Address Selection for Efficient Barriers on the Intel®Xeon Phi™

Romain Dolbeau
CAPS entreprise
Rennes, France
Email: romain.dolbeau@caps-entreprise.com

Abstract—Synchronization primitives are a long-standing issue in parallel programming. Barrier in particular are ubiquitous as common paradigm such as OpenMP makes extensive use of them by ending all parallel sections on a barrier by default. The rising number of simultaneous threads in commodity hardware only exacerbate the problem as for a given amount of computation each thread will take less time to finish before having to wait a longer time for threads to synchronize. This paper focuses on the current Intel®Xeon Phi™which can distribute work to up to 244 threads on 61 cores, and the new challenges created by its specificities and in particular the ring bus connecting the cores. We will show that inter-core communication speed is highly dependent on the physical address of the variable being communicated, and that this fact has implications when building an efficient barrier. Carefully selecting the physical locations of variables involved in inter-thread communications lead to a 15% improvement in latency for the best barrier on the Xeon Phi™.

Revision 1.40, 2014/04/08 07:41:26

Index Terms—Barrier, Intel®Xeon Phi™, Synchronization Primitives, Combining Tree

Index Terms—arrier, Combining Tree, Intel®Xeon Phi™, Synchronization Primitivesarrier, Combining Tree, Intel®Xeon Phi™, Synchronization PrimitivesB

I. Introduction

Shared-memory parallelism has become an ubiquitous problem in contemporary computer science. Nearly all commodity hardware is now built using this paradigm, from cell phones to desktop computers to HPC nodes in the largest-scale machines. The ability to efficiently exploit such systems relies on the availability of fast synchronization primitives, among which mutual exclusion and synchronization barriers are keys. The case for fast barriers has been made extensively in the literature, such as by Sampson et al. [1] or Sartori et al. [2].

Many implementations have been proposed over the years, such as by Brooks [3], by Yew et al. [4] and so on. A landmark study of both mutual exclusion locks and barriers including a new algorithm for each aspect was done by Mellor-Crummey & Scott in 1991 [5]. But shared-memory parallel systems have greatly evolved since that publication. Two machines were described; the BBN Butterfly was based on the Motorola®68000 and did not have any cache, whereas the Sequent Symmetry based on the Intel®80386 only had a single-level 64 KiB data cache per CPU. Modern multi-sockets systems commonly have 3 levels of cache, of which only the third level is shared between cores - and only inside each socket. The memory wall by Wulf et al. [6] has been reached since this landmark study, and memory access cost is now more important than ever. It is conceivable that trade-offs and algorithms that were optimal in the era of custom-built parallel machine should be re-evalued in the era of off-the-shelf multi-core systems.

The Intel®MIC architecture [7] is an excellent case study for this. The current implementation in the Xeon Phi™(code-name Knights Corner) has up to 61 cores enabled, each capable of running 4 threads simultaneously. It also boasts a 64-bytes wide SIMD unit for vector-like capabilities (it includes both scatter and gather instructions). Working with 244 threads is a difficult proposition, and for the first time synchronization to such as scale is available to almost any researcher. The announced convergence in SIMD instruction set between the conventional Intel®CPU and the Xeon Phi™family with AVX-512 [8] makes it likely that such large-scale parallelism will eventually reach the mass market.

The large-scale parallelism combined with the requirements of a comparatively inexpensive chip (compared to the custom-built machines of past decades) creates some acute limitations. Gone are the sophisticated but expensive interconnection networks taught in the parallel architecture book by Culler at al [9]. The Xeon Phi™(like its more core-constrained cousins code-named Sandy Bridge, Ivy Bridge or Haswell) has to do with a much simpler ring or ring-like topology. As explained in the aforementioned book by Culler et al., N cores on a bi-directional ring means a network diameter of $N/2$, an average distance of $N/3$ and a bisection of only two links. This is far from the sophisticated Butterfly that gave its name to the BBN machine.

This paper is build in two parts. In the first, we will examine the consequences of the ring bus inside the Xeon Phi™. It is the mean by which core communicates their cache lines, and it must be taken into account to achieve good performance. In the second part, we will evaluate a few classic algorithms for barriers on the Xeon Phi™, and how the topology affect them. We will also demonstrate that not only the algorithm and its implementation are important, but that the placement of the data is as well on the Xeon Phi™.

II. Phi™Ring Bus and core-to-core speed

Intel®documentation for the Xeon Phi™ [10] section 2.1.3 explains some details of the cache hierarchy. One noticeable feature is the distributed tag directories (DTD) for the 2nd level cache. Each of the 64 DTDs hold the coherence data for 1/64th of the memory space via a hashing function. Whenever a core requires a line that is not immediately available (i.e. there is no valid usable copy in either its L1 or its L2 cache), a round-trip to the DTD responsible for the cache line is necessary before the line can be retrieved. As there is only 62 architectured cores (with only up to 61 active on most commercially available devices), DTDs are not exclusively collocated with cores. Not all cache line are equal for core-to-core communication, with cache line whose DTD is closer to a core having a lower latency for that core.

For a pair of cores communicating via a cache line, there are two possibilities:

1) The DTD responsible for the cache line is collocated with one of the core. In this case, the collocated core will have fast access, and the other core will require a round-trip. If the two cores are “close” on the ring bus, then communication will be fast.
2) The DTD responsible for the cache line is not collocated with either core. Both cores will require round-trip when requesting the cache line from the other. Depending on the relative position on the ring bus of the three elements, speed will vary.

To measure the core-to-core latency, we used a ping-pong code whose idea came from Volkov [11]. Two threads share a single variable. The first thread increments the variable, then wait for the other threads to increment it to the next value. During the wait, the cache line is prefetched in the exclusive state so that the next iteration will avoid an additional shared to exclusive transition. The second thread behave symmetrically, prefetching exclusively until it sees the first thread’s increment, then increment the variable itself. Both threads loop until a fixed number of increments has been reached. The result is a ping-pong effect of the shared cache line between the two cores.

As the DTD is selected by hashing the physical address of the cache line using an undocumented hash function, experiments are hard to reproduce. Multiple run of the same code are unlikely to yield the same physical address for the shared variable in each run. Because of this, we had to implement a specific user-land mechanism of trial-and-error to make the physical address reproducible. A physical address assumed to be available on the system is selected. A single huge page is allocated to a fixed virtual address. The page address is translated to the physical address. If the physical address matches our selected address, it is then re-mapped to a reproducible virtual address. If not, another page is allocated and translated, until the chosen physical page is obtained. Note that in some cases the page is never obtained because it is in use elsewhere in the system, requiring a restart of the Xeon Phi^{TM}. The only criteria to select the physical address is that it is easily obtained after a restart, as we only need to fix the address to ensure reproducibility. It is certainly possible to implement a better mechanism in kernel-mode, but that is out of the scope of this paper.

Figure 1 plots the observed minimum ping-pong time between each pair of cores (core numbers are on the X and Y coordinate, time in on the Z coordinate). The various colors represent different ranges of time. It is clearly visible that the data stored in the cache line at this particular physical address (the first of the page we had chosen) moves around much faster when used by cores close to core 23. Figure 2 represents the same measurement, but done using a cache line situated 192 bytes (3 cache lines) further into the allocated page. The graph is similar to the previous one, but with the minimum located in a different place of the 2D coordinates; this times cores 2 and 3 are the most efficient at communicating with this cache line. All tests were run on a state-of-the-art Xeon Phi^{TM}7120P, with 61 cores at 1.238 GHz and the latest software stack from Intel.

Fixing the address and trying all pairs is one way to measure the effect; the other is to fix the pair of cores and try many possible cache line to observer the variation. We did this for the combination of core 23 and each of the other cores in figure 3. The curves are what is expected: the average is extremely stable (with an oddity for core 37 that might be a measurement error), and fairly symmetrical minimum and maximum. That is to be expected: if two cores are physically close on the ring, then a cache line with a DTD close to either one will be extremely fast. But a cache line with a DTD very far away will cause an expensive round-trip for both cores as well. On the other hand, if two cores are very far away on the ring (for core 23, that would be core 55), no cache line can provide excellent performance as the DTD cannot be close to both core, but no cache line will perform very poorly, as the DTD cannot be far from both core either.

Figure 4 is the plotting of all minimums between all pair of cores. This is the minimum ping-pong time achievable between any two cores when selecting the best possible cache line. The 2D mapping at the bottom shows clearly that the network topology is not quite a perfect ring. There is clearly a pattern of four by four “squares”, with a clear transition between them. The well-defined transition between “squares” indicates an irregularity in the topology, or a feature other than a core (such as the memory controllers or the disabled cores) taking up space on the network.

III. Barriers and communications of shared variables

Barriers are a very important consideration for parallel program. An excellent justification and study of related work concerning barriers on the Xeon Phi^{TM}can be found in the work of Caballero et al. [12]. They compare the implementation of an OpenMP barrier using the Phi^{TM}SIMD instructions versus the implementation supplied by Intel in their tools. For our work we are interested in a wider range of algorithms, and the influence of data placement on them.

All synchronization primitives including barriers require some sort of shared data between threads to do their work. The amount of data required, the degree of sharing and the set of threads sharing each atom of data is highly dependent on the specific algorithm and possibly its implementation.

A. Centralized barrier

For instance, the simplest of all barrier in the centralized barrier: a single shared counter is used to assert the number of threads that have reached the barrier. In our implementation, we use a separate sense-reversal flag to free all the threads once the last one reach the barrier. Most of the cost of the barrier is in the need for all threads to update the counter, that is, to acquire the cache line in the ’Modified (M)’ state. This causes a very large amount of cache coherency traffic. The second cost is the release phase. When the last thread updates the flag, every other copy is invalidated, and then every cores must reload the cache line in the ’Shared (S)’ state (the flag is read-only for them).

Figure 5 plots the minimum time required to complete a barrier. 32 different runs were done, each using a different starting point when allocating the shared variable. Run 0 was done with no offset relative to the large page (the same physical large page selected in section II), i.e. using the first cache line in the page. Run 1 was done starting with the second cache line in the page, or an offset of 1. Run 2 was done with an offset of 2, and so on. The plot include the minimum and the maximum values across those 32 runs, plus the relative difference between the minimum and the maximum. All runs were done with four threads per core, using the compact affinity.

The relative cost tells the story very clearly: when a small number of cores is in use, the difference between the best-case scenario and the worst-case scenario can be a factor of more than two, despite the fact than 32 different cache lines do not cover all the possible DTDs. There possibly could be cache lines that are faster, or slower, for the various configurations of threads. Another observation is the near-linear decrease in relative cost with the number of threads. This is expected as well: for a small number of cores, near to each other on the ring, some cache line’s DTD will be very close to the cores, where others will be very far. When most or all cores are involved, the distance is averaged, and the difference is only about 20% - still a significant value.

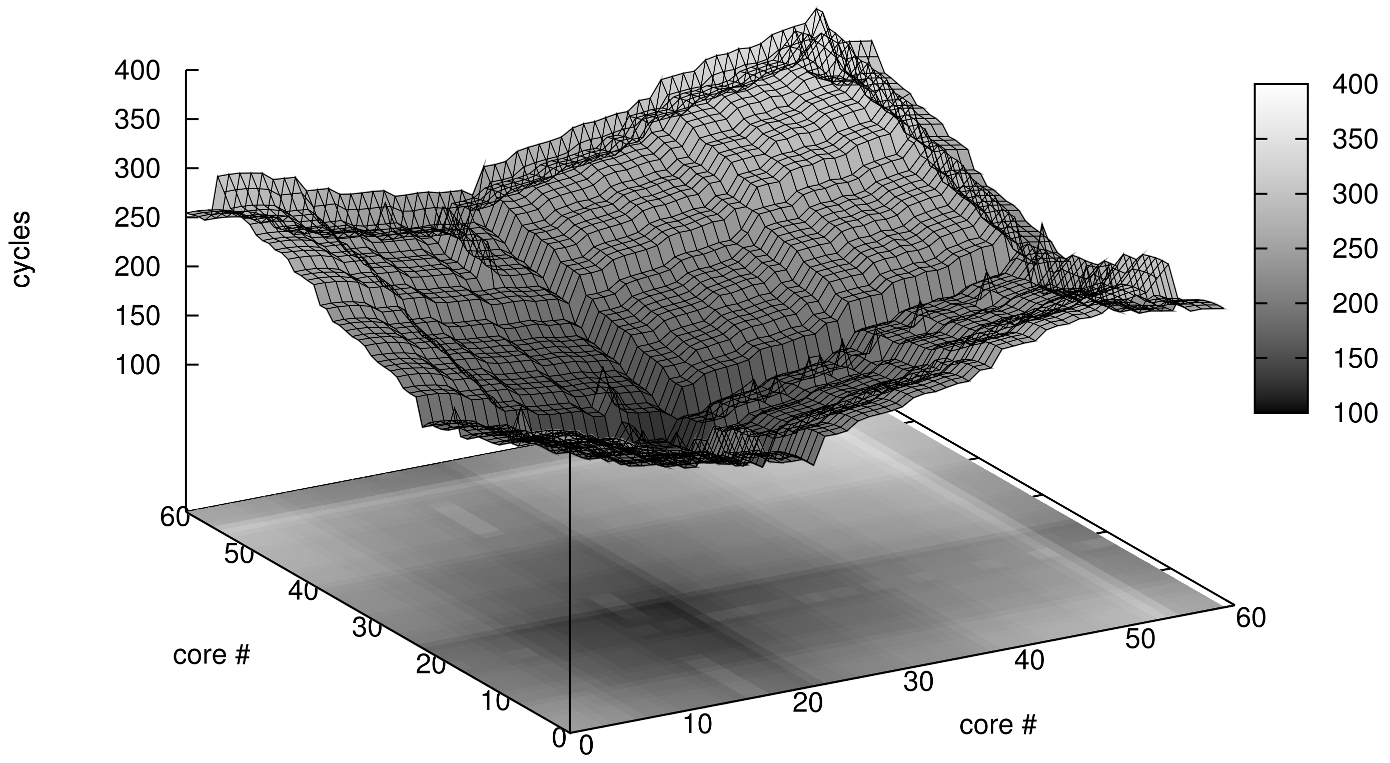

Fig. 1. Minimum ping-pong time for all pair of cores on a Xeon Phi^{TM}7120P, offset 0x0000

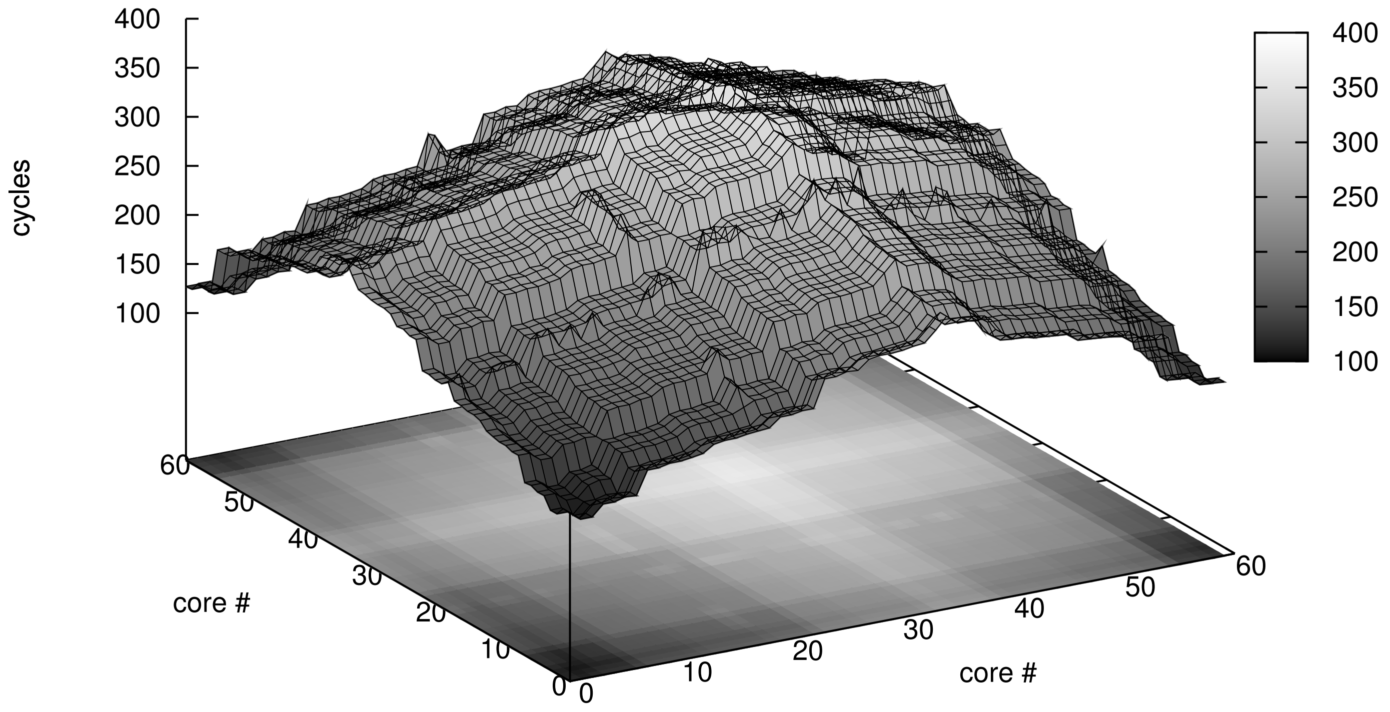

Fig. 2. Minimum ping-pong time for all pair of cores on a Xeon Phi^{TM}7120P, offset 0x00c0

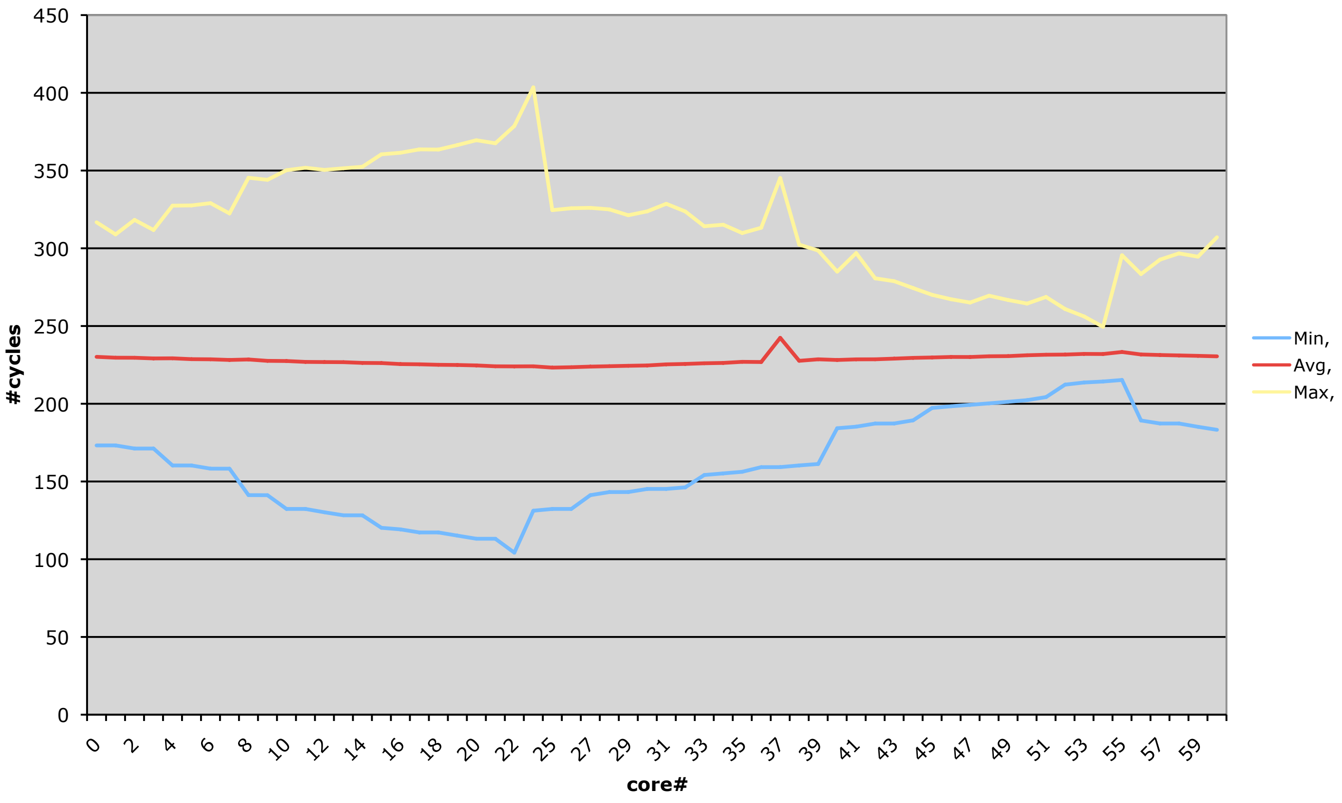

Fig. 3. Ping-pong time (minimum, average and maximum across 16384 lines) for core 23 to all other cores.

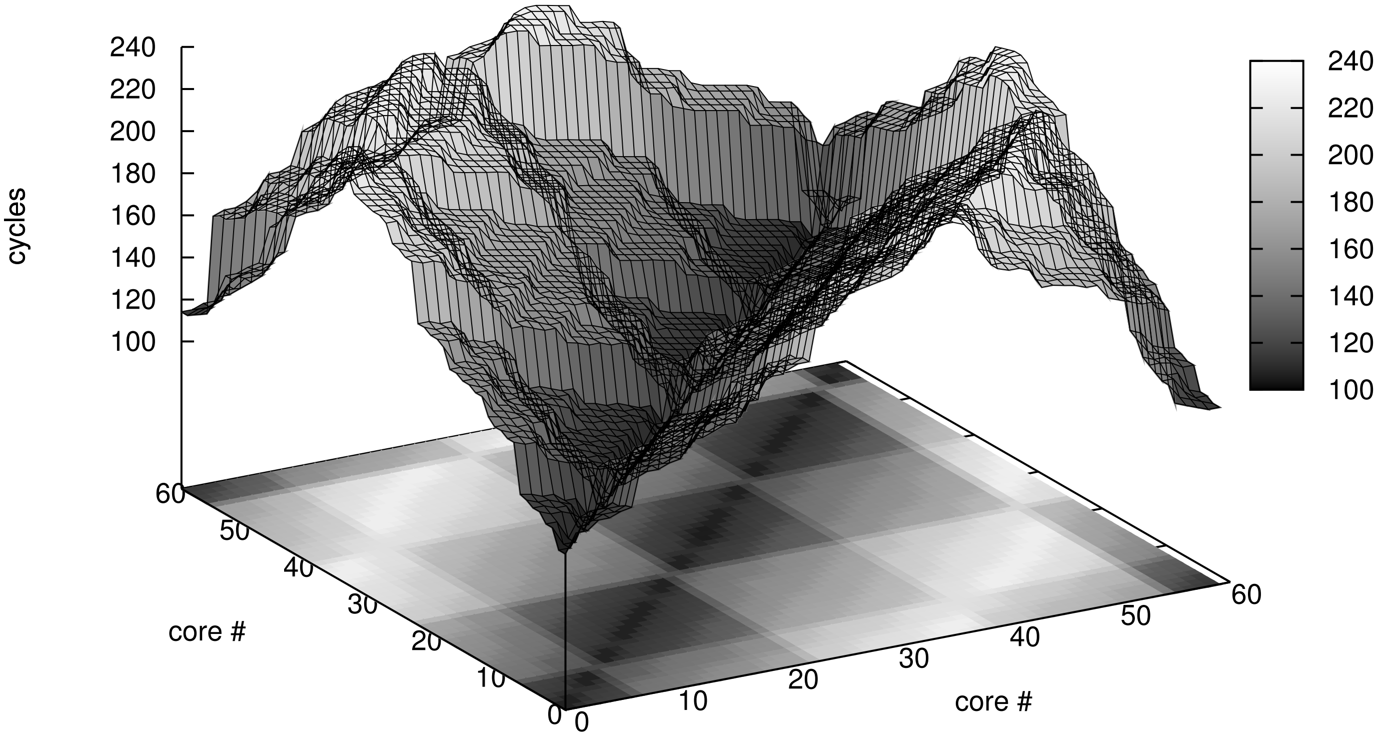

Fig. 4. Minimum ping-pong time for all pair of cores on a Xeon Phi^{TM}7120P across all offsets

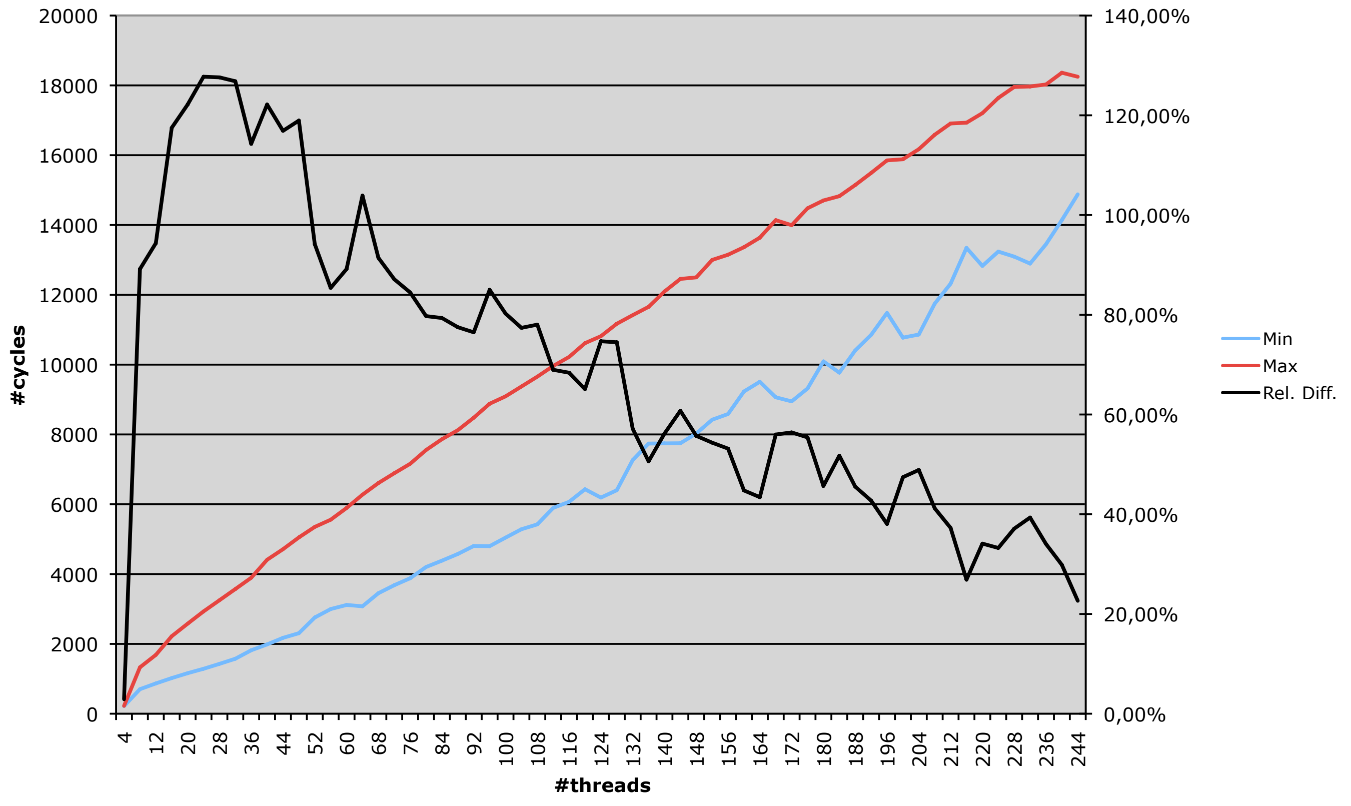

Fig. 5. Time for a centralized barriers vs. number of threads.

B. *Dissemination barrier*

The dissemination barrier originally by Hensgen et al. [13] is a classic algorithm for barrier, similar to the Butterfly algorithm by Brooks [3]. The principle is to have each thread synchronizes with a single partner thread in an "instance", and repeats until completion. Therefore log2(number of threads) "instances" are required to complete the full barrier.

Our implementation puts each element of the "Answers" array of the original paper into its own cache line, minimizing conflict at the expense of memory consumption. Each shared cache line is therefore used only once for each instance of the barrier: at every "instance", one thread in a partnership write the value, while the other thread in the partnership reads it. Because of the way the partnership are established in the dissemination barrier in each "instance", threads are not paired by proximity - the first "instance" will involve neighbor threads, but the last will offset them by half the total number of threads.

Unlike the centralized barrier, the dissemination barrier has many shared data. Our implementation does not allocate them independently; they are allocated in consecutive cache lines in our large page. We therefore do not expect a very large discrepancy in performance between the best and worse case scenario for two reasons:

1) The placement in memory of the shared data will influence some of the first "instances" (where the distance between threads is low, with the exception of "instance" 0 and 1 where the threads share a core), but it will have less and less influence on later "instances" when the distance between partners is higher.
2) The large number of shared cache line should average performance, as "instances" are synchronous - if only one partnership is slow, the entire "instance" is slowed down.

Indeed, figure 6 show that except for small number of cores (where both reasons are invalid), the relative difference between the minimum and maximum over our 32 offsets is quite low, below 6% from 60 threads to 244 threads, below 5% from 104 threads and below 4% from 192 threads. The plot also show the characteristic behavior from the dissemination barrier, with a new, higher plateau every time the base-2 logarithm of the number of threads is raised by one (i.e., the number of thread crosses a power-of-2 threshold).

C. *Software Combining Tree barrier*

The software combining tree barrier from Yew et al. [4] is another classic implementation for barriers. This time, groups of threads are synchronized by a group-local counter (similar to the centralized barrier). A master thread in each group subsequently synchronizes, by the same mechanism, with its peer. The process can be repeated an arbitrary number of time in a tree structure.

Our implementation is based upon the optimized version presented by Mellor-Crummey et al. [5]. The tree can be built from nodes of arbitrary fan-in values. After exploring many combinations, we retained a configuration with 4 threads per leaf group (i.e. an single core worth of threads), and 8 groups per groups above that - so at most three levels for 33 to 244 threads. This require less memory than the dissemination barrier. All leaf group will spin on a local variable (as it is not shared with any other core, the cache line will not leave the L1 cache). The up to 8 intermediate groups will share a single variable with only neighboring cores. Finally, the root level group will be accessed by sparsely distributed cores across the ring.

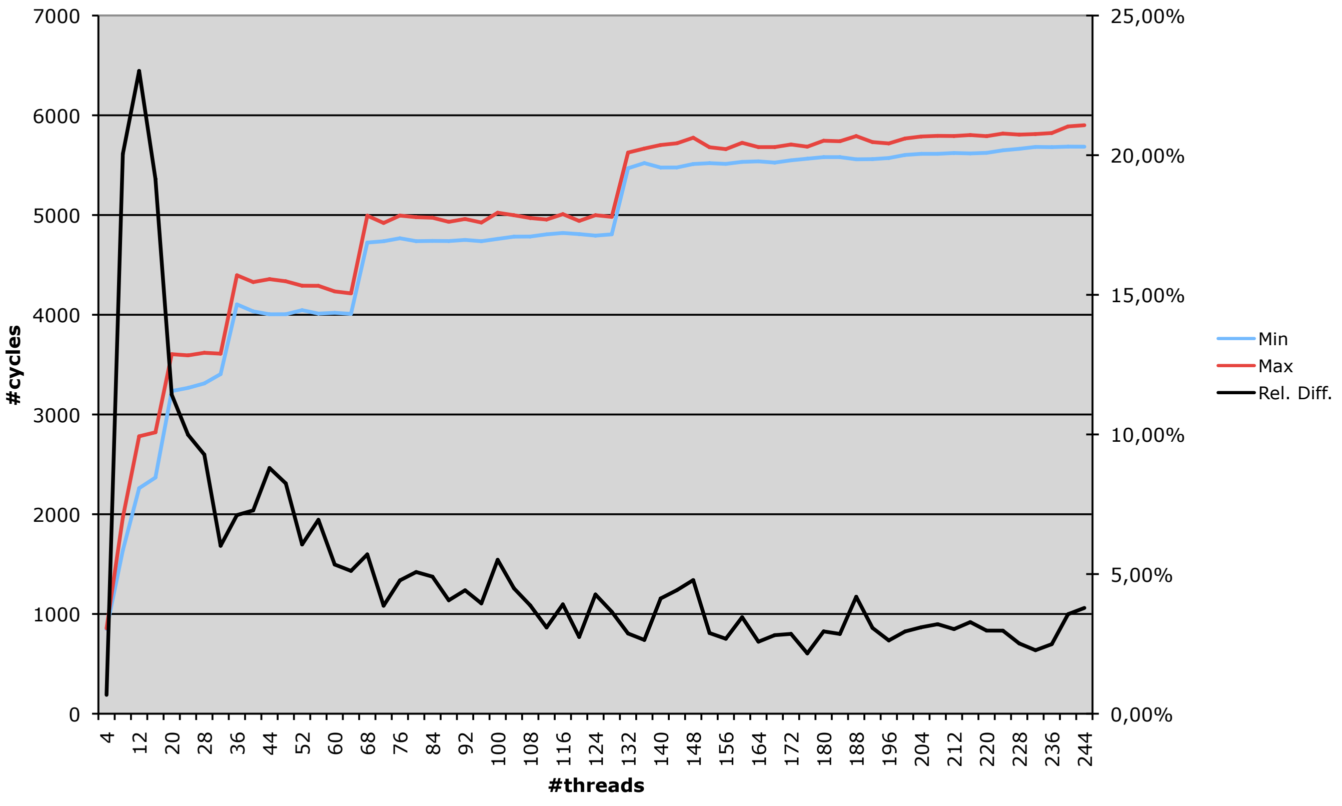

Fig. 6. Time for a dissemination barriers vs. number of threads.

Figure 7 show the results for this type of barrier, again displaying minimum and maximum observed valued across 32 starting points for allocation in a pre-set physical page. With fewer variables than the dissemination barrier, it is more sensitive to placement, but less so than the single-variable centralized barrier. Only the intermediate nodes in the tree should be sensitive: the leaf node do not share, while the root node is accessed from all area of the ring.

D. Mellor-Crummey & Scott barrier

This is the barrier proposed by Mellor-Crummey & Scott in [5]. It is also based on a tree structure. The original justification for this new barrier was that the software combining tree requires threads to spin on non-statically localized, shared variable. That is, each thread must spins on a variable whose address cannot be forced to be local to the thread. However, they recognize that it is not a problem for "broadcast-based cache-coherent machines". It is indeed not an issue for any cache-coherent machine such as the Xeon Phi^{TM}, as the threads will spin read-only on local copies of the variable, with only the release update causing coherency traffic. Another justification for the new barrier is that it does not use atomic operation, relying exclusively on load and store. But on the Xeon Phi^{TM}, like all x86-based machine, atomic operation on a variable in the 'Exclusive (E)' or 'Modified (M)' state in the local L1 cache is very efficient.

We first did a by-the-book implementation of this barrier, but did not obtain good performance - for large number of threads, it was more than twice as slow as the dissemination barrier. We then replaced the hard-coded fan-in of 4 and fan-out of 2 by configurable values. One implementation could use values up to 8 (by using an 8-bytes `long long` variable instead of a 4-bytes `int` for the whole array), while a second could use values up to 64 (by using the Xeon Phi^{TM}SIMD instructions, in an approach similar to that of Caballero et al. [12]). While larger values helped with performance, it still could not match either our dissemination barrier or our configurable tree barrier. Removing the pre-computed pointers to the parent and child (and computing targets on the fly) also helped, but the resulting barrier was still slower than an optimized tree barrier.

IV. Improving the Tree Barrier with Address Selection

We choose to try and improve the performance of the tree barrier for two reasons. One, it was already the fastest barrier we had for large number of threads on the Xeon Phi^{TM}. Two, our implementation already allocate each node independently, making fixed-address nodes easier.

We choose to only fix the address in the second, intermediate level of the tree. The root node is going to be accessed by cores scattered all over the ring, so there is little room for improvement. The leaf nodes are only ever accessed by a single core in the (8,4,4) configuration, so the position of the DTD is mostly irrelevant as long as the cache lines are kept in the local L1 or L2 cache. Note that we did not alter the structure of the tree or the mapping of threads to the tree to take into account the topological irregularities previously shown in figure 4.

Two different variables are shared in our implementation: the atomically updated "count" variable, and the spinning variable "local_sense". However, our implementation places them in consecutive cache lines, as they are part of the same structure but with explicit alignment requirement. So without modifying the implementation, we could only select a "good" address for one

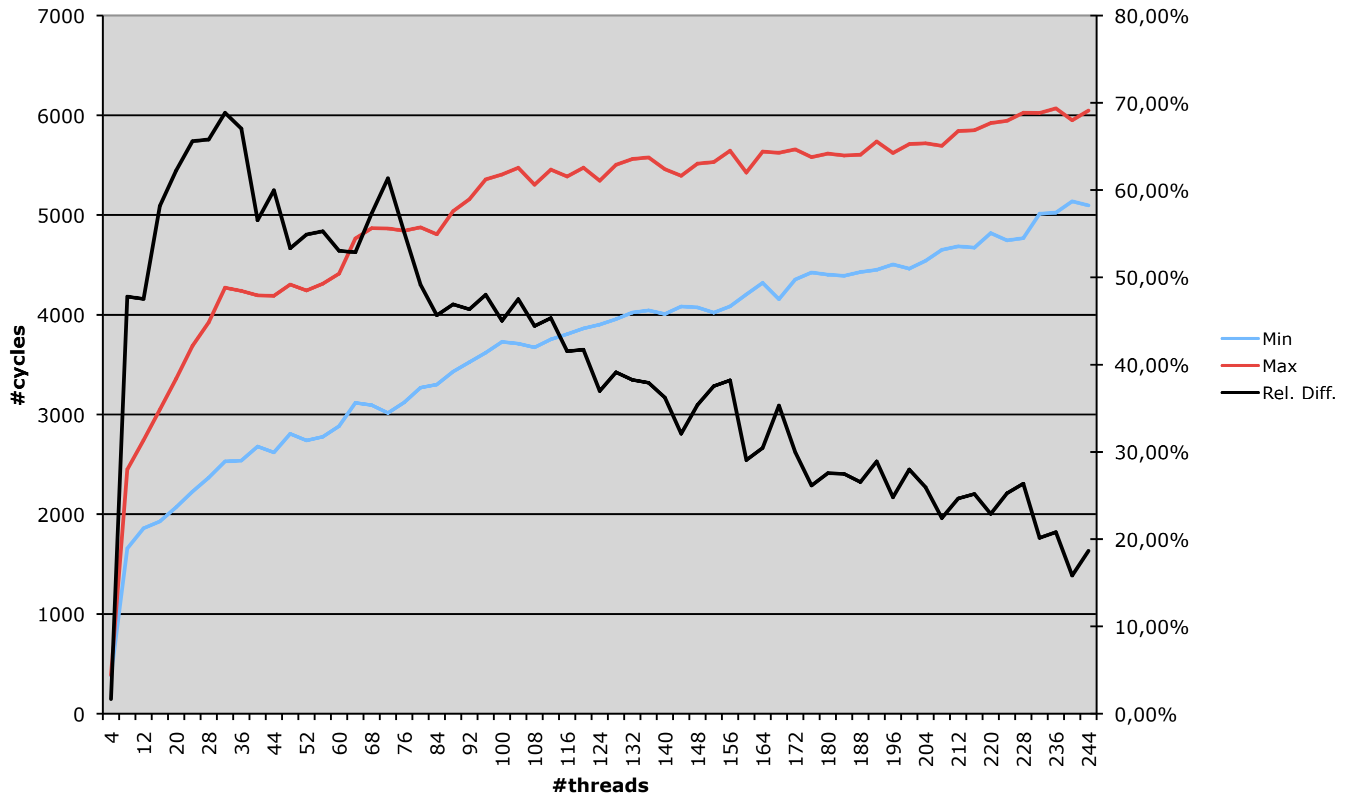

Fig. 7. Time for three-level (8,8,4) barriers vs. number of threads.

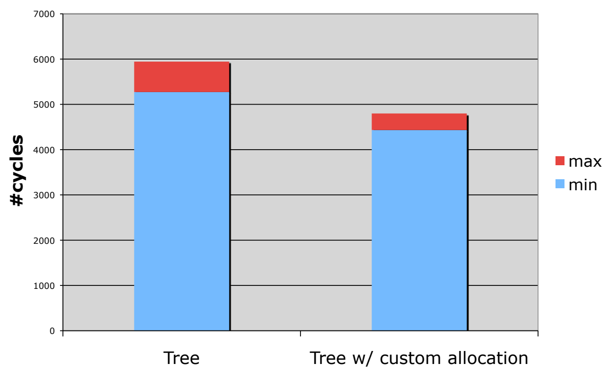

Fig. 8. Effect of hard-coding addresses of shared variables for a tree barrier on 244 threads

of the two. We choose to pin the address of the "count" variable, the most obvious candidate as it has read-write contention.

To select the specific addresses, we first did an exhaustive study of our selected large page. For each cache line, we measured the ping-pong latency (using the technique explained in section II) between every pair of cores. We then extracted the subset of cores that had low-latency access to all other in the subset. We also extracted which core or cores had low-latency access to several neighbors. From this information, and from the known topology of the tree (8,8,4), we could select cache lines that were efficient for communicating inside each group of 8 cores, and that did not conflict with each other.

The results for 244 threads is shown in figure 8. The minimum and maximum values are again across 32 different base addresses for the allocation of other shared data. Fixing the addresses from the intermediate layer in the tree offers a gain of between 15% and 20% for this type of barrier. We also show in figure 9 the results from the "barrier" benchmark in the EPCC OpenMP Microbenchmarks V3 [14][1]. In those results the first labelled "OpenMP" refers to the Intel-supplied OpenMP barrier. The other two replaces the barrier directive by a call to our barrier. The first labelled "Tree" refers to our tree barrier implementation using the default addresses, and the second labelled "Tree w/ addr. select." refers to our tree barrier implementation using the address selection process. The first four columns are rounded results from the benchmark in microseconds, with the fifth "Avg. (cycles)" containing the "Average" value converted to CPU cycles. Both our implementations are significantly faster than the OpenMP reference implementation. Fixing the addresses to carefully selected values shows a 15% improvement in this benchmark as well.

V. Conclusion & Future Work

In this paper we have examined the behavior of the ring bus and the cache implementation on the Xeon Phi^{TM}. We have seen the physical address of a shared variable has an influence on the performance when communicating between cores. We have also examined a few classic algorithms for barriers on the Xeon Phi^{TM}, and seen how they were affected by this specific behavior

[1]Sources for the EPCC Microbenchmarks Suite are available at http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html

Barrier	Average	Min	Max	Std. Dev.	Avg. (cycles)
OpenMP	11.06	10.72	12.34	0.458	13691
Tree	4.573	4.425	5.206	0.217	5661
Tree w/ addr. select.	3.873	3.775	4.414	0.181	4794

Fig. 9. EPCC OpenMP Microbenchmarks V3 results for "OpenMP", "tree", and "tree with fixed address" barriers

of the Xeon Phi™. We have shown that carefully selecting the physical location of a shared variable can lead to significant improvements for some synchronization primitives, with a gain of 15% on the software combining tree barrier. The summary of recommendations made by Mellor-Crummey & et al. in [5] holds true today, with the best choice remaining tree-based algorithms for barriers, but with the addition of address selection on hardware such as the Xeon Phi™.

Selecting the addresses is however still an open issue. For this work, it was done by hand from extensive measurements for the full complement of 244 threads. However for this work to become useful in practice, the selection process and the barrier-building will have to be automated. In addition, a selection valid for one particular Xeon Phi™cannot be assumed to be valid for another: as at most 61 cores are active from the 62 architectured in the silicon, the best selection for a set of cores may vary from device to device. And the physical addressing of the memory cannot be relied to be identical from one implementation to another. It is likely that a kernel boot-time procedure will be required to reserve a reliably available set of addresses, combined with a one-time procedure to select the set of addresses for various threads combinations.

Another aspect would be to extend this work for mutual exclusion (locks) in set of threads and/or cores. Locks also rely on shared variables, but with requirements different from that of barriers, and could benefit from address selection as well.

Acknowledgments. Intel, Xeon, Xeon Phi and Many Integrated Core are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and names are the property of their respective owner. This work has been partially funded by the H4H-PerfCloud ITEA2 project http://www.h4h-itea2.org. The authors wish to thank Intel France for supplying the Xeon Phi used in this work.

References

[1] 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 Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 235–246. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.23

[2] J. Sartori and R. Kumar, "Low-overhead, high-speed multi-core barrier synchronization," in Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers, ser. HiPEAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 18–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11515-8_4

[3] I. Brooks, Eugene D., "The butterfly barrier," International Journal of Parallel Programming, vol. 15, no. 4, pp. 295–307, 1986. [Online]. Available: http://dx.doi.org/10.1007/BF01407877

[4] P.-C. Yew, N.-F. Tzeng, and D. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," IEEE Transactions on Computers, vol. 36, no. 4, pp. 388–395, 1987.

[5] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Trans. Comput. Syst., vol. 9, no. 1, pp. 21–65, Feb. 1991. [Online]. Available: http://doi.acm.org/10.1145/103727.103729

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," SIGARCH Comput. Archit. News, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[7] Intel Corporation. (2013) Intel®xeon phi™coprocessor - the architecture. [Online]. Available: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[8] ——. (2013) Avx-512 instructions. [Online]. Available: http://software.intel.com/en-us/blogs/2013/avx-512-instructions

[9] D. E. Culler, A. Gupta, and J. P. Singh, Parallel Computer Architecture: A Hardware/Software Approach, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[10] Intel Corporation, Intel®Xeon Phi™Coprocessor System Software Developers Guide, June 2013, no. 328207-002. [Online]. Available: http://software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemssoftwaredevelopersguide_0.pdf

[11] V. Volkov, "Intro to MIC performance," presentation at the Berkeley Benchmarking and Optimization Group, 2012.

[12] D. Caballero, A. Duran, and X. Martorell, "An openmp* barrier using simd instructions for intel®xeon phi™coprocessor," in OpenMP in the Era of Low Power Devices and Accelerators, ser. Lecture Notes in Computer Science, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Springer Berlin Heidelberg, 2013, vol. 8122, pp. 99–113. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40698-0_8

[13] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," Int. J. Parallel Program., vol. 17, no. 1, pp. 1–17, Feb. 1988. [Online]. Available: http://dx.doi.org/10.1007/BF01379320

[14] J. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for openmp tasks," in OpenMP in a Heterogeneous World, ser. Lecture Notes in Computer Science, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds. Springer Berlin Heidelberg, 2012, vol. 7312, pp. 271–274. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30961-8_24