Effective Barrier Synchronization on Intel Xeon Phi Coprocessor

DOI:
10.1007/978-3-662-48096-0_45
10.1007/978-3-662-48096-0

Effective Barrier Synchronization on Intel Xeon Phi Coprocessor

Andrey Rodchenko, Andy Nisbet, Antoniu Pop, and Mikel Luján

School of Computer Science, The University of Manchester, UK
{andrey.rodchenko, andy.nisbet, antoniu.pop, mikel.lujan}@manchester.ac.uk

Abstract. Barriers are a fundamental synchronization primitive, underpinning the parallel execution models of many modern shared-memory parallel programming languages such as OpenMP, OpenCL or Cilk, and are one of the main challenges to scaling. State-of-the-art barrier synchronization algorithms differ in tradeoffs between critical path length, communication traffic patterns and memory footprint. In this paper, we evaluate the efficiency of five such algorithms on the Intel Xeon Phi coprocessor. In addition, we present a novel hybrid barrier implementation that exploits the Xeon Phi's topology, memory hierarchy and streaming stores to achieve a 3× lower overhead than the Intel OpenMP barrier implementation (ICC 14.0.0), thus outperforming, to the best of our knowledge, all other implementations, and which we evaluate on the CG and MG kernels from the NAS Parallel Benchmarks, the direct N-body simulation kernel and the EPCC barrier OpenMP microbenchmark.

Keywords: barrier synchronization; scalability; algorithms; many-core architectures; Intel Xeon Phi;

1 Introduction

Multi- and many-core systems have become the norm, and their efficient exploitation requires efficient and scalable synchronization mechanisms. Barriers are one of the fundamental synchronization primitives, underpinning the parallel execution models of many modern shared-memory parallel programming languages such as OpenMP, OpenCL or Cilk. The optimization of software barrier synchronization has been widely studied [3, 17, 9, 12, 8, 10], yet no algorithm has proven optimal across the wide variety of parallel architectures. Indeed, each algorithm comes with its own set of tradeoffs with respect to communication complexity (volume) and patterns, length of the critical path, and memory footprint. For any given architecture, the optimal algorithm is largely dependent on factors such as the system's topology, the structure of the memory hierarchy, and the characteristics of the system interconnect.

The focus of this paper is to analyze and optimize the efficiency of barrier synchronization on the Intel Xeon Phi coprocessor. Based on the Intel MIC (Many Integrated Core) Architecture, that provides a commodity off-the-shelf many-core systems, the Xeon Phi has up to 61 cores, each 4-way SMT, for a maximum

of 244 logical threads. At this scale, the efficiency of barrier synchronization is crucial for performance in synchronization intensive workloads.

Our **first contribution** is a thorough evaluation of the behavior of current state-of-the-art barrier algorithms, and an analysis of their tradeoffs for the Xeon Phi's memory hierarchy. We show that while the best algorithm depends on run-time conditions, a single statically chosen algorithm is only marginally outperformed in a small number of cases. Our **second contribution** is the proposal of a more efficient hybrid algorithm, mixing different (existing) barrier algorithms at different levels of granularity of synchronization, and optimized with *streaming store* instructions to write full cache lines, that eliminate a costly read-for-ownership cache coherency operation. We show that our hybrid approach outperforms all previous algorithms on the Intel Xeon Phi coprocessor.

Section 2 presents key features of the Xeon Phi and the resulting methodological constraints for experiments. Section 3 reviews the state-of-the-art barrier synchronization algorithms and their implementations. Our optimizations and our new hybrid synchronization scheme are presented in Section 4 [1]. Section 5 presents our experimental findings. Finally, Section 6 discusses previous work on Xeon Phi barrier optimization, and Section 7 summarizes our work.

2 Intel Xeon Phi Coprocessor

We experimented on a 60-Core Xeon Phi 5110P clocked up to 1.053GHz. Cores are in-order, 4-way SMT, using a bidirectional ring interconnect. Each core has 32 KB of L1 and 512 KB of L2 cache. The state of the distributed L2 cache is controlled by a distributed tag directory implementing the GOLS protocol. The coherence of L1 and L2 caches is maintained by a modified MESI protocol. However the GOLS protocol makes it possible to emulate the *Owner* state, enabling a MOESI-like functionality. 8GB of GDDR5 RAM is accessed through 8 dual channel memory controllers connected through a ring interconnect interface.

The 512-bit SIMD instructions were used to optimize barrier synchronization in the *SIMD barrier* [5]. SIMD stores, also known as *streaming stores*, use a vector size matching that of a cache line. As a result, such store instructions do not need to issue a *read-for-ownership* request in the cache coherence protocol. Store instructions with the *no-read* hint can be either globally ordered (`vmovnrap[d/s]`), providing a *total store order* type consistency, or non globally ordered (`vmovnrngoap[d/s]`), leading to a weaker memory consistency.

The execution of HW threads can be paused, using the `delay r32/r64` instruction, which forces the processor to halt the fetch and issue of further instructions for a parametric number of cycles. The `pause` instruction is not available.

3 Barrier Synchronization Algorithms

Many algorithms have been proposed for barrier synchronization. One of the simplest, and least scalable, is the *Sense-reversing centralized barrier*. It uses two

[1] All source code is available at https://github.com/arodchen/cbarriers

global variables, a counter and a flag, and one thread-local flag called *sense*. The synchronization counter is initialized with the number of threads in the barrier, the global flag is set to `false` and all local sense flags are set to `true`. Upon reaching a barrier, each thread registers its arrival by atomically decrementing the counter, then waits while the value of the global flag is different from its local flag. The last thread to complete the atomic operation, when the counter reaches zero, re-initializes the counter, flips the global flag and its local flag. The other threads eventually perceive that the global sense has changed and pass the barrier, flipping their own flag. This technique, called *sense-reversing*, allows the reuse of the same barrier variables for the next synchronization round.

To reduce the contention on shared variables, the *combining-tree barrier* [17] organizes the participating threads in a tree, using an algorithm similar to a centralized barrier at each node of the tree: the last thread to decrement the synchronization counter of a node recursively proceeds to decrement the counter of its parent, while the other threads wait on node-level release flags for notification that the barrier has passed. It is also possible to use a global release flag, trading a shorter critical path for additional contention on the global flag.

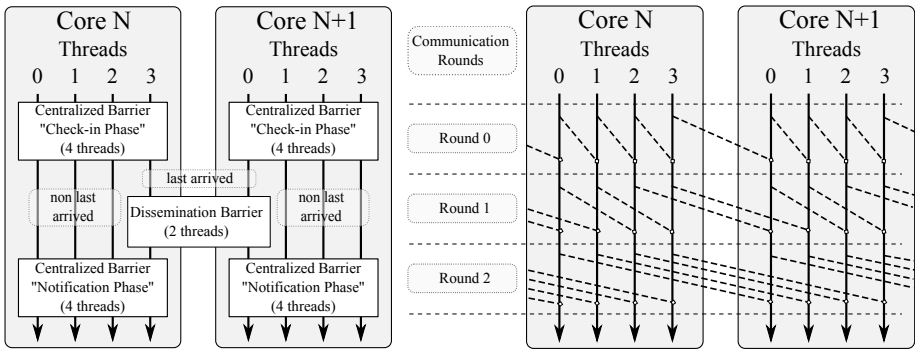
Realizing that atomic operations were only necessary to reach a consensus on the arrival order of threads, Hensgen *et al.* proposed the *static tournament barrier* [9]. Instead of discriminating on arrival order, it relies on a statically determined thread, called *winner*, that will automatically progress to the next round once all other threads, the *losers*, have arrived at a given node of the tree. In this way, it is only necessary to determine whether all threads are accounted for, irrespectively of their arrival order, which does not require atomic operations. The initial version of the algorithm by Hensgen et al. [9] was later improved by Mellor-Crummey et al. [12] with a tree-based "Notification Phase" and sense-reversing to avoid re-initializing the barrier state. The *static f-way tournament* by Grunwald et al. [8] generalizes the static tournament approach, with an arbitrary number of participants per round. To avoid cases where a static winner arrives early and busy-waits on a location yet to be set by one or multiple losers, the *dynamic f-way tournament barrier* [8] lets the winner identify itself by checking the value of adjacent memory locations marked by other threads upon arrival.

A shortcoming shared by all previous algorithms is that they all require two phases: registration of threads arrival - "Check-in Phase" and notification of threads arrival - "Notification Phase". However, these phases can be merged, at the cost of additional communication, by providing each thread with sufficient information to locally decide when the barrier can be passed. The first instance of this class of barriers was the *butterfly barrier* [3]. When the number of threads participating in the barrier is not a power of 2, the *dissemination barrier* [9] can be a more efficient solution. In the dissemination barrier pattern, a thread i in round r notifies another thread $j = (i + 2^r) mod N$ by writing its local flag, which is sense-reversed, to a dedicated memory location, where N is the number of threads, $i, j \in [0, N-1]$ and $r \in [0, \lfloor \log_2 N \rfloor]$. Each thread can proceed to the next round as soon as its notification variable is set to the appropriate value for the current round. In this way, synchronization is achieved in $\lceil \log_2 N \rceil$ rounds. The *f-way*

dissemination barrier [10] is a generalized version, where each thread can notify f other threads in one round, requiring only $\lceil \log_{f+1} N \rceil$ rounds to complete. If $f = N-1$, then it will be a broadcast barrier (all-to-all communication) requiring $N * (N-1)$ notifications.

4 Hybrid Barrier

We came to the design of the hybrid barrier from the observation that, on systems with a hierarchical topology, different algorithms can be optimal for different levels in the hierarchy. As always, the objective is to minimize inter-core communication while exploiting the low cost of intra-core communication. During our preliminary evaluation, we observed that the most efficient algorithms were the dissemination barrier and the combining tree with arity 4, which already is intrinsically hierarchical. Figure 1b shows the communication patterns of a dissemination barrier. Evidently, each round contains at least one inter-core communication edge, which will be slower and will consequently determine the critical path for each round. We therefore propose to rely on the centralized sense-reversing barrier (equivalent to the combining tree with arity 4) for the intra-core phase, then revert to dissemination once a single thread remains per core. A similar approach was discussed by Cownie [6]. Figure 1a and Listing 1.1 show the scheme and pseudocode of the hybrid algorithm.



(a) scheme of hybrid dissemination barrier

(b) communication pattern of dissemination barrier

Fig. 1: Hybrid dissemination barrier rationale.

Busy-Waiting Amortization Busy-waiting on the same memory location can have a negative impact on performance because of the increased memory traffic, leaving less bandwidth to other threads. This effect was previously noted [2] and we empirically determined (see Section 5) the appropriate delay to amortize the

impact of busy-waiting while balancing the additional latency this introduces for a thread to perceive a store operation. The delay can be introduced with the `_mm_delay32` intrinsic, as shown in Listing 1.2, the parameter specifies the number of idle cycles.

```
void                    /* Hybrid Barrier */
hb_wait( hb_Bar_t * hb, /* Shared Data */
         tp_Data_t * tp) /* Thread Private Data */
{
  sr_Bar_t * srB = hb->srB [tp->srBId];
  int srBC = fetch_and_add( & (srB->count), -1);
  int tpsrBS = tp->srBSense;

  if ( srBC == 1 )
  { /* last thread inside core to arrive */
    /* call to dissemination barrier */
    dsmn_wait( hb->dsmnB, tp);
    mic_store( & (srB->count), srB->threadsNum);
    mic_store( & (srB->sense), tpsrBS);
  } else
  { /* non-last thread to arrive */
    while ( tpsrBS != srB->sense )
    {
      mic_pause( );
    }
  }
  mic_store( & (tp->srBSense), ! tpsrBS);
}
```

Listing 1.1: Hybrid barrier wait method.

```
inline static void
mic_pause( )
{
    _mm_delay_32( ARCH_MIC_DELAY);
}
```

Listing 1.2: Busy-waiting delay.

```
static inline void
mic_store( volatile void *addr, int data)
{
#if __use_streaming_stores
  _mm512i siVec =
    _mm512_set1_epi32( data);
  _mm512_storenr_ps( addr,
    _mm512_castsi512_ps( siVec));
#else
  *addr = data;
#endif
}
```

Listing 1.3: Utilization of streaming stores.

Streaming Stores Streaming stores can reduce barrier overhead [11] when storing notification values to flags in the "Notification Phase" or reinitializing counters in the combining tree or sense-reversing centralized barrier. Listing 1.3 details the implementation of the `mic_store` function. Note that, contrary to Caballero et al. [5], we use the *globally ordered* version of streaming stores. The rationale for this is detailed in Section 5.

5 Experimental Results

5.1 Benchmarks

EPCC OpenMP Microbenchmarks. We implemented a part of the EPCC OpenMP microbenchmark [4] evaluating the overhead of the standalone barrier primitive, which will be referenced as EPCC.

NAS Parallel Benchmarks. We chose the CG and MG kernels from the C versions [15] of the *NAS Parallel Benchmarks* (NASPB) [1] in order to evaluate the efficiency of barrier synchronization [5, 14, 16]. As the original inputs for these benchmarks lead to a low frequency of barrier synchronization, even using the smallest class S, which makes it difficult to observe barrier overhead, we introduced our own input class Y for these 2 NASPB kernels. The frequencies of barrier synchronization in both classes S and Y are presented in Table 1. Class Y consists of inputs { na=240; nonzer=2; niter=300; shiftY="5.0" } for CG and { problem_size=16; nit=800 } for MG. We also added a `collapse(2)` clause to the relevant OpenMP parallel loops in the MG kernel, as suggested in [5],

to increase the amount of parallelism which would otherwise not lead to a reasonable load balance. The `collapse` clause is used to specify that a loop nest is not only parallel on the first loop construct annotated, but also at a deeper level (parameter of the clause), which allows to collapse multiple loops into a single loop that is subsequently parallelized.

NAS Parallel Benchmark Kernel	CG		MG	
Input Class	S	Y	S	Y
Frequency, 10^3 barriers per second	6.4	21.6	8.7	12.4

Table 1: Barrier frequency in NASPB for inputs Y and S.

Direct N-body Simulation. We implemented a direct N-body simulation kernel to evaluate the efficiency of barrier synchronization in the task where synchronization cannot be relaxed. It was shown that using more relaxed synchronization constructs, like phasers, is more efficient than using barriers for CG and MG [16]. In the direct N-body simulation kernel we have implemented, each thread calculates the coordinates and velocities of a single particle, so that the frequency of barriers will be the highest. Coordinates and velocities of a single particle are stored within a private memory location which can fit into a single cache line. This kernel will be referenced as NBODY in the rest of the article.

To test our barrier implementations without interfering with the rest of the OpenMP implementation, we replaced the calls to the `__kmpc_barrier` function, which is the internal barrier function in the Intel OpenMP library, with a trampoline, `barrier_trampoline_`, that calls the function implementing the desired barrier algorithm.

5.2 Naming Convention and Methodology

For the remainder of this paper, the *geomean* overhead of a barrier (or execution time per barrier) is measured in our experiments on EPCC as the geometric mean of its overhead across the different thread counts; so $O_{geomean} = \sqrt[N]{\prod_{i=1}^{N} O_{n_i}}$, where N is the number of different thread counts, n_i is the number of threads in element i in the vector of different thread counts, and O_n is a barrier overhead for n participating threads. For CG, MG, and NBODY, we use execution time of a kernel instead of the overhead of a single barrier. On charts representing the geomean overhead (execution time), horizontal lines show the best (green, low horizontal line) and the worst (red, high horizontal line) geomean overhead respectively, calculated as geometric mean of the best and the worst barrier overheads for each number of threads amongst all algorithms.

The **best geomean overhead** represents the *practical lower bound of synchronization overhead*, which could be achieved given an oracle that predicts the best

possible algorithm in every configuration. The performance of this **ideal meta-algorithm** shows the loss of performance resulting from using a single algorithm compared to the ideal performance that could (theoretically) be achieved by selecting algorithms dynamically with an oracle.

Individual barrier algorithms on the charts can be identified by their signatures. A signature uniquely identifies an algorithm by two 3-5 letter abbreviations and a number. The first abbreviation corresponds to the algorithm and its variations: **"sr"** - centralized sense-reversing barrier; **"dsmn"** - dissemination barrier; **"dsmnH"** - hybrid dissemination barrier; **"ct"** - combining tree barrier; **"stn"** - static tournament barrier; **"dtn"** - dynamic tournament barrier; **"ls"** - tree-based notification with local flags; **"gs"** - broadcast notification using a single global flag; **"omp"** - Intel OpenMP barrier.

The second part of the signature defines whether the waiting loops contained the *64-cycle delay* (which will be discussed below) - **"pause"**, or contained no delay - **"spin"**. This part is meaningless for Intel OpenMP barrier.

The last part is a **number** corresponding to the arity of the tree for tree algorithms, the number of ways for a *n-way* dissemination barrier and meaningless for the centralized sense-reversing barrier and Intel OpenMP barrier.

All of our experiments rely on a "balanced" strategy for thread mapping, mapping threads to cores with the least load first. This type of thread affinity is enabled in Intel OpenMP by setting the environment variable `KMP_AFFINITY` to `balanced`; the `KMP_LIBRARY` variable was set to `"turnaround"`; the `KMP_BLOCKTIME` variable was set to `"infinite"`; and the number of threads was controlled by setting the `OMP_NUM_THREADS` variable.

Each data point was obtained from 10 measurements and represented by a box plot. Unless otherwise indicated in a Figure, the number of threads varies from 8 to 232 in increments of 8. The arities tested were: 2, 3, 4, 8, 16 and 32 for the combining tree barrier; 2, 3, 4 and 5 for the static tournament barrier; and 2, 3 and 4 for the dynamic tournament barrier.

5.3 Experimental Data and Discussion

Figure 2 shows the results for *geomean overhead* of barrier synchronization algorithms on EPCC, and Figure 3 shows the results of the *ideal meta-algorithm* on EPCC.

Global Sense vs Local Sense The overhead of tree barriers with global notification flag **"gs"** is much higher than that of barriers with a combining tree "Notification Phase" **"ls"** as can be seen on Figure 2, where the fastest **"gs"** variant is close to 2× slower than the slowest **"ls"** variant.

It can be observed that a combining tree with global flag in "Notification Phase" has higher geomean overhead than a centralized sense-reversing barrier, and the higher the tree arity, the less the overhead. This indicates that another hybrid algorithm should be investigated: a single global counter for the "Check-in Phase", like in the centralized sense-reversing barrier, and a tree-based "Notification Phase".

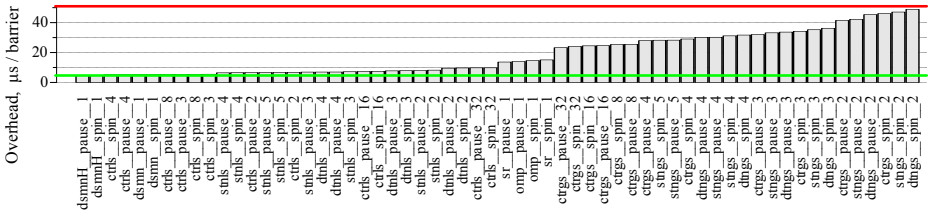


Fig. 2: Geomean overhead of barrier synchronization algorithms on EPCC.

Delayed Busy-Waiting As it can be seen from Figure 2, where the geomean overhead for different algorithms follows increasing order, the same algorithm with delayed busy-waiting **"pause"** outperform the undelayed spinning variants **"spin"** in the majority of cases. Due to this fact, we did not consider undelayed busy-waiting in our further experiments.

Sizing the Delay for Spinning To determine a suitable value (number of sleep cycles) to pass as parameter to the `delay` instruction conveniently provided on Xeon Phi, we evaluated delay values in the range from 0 to 128 cycles with step 8 on EPCC. Above 128 cycles, the performance starts to degrade as the delay introduces too much latency for waiting threads. The optimal performance was obtained with a 64 cycle delay that was used in all subsequent experiments. The optimality of this decision should be investigated further as it is likely to depend on runtime conditions, such as the level of contention on the interconnect.

Hybrid Barrier Figures 2 and 3 show that the hybrid dissemination barrier is the closest to the ideal meta-algorithm on synthetic benchmarks. Indeed, as confirmed by the results presented in Figure 4, the few instances where the hybrid barrier is not the most efficient on Figure 3 only correspond to minor timing variability up to 60 threads, up to which point the algorithms have similar overhead. Above 72 threads, the hybrid barrier overhead is considerably lower than that of the dissemination barrier.

Streaming Stores A non globally ordered streaming store is unordered in respect to other stores, meaning that other store instructions issued subsequently by the same thread can overtake it and become visible to other threads earlier. This relaxation of the memory ordering constraints makes it tempting to rely on this instruction for implementing barriers, as suggested by Cownie [6] and implemented by Caballero et al. [5]. However, we observed 5% less overhead on average for top performing barrier implementations on EPCC when using globally ordered streaming stores over both ordinary stores and non globally ordered streaming stores, having on average the same effect on barriers overhead.

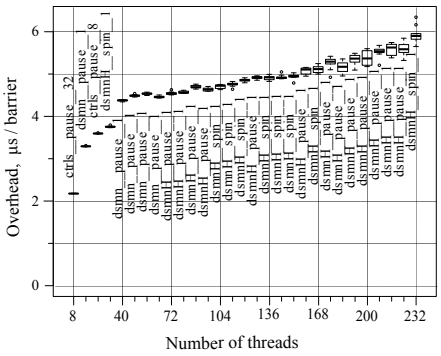


Fig. 3: Overhead of the ideal barrier synchronization meta-algorithm on EPCC.

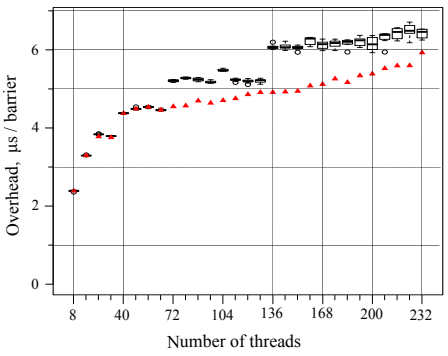
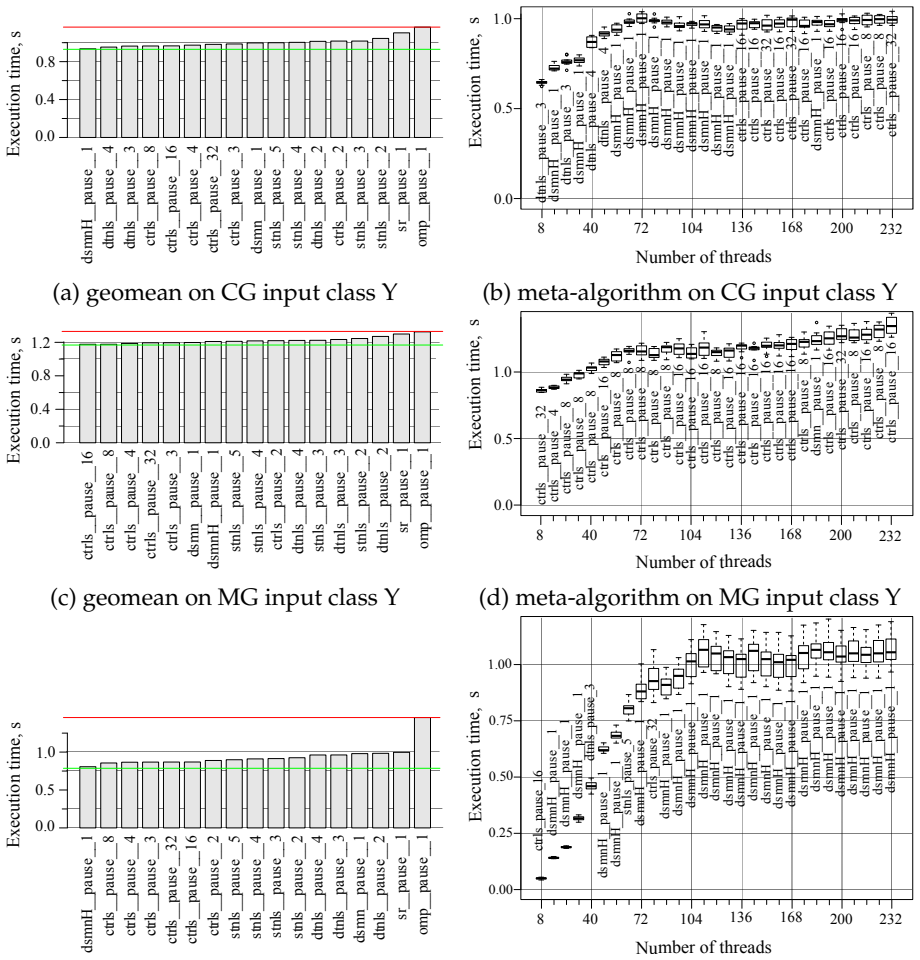


Fig. 4: Dissemination barrier (black boxplots) compared to hybrid dissemination barrier (red triangles).

Real-world Kernels The most efficient barrier algorithms selected above were evaluated on the CG and MG kernels of the NAS Parallel Benchmarks and direct N-body simulation kernel. Our results are presented in Figure 5. Hybrid dissemination barrier is superior over the other algorithms on CG and NBODY, while combining tree barrier with arities grater than 2 is slightly better than dissemination barrier on MG.

Effects of the Ring Interconnect As discussed by Dolbeau [7], the address of the shared memory location used for communicating among threads may have a significant effect on latency, and therefore on barrier overhead. We observed the same behavior, induced by the ring interconnect and the distributed tag directories, on a centralized sense-reversing barrier. Figures 6a, 6b, and 6c show three sets of performance results obtained on EPCC for 1 to 60 threads. Within each experiment, the data for a given number of threads was obtained in a single execution, containing multiple iterations where the same memory locations are re-used to store synchronization variables. The only difference between the three experiments is the allocation of the memory region used for synchronization, which contains different addresses. It is apparent that the variability of performance results is negligible within a given set, but it is significant in-between sets. The graph in Figure 6d shows the same benchmark, but for each thread count the benchmark was re-launched for every iteration, thus allocating different memory regions.

This variability is explained by the ring topology and the distributed tag directories. Indeed, in this configuration, each cache line is attributed a tag directory which is queried whenever a core misses in both L1 and L2 for that specific line, requiring a round-trip from the core to the adequate tag directory. This means that the delay of communications between threads during a barrier is dependent on the distances between threads and the tag directories that are responsible for the cache lines used in the synchronization. As threads are

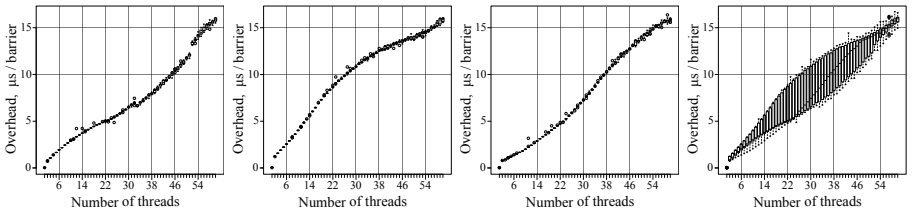


(a) geomean on CG input class Y (b) meta-algorithm on CG input class Y

(c) geomean on MG input class Y (d) meta-algorithm on MG input class Y

(e) geomean on NBODY (f) meta-algorithm on NBODY

Fig. 5: Comparison of barrier synchronization algorithms on CG and MG kernels of NAS Parallel Benchmarks and direct N-body simulation kernel.



(a) experiment 1 (b) experiment 2 (c) experiment 3 (d) experiments 1-3

Fig. 6: Impact of non-uniform cache line access latency.

mapped to cores in a circular order along the ring interconnect, the barrier overhead will follow an S-shaped path as in Figure 6c. If the selection of a tag directory is equiprobable, then (on average) the overhead of the barrier will increase linearly with the number of threads, following the average straight line apparent in Figure 6d. Unfortunately the selection of tag directories is not under explicit user control, which introduces extra variability in the barrier overhead.

6 Related Work

A barrier using SIMD instructions was proposed by Caballero et al. [5], achieving a 2.84× lower barrier overhead on EPCC than Intel's OpenMP barrier. However, they implemented their algorithm with non globally ordered streaming stores, which we showed to lead to a missed opportunity for 5% overhead reduction.

Dolbeau [7] shows that address selection is an important factor influencing barrier overhead due to non-uniform access time to distributed tag directories. The author shows that a combining tree of specific topology and topology-aware memory allocation would allow to lower the overhead of barrier synchronization. However, there is no explicit way to control topological aspects of memory allocation on Intel Xeon Phi systems. The author compares his barrier implementation against the Intel OpenMP barrier with an initial speedup of 2.41×, further showing that address selection leads to an improvement to 2.85×. The technique employed to control memory allocation for this result is based on a trial-and-error approach for reverse-engineering the hashing function used by the tag directories.

Finally, Ramos and Hoefler [13] propose a model for dissemination barrier synchronization and also compare with the Intel OpenMP barrier. However, the authors only show equivalent performance with the Intel implementation.

7 Conclusions

We optimized the performance of five state-of-the-art barrier synchronization algorithms on the Intel Xeon Phi coprocessor and provided a novel hybrid variant based on different algorithms to synchronize at intra-core and inter-core levels. Comparing our hybrid algorithm with previous implementations, we observed lower overheads in our experiments on EPCC barrier microbenchmark and an improved performance on direct N-body simulation kernel and on two NAS Parallel Benchmarks, CG and MG. In other words, we have presented the fastest known barrier implementation for Intel Xeon Phi.

We further provided an analysis of key specificities of the Xeon Phi system, in particular characterizing: (1) the impact of the ring interconnect and distributed tag directories leading to non-uniform cache line access latencies; (2) the performance degradation that can result from spin-based synchronization with insufficient delay; and (3) the positive impact of using globally ordered streaming stores for fine grained synchronization.

Acknowledgments. This work is supported by EPSRC grants DOME EP/J016330/1, PAMELA EP/K008730/1 and EP/M004880/1. A. Rodchenko is funded by a Microsoft Research PhD Scholarship, A. Pop is funded by a Royal Academy of Engineering Research Fellowship and M. Luján is funded by a Royal Society University Research Fellowship. We also thank the anonymous reviewers for their constructive feedback.

References

1. Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html
2. Agarwal, A., Cherian, M.: Adaptive backoff synchronization techniques. In: Proc. of the International Symposium on Computer Architecture. pp. 396–406. ACM (1989)
3. Brooks, III, E.D.: The butterfly barrier. Int. J. of Parallel Programming 15(4), 295–307 (1986), http://dx.doi.org/10.1007/BF01407877
4. Bull, J.M.: Measuring synchronisation and scheduling overheads in openmp. In: Proceedings of First European Workshop on OpenMP. vol. 8, p. 49 (1999)
5. Caballero, D., Duran, A., Martorell, X.: An openmp* barrier using simd instructions for intel xeon phi coprocessor. In: OpenMP in the Era of Low Power Devices and Accelerators, Lecture Notes in Computer Science, vol. 8122, pp. 99–113 (2013)
6. Cownie, J.: Fastest possible barrier (intel developer zone forum). http://software.intel.com/en-us/forums/topic/392587 (2013), [Online; accessed 2-Feb-2015]
7. Dolbeau, R.: Address selection for efficient barriers on the intel xeon phi. http://www.dolbeau.name/dolbeau/publications/barrierphi.pdf (2014), [Online; accessed 2-Feb-2015]
8. Grunwald, D., Vajracharya, S.: Efficient barriers for distributed shared memory computers. In: Proc. of Intl. Parallel Processing Symposium. pp. 604–608 (April 1994)
9. Hensgen, D., Finkel, R., Manber, U.: Two algorithms for barrier synchronization. IJPP 17(1), 1–17 (1988), http://dx.doi.org/10.1007/BF01379320
10. Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: Fast barrier synchronization for infiniband. In: 20th Int. Parallel and Distributed Processing Symposium (April 2006)
11. Krishnaiyer, R., Kultursay, E., Chawla, P., Preis, S., Zvezdin, A., Saito, H.: Compiler-based data prefetching and streaming non-temporal store generation for the intel(r) xeon phi(tm) coprocessor. In: 27th IEEE IPDPSW. pp. 1575–1586 (May 2013)
12. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM TOCS 9(1), 21–65 (1991)
13. Ramos, S., Hoefler, T.: Modeling communication in cache-coherent smp systems: A case-study with xeon phi. In: HPDC '13. pp. 97–108. ACM (2013), http://doi.acm.org/10.1145/2462902.2462916
14. Sartori, J., Kumar, R.: Low-overhead, high-speed multi-core barrier synchronization. In: Proc. of the 5th HiPEAC. pp. 18–34 (2010)
15. Seo, S., Jo, G., Lee, J.: Performance characterization of the nas parallel benchmarks in opencl. In: 2011 IEEE International Symposium on Workload Characterization (IISWC). pp. 137–148 (November 2011)
16. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In: Proceedings of the 22nd Annual International Conference on Supercomputing. pp. 277–288. ICS '08, ACM (2008), http://doi.acm.org/10.1145/1375527.1375568
17. Yew, P.C., Tzeng, N.F., Lawrie, D.H.: Distributing hot-spot addressing in large-scale multiprocessors. IEEE Transactions on Computers C-36(4), 388–395 (April 1987)