Assessment of Barrier Implementations for Fine-Grain Parallel Regions on Current Multi-core Architectures

Simon A. Berger and Alexandros Stamatakis
The Exelixis Lab
Department of Computer Science
Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
Email: bergers@in.tum.de, stamatak@in.tum.de
WWW: http://wwwkramer.in.tum.de/exelixis/

***Abstract*—Barrier performance for synchronizing threads on current multi-core systems can be critical for scientific applications that traverse a large number of relatively small parallel regions, that is, that exhibit an unfavorable computation to synchronization ratio. By means of a synthetic and a real-world benchmark we assess 4 alternative barrier implementations on 7 current multi-core systems with 2 up to 32 cores. We find that, barrier performance is application- and data-specific with respect to cache utilization, but that a rather naïve lock-free barrier implementation yields good results across all applications and multi-core systems tested. We also assess distinct implementations of reduction operations that are computed in conjunction with the barriers. The synthetic and real-world benchmarks are made available as open-source code for further testing.**

***Keywords*-barriers; multi-cores; threads; RAxML**

I. Introduction

The performance of barriers for synchronizing threads on modern general-purpose multi-core systems is of vital importance for the efficiency of scientific codes. Barrier performance can become critical, if a scientific code exhibits a high number of relatively small (with respect to the computations conducted) parallel regions, for instance, parallelizations of `for`-loops over vectors, that are implemented based upon the fork-join paradigm. Deploying the "classic" fork-join paradigm (see Figure 1), that is, explicitly creating and joining threads for every parallel region is usually highly inefficient. To this end, a pool of threads is used (typically one will use as many threads as there are physical cores available). Those threads will be created only once at program initialization and joined once when the program terminates. One thread will assume the role of the master thread for executing the sequential parts of the code, while the remaining threads will usually conduct a busy wait (also denoted as spin-lock) until a parallel region is reached.

Here, we consider the case where parallel regions are triggered by the master thread who sets a variable on which worker threads wait busily and assess the performance of alternative barrier implementations for synchronizing the threads at the end of such parallel regions (see Figure 2).

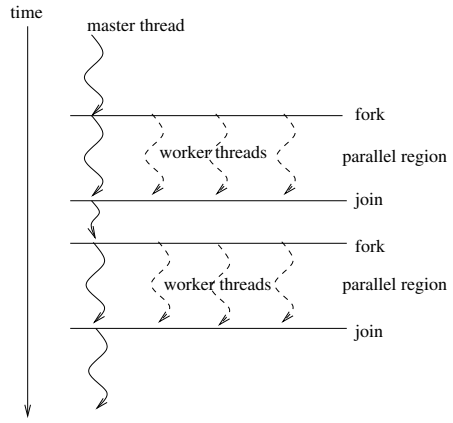

Figure 1. Classic fork-join paradigm.

In addition, we analyze the efficient implementation of reduction operations (sums over the double values produced by each `for`-loop iteration), that are frequently required in conjunction with barriers. For reduction operations, we impose the restriction that, results of reductions need to be deterministic, that is, the order in which the per-thread (per-core) partial sums are reduced (added in our case) needs to be always identical. For instance, four partial sums from four threads, t_0, t_1, t_2, t_3 need to always be summed in the same order by using a tree-based order $(t_0+t_1)+(t_2+t_3)$ or some other arbitrary, but fixed order, for example, $(t_0+(t_1+(t_2+t_3))$. Note that, the OpenMP and MPI standards do not explicitly enforce a fixed (deterministic) order of additions for reduction operations. This means that, when applying reductions to floating point numbers, the sum across the exact same four numerical values of t_0, t_1, t_2, t_3 may yield different results. Non-determinism in reduction operations may lead to extremely hard-to-detect numerical bugs in certain applications as experienced with OpenMP [1].

We test the speed of up to 8 alternative barrier implementations and combined barrier/reduction implementations using a synthetic workload on 7 and a real Bioinformatics

workload on 4 current multi-core systems with 2 up to 32 cores.

Quite surprisingly, for the real workload, we find that, a relatively naïve implementation which does not take into account false sharing by padding and that computes reductions at the master only using SSE3 instructions, provides the best performance across all tested systems.

For the synthetic workload we created a toy test program. In the parallel regions, the threads carry out dummy calculations on -point arrays. The size of those arrays, that is, the amount of computations in the parallel regions can be specified by the user via an appropriate parameter. When no computations are conducted (the array size is set to zero), the synthetic workload can be used to measure the stand-alone execution times of the different barrier implementations. The results obtained for the experiments with the synthetic workload, partially contradict the results obtained for the real Bioinformatics workload. This suggests that, in a real-world scenario, properties such as the cache-efficiency of the barrier and the data access patterns of the application, have a significant impact on barrier performance. Hence, the most efficient barrier implementation needs to be determined in an application-dependent way. Nonetheless, we find that, as for the real-world workload, a naïve and portable lock-free barrier, yields acceptable synchronization performance and represents a good initial implementation choice.

The open-source code for the synthetic and real-world benchmarks can be downloaded at http://wwwkramer.in.tum.de/exelixis/barriers.tar.bz2.

The remainder of this paper is organized as follows: In Section II we briefly survey related work on performance analysis of barriers. In Section III we provide a description of the various barrier implementations we have tested. In the subsequent Sections we described the synthetic (Section IV) and real Bioinformatics (Section V) workloads we used. Section VI comprises a description of the test systems used and the experimental results. We conclude in Section VII.

II. Related Work

There are comparatively few related articles that assess implementations of barrier mechanisms on current multi-core systems.

Berrendorf *et al.* [2] evaluated the performance of OpenMP barrier implementations on vector-based parallel systems as well as on ccNUMA (cache coherent Non-Uniform Memory Access) SMP (Symmetric Multi-Processing) systems. On ccNUMA systems, they observed that, data placement and processor geometry are important factors for barrier performance. In our barrier implementations, we therefore use explicit thread pinning and memory allocation mechanisms to enforce data locality as well as appropriate thread to core pinnings, particularly on multi-socket systems.

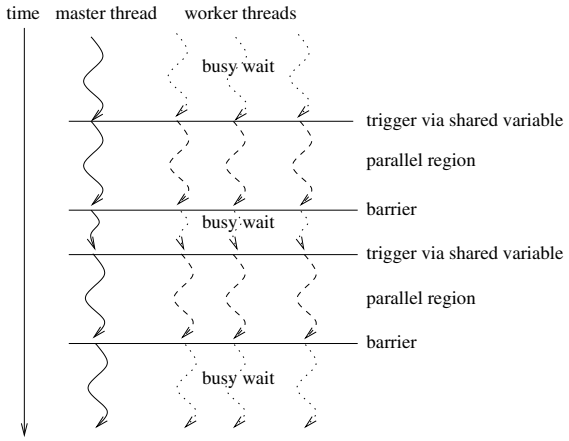

Figure 2. Fork-join paradigm with a thread pool and busy-wait.

Unfortunately, current multi-core systems, do not offer dedicated hardware support for barriers. To this end, barrier implementations must either use a flavor of shared memory-based mechanisms in combination with busy-waiting, or must resort to OS (operating system) specific mechanisms. Generally, the high runtime requirements of OS-specific mechanisms preclude their use for frequent, fine-grain synchronizations. Sampson *et al.* [3] show that, barrier performance for fine-grain synchronization can potentially be improved by introducing additional logic (HW) for enhancing cache implementations on current CPUs and thereby improve barrier performance. Along the same lines, Zhu *et al.* [4] advocate the introduction of specialized memory buffers for achieving performance improvements.

Chen *et al.* [5] have conducted comparisons between centralized barriers (similar to our lock-free barriers; see Section III-A) and queue-based barriers on shared bus UMA (Uniform Memory Access) and on ccNUMA SMP systems with up to 4 cores. The queue-based barriers performed better than centralized barriers on older—currently almost obsolete—bus-based SMP systems. However, on ccNUMA systems, that also allow for a higher number of threads, the centralized barriers performed better [5] which is in line with our results.

Here, we concentrate on different alternative implementations of centralized barrier schemes. All multi-socket systems we have assessed are ccNUMA systems. Currently, shared front-side bus SMP systems with comparable core counts are not produced any more.

Finally, Franz Franchetti has assessed barrier implementations on smaller multi-core systems with less cores and has made available the benchmark source code as well as some experimental results at http://www.spiral.net/software/barrier.html. This benchmark was developed within the framework of the SPIRAL project (see, e.g., [6]) for automated code generation for digital signal processing algorithms. The fastest barrier implementations are conceptually

similar to our barriers and the effect of padding has also been analyzed and yielded comparable results. However, this benchmark exclusively measures stand-alone barrier performance, that is, it does not take into account the effect of real workloads and associated data access patterns and requirements.

We are not aware of any comparative study on barrier performance using a large variety of barrier flavors and current multi-core systems.

III. Barrier Implementations

As already mentioned, we consider an application scenario with a huge number of parallel regions that are triggered by a master thread (thread # 0) that also executes the sequential parts of the code. All threads (using Pthreads) are created once at program initialization, and are joined once at program termination. The parallel regions are triggered by the master thread using a call to the `masterBarrier(...)` function that operates as indicated below[1]. The function `masterSync(...)` is used to wait for all threads to finish whatever workload is executed in `executeWork(...)`.

```
volatile int jobCycle = 0;

void masterBarrier(int tid, int n)
{
   jobCycle = !jobCycle;
   executeWork(tid, n);
   masterSync(tid, n);
}
```

The worker threads operate as follows: They wait for the master thread to trigger a parallel region by changing the value of `jobCycle`.Thereafter, they will execute their work (`executeWork(...)`) and subsequently enter the barrier `workerSync(...)` to notify the master thread, that the computation of the parallel region has been completed by the thread with the number `tid`.

```
void workerThread(int tid, int n)
{
   int mycycle = 0;
   while(1)
     {
        while(myCycle == jobCycle);
        myCycle = jobCycle;
        executeWork(tid, n);
        workerSync(tid, n);
     }
}
```

We explicitly distinguish between the barrier calls at the master (`masterSync(...)`), and at the workers (`workerSync(...)`), where `tid` is the thread ID and `n` the total number of threads, because, depending on the barrier implementation that is deployed, the implementations on the master- and the worker-side may need to be different.

[1]We explicitly use C syntax instead of pseudo-code for easier reproducibility of our results

The shared arrays that are used for synchronization are all aligned to 64 byte boundaries. For those implementations where padding was used to assess the impact of false sharing, we deployed 64 byte offsets for padding.

A. Lock-Free Barrier

The lock-free barrier essentially uses a shared `char` array (`barrierBuffer[]`) that has as many entries as there are threads. Hence, the worker threads only need to set their corresponding entry to 1 in `workerSync(...)`, that is, `barrierBuffer[tid] = 1`, once they have completed their computations.

The master thread simply needs to loop over the `barrierBuffer[]` until all worker threads have arrived at the barrier:

```
void masterSync(int tid, int n)
{
  int i, sum;
  do
    {
      for(i = 1, sum = 1; i < n; i++)
        sum += barrierBuffer[i];
    }
  while(sum < n);

  ....
}
```

and thereafter set the `barrierBuffer[]` entries to `0` again (alternatively, one can alternate between waiting for all entries to be set to zero or one). We can design a padded version of this lock-free barrier by multiplying the indices `i` and `tid` with the respective byte offset of 64 (note that, we are using a char array here).

B. Recursive Lock-Free Barrier

The recursive lock-free barrier works in an analogous way as the lock-free barrier via shared `char` arrays. The only difference is that we use a "classic" reduction tree with $\lceil \log_2(n) \rceil$ steps, where n is the total number of threads. Given $n = 4$, thread 0 will initially wait for thread 1 to finish (spin-lock on `barrierBuffer[1]`), and, at the same time, thread 2 will wait for thread 3 to finish (spin-lock on `barrierBuffer[3]`). Thereafter, thread 0 will wait for thread 2 to finish (spin-lock on `barrierBuffer[2]`). When thread 0 is done, the barrier has been reached by all threads (source code not shown).

We have also implemented a padded version of the recursive lock-free barrier.

C. Intrinsic Atomic Increment Barrier

Another alternative for implementing barriers is to use the x86 intrinsic atomic increment operation `__sync_fetch_and_add()` that was available on all test systems.

```
volatile int counter = 0;
```

```
void masterSync(int tid, int n)
{
  int workers = n - 1;

  while(counter != workers);
  counter = 0;
}
```

The `workerSync(...)` functions then simply need to call `__sync_fetch_and_add( &counter, 1)` to atomically increment `counter` by one.

D. Lock-based Barrier

The lock-based barrier works in an analogous way as the intrinsic atomic increment barrier. The `masterSync(...)` function is exactly identical. In the respective `workerSync(...)` functions, we protect the incrementation of `counter` by a Pthread mutex variable as indicated below:

```
void workerSync(int tid, int n)
{
  pthread_mutex_lock(&mutexCounter);
  counter++;
  pthread_mutex_unlock(&mutexCounter);
}
```

E. Reduction Flavors

In our real-world workload (see Section V), we also need to conduct reduction operations by summing over the partial per-thread sums of the first and second derivative of a function in conjunction with the barrier. An additional constraint that we impose is that, additions of partial sums must always be conducted in the same order (see Section I) for avoiding slight variations in the results of reductions on exactly the same partial numerical values due to floating-point rounding errors. That is, the accuracy or lack of accuracy induced by reduction operations per se, is not the problem we intend to address. For a discussion of issues related to accuracy of reduction operations see [7]. Instead, we assess different approaches for enforcing arbitrary, but deterministic reduction orders.

We consider two basic options. One option is that the worker threads simply store their partial sums in a shared array. After the barrier, the master thread can then simply compute the sum over those arrays to obtain the overall value. Conducting this operation sequentially at the master, may decrease parallel performance, if a large number of cores is available. To this end, we also vectorized the computation of the partial sums over the two `double` arrays for the first and second derivative using SSE3 intrinsics.

The second alternative is to conduct the reduction using the "classic" tree-like parallel reduction scheme when the lock-free recursive barrier is used. That is, if a reduction is required in conjunction with the barrier, we also recursively compute the reduction (which ensures a deterministic order of floating-point operations) at the same time as the barrier. We implemented this option for the padded and unpadded versions of the recursive lock-free barrier.

IV. Synthetic Workload

As synthetic workload, we implemented a simple benchmark program in C++. We used the `boost::thread` library, which provides a C++ wrapper for the Pthreads library. After initialization, the threads immediately enter the outer loop of the benchmark program.

The number of iterations N conducted in the outer-loop is a user-defined parameter. Inside the loop, the worker-threads perform a busy-wait for a signal by the master that triggers the parallel region (see Section III). Thereafter, the threads execute their workload (see below) and notify the master thread that they have completed their work via a barrier. The master thread also executes the workload computations. After completion of the parallel region, the master thread can optionally verify the correctness of the results computed by the worker threads.

The workload in this synthetic benchmark, consists of simple operations on floating-point arrays. Each thread holds three arrays $(v1, v2, v3)$ of fixed size M. The arrays $v1$ and $v2$ are initialized with random values when the threads are started. To avoid non-determinism, each thread always generates the same random values by using exactly the same random number seed.

Thereafter, each thread computes values for $v3$ as follows: $v3_i = v1_i \cdot v2_i$, where $i = 1, ..., M$. Then, the threads also calculate the respective integer sum across $v3$: $res = \sum_{i=1}^{M} \lfloor v3_i \rfloor$. After the completion of the above operations, res can be written to a `volatile` array by each thread, which can then optionally be read by the master-thread to verify the results computed by the individual worker threads. Note that, $v1$ and $v2$ are initialized with the same random values across all threads. Thus, after the barrier, the master-thread can simply compare its local value of res to the values computed by the worker-threads to verify consistency and correctness. If any of the worker-threads has not finished the calculations at this point, the respective entry for res in the `volatile` result array, will still contain its initial value of 0. When M is set to zero, the threads will not execute any computations before entering the barrier. Thus, the benchmark will effectively only measure the runtime for N thread synchronizations. When the N iterations of the outer-loop (with or without workload) have been completed, the master-thread will report the time required by the outer loop.

For the synthetic benchmark, we used statically linked executables compiled with gcc v4.5.0. All executables were compiled without optimization (flag `-O0`) on the same system for 64bit Arch Linux. Compiler optimizations were disabled, to minimize the influence of the compiler on the different barrier implementations. The values for N and M were set at compile-time. If not stated otherwise, all threads are pinned to individual, physical cores. By using anonymous `mmap(...)` calls for memory allocation, we

ensured that, the main memory pages containing the three arrays $v1, v2, v3$, were located as closely as possible to the physical cores (of the distributed shared memory systems) on which the respective threads were running.

This is particularly important for the NUMA-based multi-socket systems we used (i.e., the 8-core Nehalem, 16-core Barcelona, and 32-core SUN x4600 systems), where memory access speeds are not uniform over the address space. We also deploy `mmap(...)` to ensure that, the barrier arrays and counter variables used for synchronization, are located as closely as possible to the core that executes the master-thread.

V. Real Workload

As real workload, we used the Pthreads-based parallel version of RAxML [1], [8]. RAxML is a widely-used open-source code for reconstruction of phylogenetic (evolutionary) trees from molecular and morphological sequence data under the maximum likelihood (ML [9]) criterion and is freely available for download at http://wwwkramer.in.tum.de/exelixis/software.html.

The input for a phylogenetic analysis is a multiple sequence alignment (MSA) of the organisms under study. The output is an unrooted binary tree; the currently living organisms in the MSA are located at the tips of such a tree, while the inner nodes represent extinct common ancestors. The main bulk of the floating-point-intensive likelihood computations, consists of computing the probabilities for observing an A, C, G, or T (for DNA data) at the ancestral nodes for every position/column of the MSA. Since the overall tree likelihood is computed with respect to a virtual root (for details, see [9]) via a post-order traversal of the tree, this operation entails a large amount of synchronization points.

Here, we focus on the ML function in RAxML, that requires the largest amount of synchronization per computation and typically 30% of overall run time: the optimization of branch lengths with respect to the overall likelihood of the tree. In order to compute the ML value, for a given—fixed—tree topology, one needs to find the optimal branch length configuration for the given tree. In RAxML, as well as in other popular ML-based programs, branch length optimization algorithms traverse the tree (in order), and optimize one branch at a time via a Newton-Raphson method, that requires the calculation of the first and second derivative of the likelihood function. We call one such tree traversal that optimizes all branches, a branch length optimization cycle. Typically, several branch length optimization cycles will be carried out (the tree is traversed repeatedly) until either the overall change in the likelihood of the tree is smaller than some pre-defined ϵ, or the induced branch length change is smaller than some ϵ.

For testing barrier implementations, branch length optimization represents the most challenging operation because (i) synchronization is required between each of the $2n - 3$ (n is the number of organisms) branches of a branch length optimization cycle, (ii) the Newton-Raphson procedure requires synchronization between each Newton-Raphson iteration at a single branch, and (iii) the Newton-Raphson procedure requires reductions on two double values (to sum over the 1st and 2nd derivative of the likelihood function per MSA column across all columns) after each iteration at each branch. An abstract representation of the branch length optimization procedure in RAxML is provided in Figure 3.

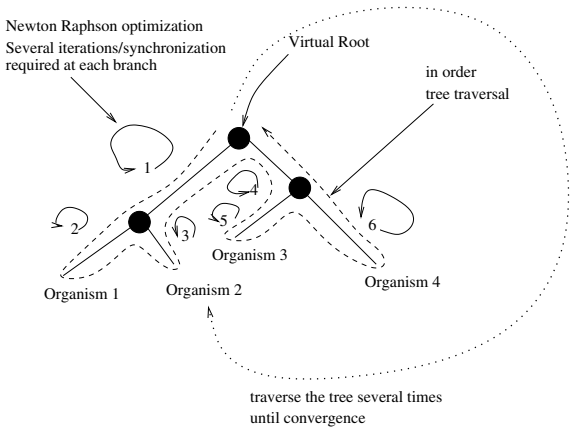

Figure 3. Outline of the branch length optimization procedure in RAxML for an example tree with 4 organisms.

We have implemented a dedicated RAxML version that predominantly executes branch length optimization operations on a fixed tree. The modified version reads in a given—fixed—tree topology and will then simply optimize branch lengths, that is, conduct as many branch length optimization cycles as are required until convergence. To obtain slightly more stable execution times, we reset the branches to the default values and conduct the full branch length optimization procedure five times. We only measure execution times around the five invocations of the branch length optimization routine. The alternative barrier implementations are integrated via appropriate compiler directives.

As test data, we used two MSAs for DNA data with 404 organisms and 7,429 distinct column patterns and 1,481 organisms and 1,241 distinct column patterns, respectively. The per-column likelihood scores (1st and 2nd derivatives) of the 7,429 patterns (404 organisms) and 1,241 (1,481 organisms) can be computed in parallel, before the synchronization and reduction operation. The dataset with 404 organisms scales reasonably well up to 32 cores; scalability is proportional to the number of distinct column patterns [10]. The dataset with 1,481 organisms scales less well, because the tree comprises approximately three times more branches and significantly less column patterns, that is, more synchronization per computation is required.

On the 404 organism MSA, the code region for which we measured execution times carries out a total of 193,865 barrier calls. 129,535 out of those 193,865 barriers are invoked by the Newton-Raphson procedure, that is, 129,535 invocations also require a reduction operation on two partial per-thread sums (1st and 2nd derivative of the likelihood function). On the 1,481 organism MSA, we execute 738,835 barriers of which 502,185 invocations include reduction operations.

VI. Results

A. Test Systems

We used the following test systems: 2-core Intel Core2 Duo P9400, 4-core Intel Core 2 Quad Q9550, 4-core Intel Core i5 750, 8-core (2-way Quad-Core) Nehalem system with 32GB of RAM (Intel E5540 CPU at 2.53 GHz, gcc v4.4.1), 16-core (4-way Quad-Core) AMD Barcelona system with 128GB RAM (AMD 8354 CPU at 2.2 GHz, gcc v4.4.3), a 32-core SUN x4600 (8-way Quad-Core) AMD-based system with 64GB RAM (AMD 8384 CPU at 2.53 GHz, gcc v4.3.2), a 24-core SUN x4440 (4-way Six-Core) AMD based system with 128GB RAM (AMD 8431 CPU at 2.4 GHz, gcc v4.4.3).

The gcc compiler versions indicated in parentheses were used to compile the RAxML workload on the larger systems.

B. Synthetic Workload

In Table I, we provide the execution times for the synthetic benchmark without workload (i.e., $N := 10,000,000, M := 0$). Thus, using this setup we only measure thread synchronization times. On all systems, the barrier based on Mutex-Inc-Add (see Section III-D), generates the highest synchronization overhead. Note that, for Mutex-Inc-Add, we only provide the run times for a single run (as opposed to the average over five runs), because of the excessive runtimes on the larger systems.

In this test, there is an approximately 50-fold difference in synchronization times between the smallest (2 cores) and the largest system (32 cores) for all alternative barrier implementations. For Mutex-Inc-Add, this difference is much higher, which suggests that it scales considerably worse on multi-socket systems, than other barrier flavors.

The implementations that deploy padding for preventing false sharing, perform better than the corresponding implementations without padding on the multi-socket systems. On multi-socket systems, the lock-free padded barrier yields the best overall performance.

In Table II, we provide the execution times for the synthetic benchmark *with* workload ($N := 100,000, M := 10,000$). The size M of the workload was chosen such that the data used in the computations, do not fit into the L1 cache of the cores. The cores have L1 data-caches of 32Kb, whereas the workload operates on vectors with an accumulated size (over $v1, v2, v3$) of 120Kb. On the test systems, the runtime differences between the alternative barrier implementations are substantially smaller than in the previous test, because only 100,000 instead of 10,000,000 barriers are invoked. Nonetheless, the recursive (see Section III-B) and mutex-based barriers exhibit considerably higher runtimes on some of the larger test systems. The highest runtime was measured for Mutex-Inc-Add on the largest test-system (SUN x4600 with 8 CPU-sockets). In general, either Lock-Free-Padded or Atomic-Inc-Add yield the best runtimes for a workload whose data resides in the L2-caches of all CPUs, but does not fit into the L1-caches.

1) Pinning Effects: Using less Threads than Cores: In certain special cases, the pinning (thread to core mapping) of individual threads can have a considerable impact on barrier execution times. We observed this in two cases for the synthetic benchmark without workload ($N := 10,000,000, M := 0$). This behavior was observed on the core 2 quad CPU when two threads are used, and on the Nehalem when virtual hyper-threading cores are used.

When the two threads on the core 2 quad are pinned to the two cores on the first processor, the runtime of the benchmark using the Lock-Free-Padded barrier amounts to approximately 1.8 seconds, which is comparable to the results obtained for 2 threads running on the core 2 duo system. When the threads are pinned to the first and third core (1st and 2nd processor respectively), the barrier time (Lock-Free-Padded) increases by more than a factor of two to approximately 4.3 seconds. On the core 2 quad CPU, two pairs of cores share a common L2-cache. Thus, synchronization between two cores on the same processor, is considerably faster than between cores/threads on different processors, which share the L2-cache. On the core i5, where each core has an individual L2-cache this effect of thread to core pinning can not be observed.

A similar effect can be observed on the Nehalem systems when hyper-threading is enabled. When 8 threads are pinned to the physical cores (located on the two sockets), the runtime of the barrier-only benchmark is approximately 13.6 seconds. When the 8 threads are pinned to the 4 physical and 4 virtual cores of the same CPU socket, the runtime decreases to approximately 9 seconds. In this setting, no synchronization across the sockets is necessary. Therefore, the benchmark is faster, despite not using the full number of physical cores available in the system.

C. Real Workload

For the RAxML workload, we used the full number of physical cores on the 4 larger systems, since this represents the typical application scenario. On the Nehalem, we ensured that all threads were pinned (for a discussion of performance issues pertaining to thread-to-core mappings, see [11]) to the physical cores and we did not analyze the impact of hyper-threading. On all other systems, we pinned thread 0 (the master thread) to core 0, thread 1 to core 1, etc.

Table I
EXECUTION TIMES (SECONDS) FOR THE SYNTHETIC BENCHMARK WITHOUT WORKLOAD ($N := 10,000,000$, $M := 0$).

System	core2d (2T)	core2q (4T)	core i5 (4T)	Nehalem (8T)	Barcelona (16T)	x4440 (24T)	x4600 (32T)
Lock-Free	**2.366**	7.315	4.887	15.589	50.084	66.97	111.991
Lock-Free-Padded	2.370	**6.577**	4.788	**13.559**	**47.819**	**63.488**	**104.804**
Recursive	2.390	8.337	5.200	17.258	70.774	112.151	180.155
Recursive-Padded	2.396	7.554	5.460	14.883	51.462	73.222	121.957
Atomic-Inc-Add	2.432	7.439	**4.640**	14.682	48.799	65.590	107.950
Mutex-Inc-Add	3.121	60.92	20.236	105.135	438.651	636.085	1413.045

Table II
EXECUTION TIMES (SECONDS) FOR THE SYNTHETIC BENCHMARK WITH WORKLOAD ($N := 100,000$, $M := 10,000$).

System	core2d (2T)	core2q (4T)	core i5 (4T)	Nehalem (8T)	Barcelona (16T)	x4440 (24T)	x4600 (32T)
Lock-Free	9.087	7.753	8.675	8.343	14.305	16.738	12.291
Lock-Free-Padded	9.093	7.717	8.676	**8.328**	14.324	**15.466**	12.343
Recursive	9.086	7.762	8.679	8.368	14.469	17.223	12.963
Recursive-Padded	9.105	**7.700**	8.681	8.350	14.344	15.978	12.470
Atomic-Inc-Add	**9.058**	7.747	**8.672**	8.365	**14.268**	15.996	**12.269**
Mutex-Inc-Add	9.025	7.997	8.794	9.053	18.097	17.204	23.507

In Table III we provide execution times for parallel branch length optimization in seconds using the 404 organism MSA for the SUN x4600 (32 cores), Barcelona (16 cores), Nehalem (8 physical cores), and SUN x4440 (24 cores) systems for all barrier implementations available. The respective fastest barrier is shown in bold font. Note that, execution times are averages over 5 independent runs of RAxML.

In Table IV we provide execution times for parallel branch length optimization in seconds using the 1,481 organism MSA for the SUN x4600 (32 cores), Barcelona (16 cores), and Nehalem (8 physical cores) systems for all barrier implementations available. The respective fastest barrier is shown in bold font. Execution times are averages over 5 independent runs of RAxML.

Overall, we observe, that using the intrinsics atomic increment operation, does not yield good performance, especially on the SUN x4600 32-core system. While performance appears to be better on more recent multi-core systems, the lock-free barrier implementations seem to guarantee the best performance across all systems using a real workload. Hence, lock free barriers should be used for designing multi-threaded programs that work well on most common multi-core architectures. However, given the results obtained for the synthetic benchmark, barrier performance appears to be application- and data-specific in the sense that barrier performance depends on the cache utilization of the respective application. Thus, it may be worthwhile to experimentally determine the optimal barrier for a specific application. Padding does not appear to have a clear impact on performance for lock-free barriers in RAxML. For recursive lock-free barriers however, padding appears to generally yield better performance (also observed for the synthetic benchmarks), mainly because reads are conducted by several threads at a time, as opposed to flat lock-free barriers where only the master thread reads the data written by the worker threads.

With respect to reduction operations, using SSE3 intrinsics for speeding up the flat and simple lock-based barrier reductions, is advantageous. The favorable performance impact of vectorizing master-based reductions of partial sums, may increase if systems with more cores and the new 256 bit AVX vector instructions become available. Recursive reductions using tree-based barriers may be worth further consideration, in particular if the reduction operation is some complex function $f()$, rather than a simple addition of floating point values. Thus, the choice for using a tree-based or flat lock-free barrier, will depend on the number of cycles required by respective reduction operations.

Finally, lock-based barriers yield highly inefficient code, especially on AMD-based architectures.

VII. CONCLUSIONS

We have conducted a thorough assessment of alternative barrier implementations on a large number of current multi-core systems with 2 up to 32 cores using a synthetic benchmark/workload, as well as a real workload. We find that, barrier performance is application-dependent and that application cache utilization has a major impact on which barrier performs best. We also show that, for scientific codes with a high number of relatively small parallel regions, barrier performance can have a significant impact on overall execution times.

We also assess combined barriers/reduction operations with RAxML. Overall, relatively naïve, but portable, lock-free barriers, seem to perform best across all systems and applications. Our benchmarks and barrier implementations are provided as open-source code to the community for further testing. Nonetheless, given the steady increase in core count, it would be desirable to have dedicated hardware for synchronizing threads in order to facilitate the development of scientific codes on multi- or many-core platforms.

Table III
EXECUTION TIMES (SECONDS) FOR PARALLEL BRANCH LENGTH OPTIMIZATION WITH RAXML ON A DATASET WITH 404 ORGANISMS USING ALTERNATIVE BARRIER IMPLEMENTATIONS ON THE X4600, BARCELONA, NEHALEM, AND X4440 SYSTEMS.

System	x4600 (32)	Barcelona (16)	Nehalem (8)	x4440 (24)
Lock-Free	9.260	16.421	22.916	12.559
Lock-Free-SSE3	**9.234**	**16.378**	**22.885**	**12.533**
Lock-Free-Padded	9.267	16.433	22.969	12.630
Recursive	10.444	16.679	22.919	13.058
Recursive-Padded	9.387	16.504	23.016	12.656
Recursive-Padded-Red	9.407	16.510	22.934	12.725
Atomic-Inc-Add	11.669	17.073	22.916	12.583
Mutex-Inc-Add	37.503	22.303	23.298	20.254

Table IV
EXECUTION TIMES (SECONDS) FOR PARALLEL BRANCH LENGTH OPTIMIZATION WITH RAXML ON A DATASET WITH 1,481 ORGANISMS USING ALTERNATIVE BARRIER IMPLEMENTATIONS ON THE X4600, BARCELONA, AND NEHALEM SYSTEMS.

System	x4600 (32)	Barcelona (16)	Nehalem (8)
Lock-Free	11.678	13.803	15.000
Lock-Free-SSE3	11.565	**13.705**	14.982
Lock-Free-Padded	11.590	13.734	14.920
Lock-Free-Padded-SSE3	**11.486**	13.722	14.896
Recursive	16.743	15.141	15.090
Recursive-Padded	11.935	13.946	14.996
Recursive-Padded-Red	12.062	13.936	**14.890**
Atomic-Inc-Add	21.039	16.624	15.171
Mutex-Inc-Add	126.252	39.994	18.908

Future work, will entail a more detailed study of the impact of application cache utilization and memory access patterns on barrier performance.

ACKNOWLEDGMENTS

The authors would like to thank Nicloas Salamin for providing the real-world biological dataset used in this study and Bernard Moret for access to the 16-core Barcelona system.

REFERENCES

[1] A. Stamatakis and M. Ott, "Efficient computation of the phylogenetic likelihood function on multi-gene alignments and multi-core architectures." *Phil. Trans. R. Soc. series B, Biol. Sci.*, vol. 363, pp. 3977–3984, 2008.

[2] R. Berrendorf and G. Nieken, "Performance characteristics for OpenMP constructs on different parallel computer architectures," *Concurrency: Practice and Experience*, vol. 12, pp. 1261–1273 , 2000.

[3] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 235–246.

[4] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 35–45.

[5] J. Chen and W. W. Iii, "Multi-threading performance on commodity multi-core processors," in *In Proceedings of 9th International Conference on High Performance Computing in Asia Pacific Region (HPCAsia*, 2007.

[6] Y. Voronenko, F. Franchetti, F. de Mesmay, and M. Püschel, "Generating high-performance general size linear transform libraries using Spiral," in *High Performance Embedded Computing (HPEC)*, 2008.

[7] Y. He and C. Ding, "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications," *The Journal of Supercomputing*, vol. 18, no. 3, pp. 259–277, 2001.

[8] A. Stamatakis and M. Ott, "Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study." in *PRIB*, ser. Lecture Notes in Computer Science, M. Chetty, A. Ngom, and S. Ahmad, Eds., vol. 5265. Springer, 2008, pp. 424–435.

[9] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach," *J. Mol. Evol.*, vol. 17, pp. 368–376, 1981.

[10] M. Ott, J. Zola, S. Aluru, and A. Stamatakis, "Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L," in *Proc. of IEEE/ACM Supercomputing Conference 2007 (SC2007)*, 2007.

[11] M. Ott, T. Klug, J. Weidendorfer, and C. Trinitis, "autopin-Automated Optimization of Thread-to-Core Pinning on Multicore Systems," in *Proceedings of 1st Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2008.