library that supports the fork-join programming model. Even though the OpenMP directives produce very little overhead and perform well, the local library still comes ahead for all measured benchmarks by small margins. Therefore, applications that seek portability and simplicity over absolute performance will benefit from OpenMP. However, a carefully crafted Pthread library will find a place for those who seek performance and flexibility.

ACKNOWLEDGMENT

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

REFERENCES

[1] S. Browne, C. Deane, G. Ho, and P. Mucci, PAPI: A Portable Interface to Hardware Performance Counters, In *Proceedings of DOD HPCMP Users Group Conference*, 1999.

[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 Approatch, Morgan Kaufmann, 1999.

[5] M. Curtis-Maury, X. Ding, C. Antonopoulos and D. S. Nikolopoulos, An Evaluation of OpenMP on Current and Emerging Multi-threaded/Multicore Processors, In *Proceedings of the International Workshop on OpenMP*, 2005.

[6] Fully-Buffered DIMM Technology Moves Enterprise Platforms to the Next Level,http://www.intel.com/technology/magazine/computing/Fully-buffered-DIMM-0305.htm

[7] H. Franke and R. Russel, Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux, In *Proceedings of the Ottawa Linux Symposium*, 2002.

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

[9] M. Johnson, Superscaler Microprocessor Design, Prentice Hall, 1991.

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

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

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

[13] 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.

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

[15] 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.

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

[17] AMD: BIOS and Kernel Developer's Guide for the AMD Athlon 64 and AMD Opteron Processors, 143, 2006.

[18] 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.

[19] X. Tian and M. Girkar, Effect of Optimization of Performance of OpenMP Programs, In *Proceedings of the International Conference on High Performance Computing*, 133-143, 2004.

[20] Dual-Core Intel Xeon Processor 5000 Sequence. http://www.intel.com/products/processor/xeon/index.htm#5000 .