Hardware and Software Tradeoffs for Task Synchronization on Manycore Architectures

Yonghong Yan[1], Sanjay Chatterjee[1], Daniel A. Orozco[2], Elkin Garcia[2], Zoran Budimlić[1], Jun Shirako[1], Robert S. Pavel[2], Guang R. Gao[2], and Vivek Sarkar[1]

[1] Department of Computer Science, Rice University
{yanyh,sanjay.chatterjee,zoran,shirako,vsarkar}@rice.edu
[2] Department of Electrical Engineering, University of Delaware
{egarcia@,orozco@eecis.,rspavel@,ggao@capsl.}udel.edu

Abstract. Manycore architectures – hundreds to thousands of cores per processor – are seen by many as a natural evolution of multicore processors. To take advantage of this massive parallelism in practice requires a productive parallel programming model, and an efficient runtime for the scheduling and coordination of concurrent tasks. A critical prerequisite for an efficient runtime is a scalable synchronization mechanism to support task coordination at different levels of granularity.

This paper describes the implementation of a high-level synchronization construct called *phasers* on the IBM Cyclops64 manycore processor, and compares phasers to lower-level synchronization primitives currently available to Cyclops64 programmers. Phasers support synchronization of dynamic tasks by allowing tasks to register and deregister with a phaser object. It provides a general unification of point-to-point and collective synchronizations with easy-to-use interfaces, thereby offering productivity advantages over hardware primitives when used on manycores. We have experimented with several approaches to phaser implementation using software, hardware and a combination of both to explore their portability and performance. The results show that a highly-optimized phaser implementation delivered comparable performance to that obtained with lower-level synchronization primitives. We also demonstrate the success of the hardware optimizations proposed for phasers.

1 Introduction

Manycore architectures, with hundreds to thousands of cores per processor, are seen by many as a natural evolution of multicore processors. In practice, a productive parallel programming model, and an efficient runtime for thread execution and coordination, are essential to take advantage of this massive parallelism. Programming models using dynamic task parallelism, such as the ones introduced in the programming languages of the DARPA HPCS program (X10 [1] and Chapel [2]), present a promising approach to productive parallel programming on manycore processors. However, the overhead of communication and synchronization between concurrent tasks typically presents one of the greatest obstacles

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 112–123, 2011.

to achieving high performance and scalability on parallel systems. To support diverse workloads on manycore architectures, synchronization mechanisms that provide high-level operations such as barrier using different granularity levels, would be highly desirable.

Phasers, first introduced in the Habanero-Java multicore programming system [3], are synchronization constructs for task parallel programs. Phasers unify barrier operation and point-to-point synchronization in a single interface, and feature deadlock-freedom and phase-ordering. The current Habanero-Java phaser implemented on a Java virtual machine does not leverage hardware support for synchronization and only works on top of a work-sharing runtime, a much less scalable choice for task parallel runtime than workstealing [4]. In this paper, we present the evaluations of phaser implementations in a workstealing runtime using a C-based Habanero-C parallel programming language. Using the IBM Cyclops64 (C64) manycore architecture [5], we have experimented with several approaches to phaser implementations using software, hardware, and a combination of both to explore their portability and performance. The results show that a highly-optimized phaser implementation delivered comparable performance to that obtained with lower-level synchronization primitives. We also demonstrate the success of the hardware optimizations proposed for phasers.

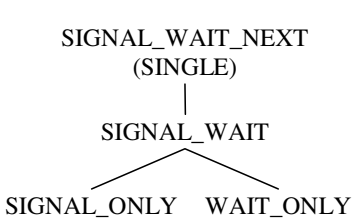
The contributions of this work includes the following. First, we have provided a highly-optimized spin-based implementation of phasers. It is software-based and portable across POSIX-compliant systems. Secondly, we have optimized a phaser implementation that leverages hardware support for synchronization to deliver superior performance over the software approach while maintaining the same interfaces and features. Finally, we have provided a runtime that is able to switch between software and hardware based implementations to better leverage hardware support, if available.

In the rest of the paper, Section 2 presents the Habanero-C task parallel programming language, and the portable software implementation of phasers. Section 3 describes the phaser implementations on Cyclops64, taking advantage of its hardware features. Section 4 presents the experimental results. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2 Asynchronous Task Parallelism and Software Phasers

Phasers were implemented in the Habanero-C research language developed at Rice University. Habanero-C language has two basic primitives, borrowed from X10 [1], for asynchronous task parallel programming: async and finish. The async statement, *async* ⟨*stmt*⟩, causes the parent task to fork a new child task that may execute ⟨*stmt*⟩ in parallel with the parent task. Execution of the async statement returns immediately, i.e. the parent task does not wait for the child task to complete. The finish statement, *finish* ⟨*stmt*⟩, performs a join operation on all the tasks created within ⟨*stmt*⟩, including transitively spawned tasks.

The async and finish constructs are simpler than the conventional pthread_create and pthread_join APIs, and more flexible than the Cilk spawn and sync keywords [6] and OpenMP task and taskwait directives. For example, the sync or



(a) Phaser Mode Lattice

```
1:  finish {
2:    new_phaser(SIGNAL_WAIT);
3:    for (int j=0;  j<ntasks;  j++)
4:      async phased IN(j) {
5:        for (int i=0; i<innerreps; i++) {
6:          delay(delaylength);
7:          printf("Task %d at step %d!\n", j, i);
8:          next; }
9:  }   }
```

(b) Barrier Example Using Phasers

Fig. 1. Phaser Mode Lattice and Barrier Example

taskwait constructs can only synchronize tasks that are created within the same function scope. Using async and finish as a foundation, we were able to easily experiment with different choices of task parallelism and target platforms.

2.1 Asynchronous Task Synchronization Using Phasers

There are several nice features to use phasers as synchronization constructs with the async and finish task parallel programming model. First, phasers unify collective and point-to-point synchronization in a single set of programming interfaces. The interfaces are ease of use, improving programmer productivity in parallel programming and debugging. Secondly, phasers have two safety properties: deadlock-freedom and phase-ordering [3]. These properties, along with the generality of its use for dynamic parallelism, distinguish phasers from other synchronization constructs in past works including barriers, counting semaphores [7], and X10 clocks [1]. Thirdly, in implementation, phasers have been integrated with a workstealing scheduler that was used in Habanero-C runtime. As a new contribution of this paper, the implementation provided reference solutions to how to map asynchronous tasks with hardware threads when performing synchronization operations. The details of these solutions are discussed in Section 3.

Figure 1(b) shows an example of using phasers to implement a barrier among multiple asynchronously created tasks. The async statement in line 4 and the j-for loop create ntasks child tasks, each registering with the phaser created in line 2 in the same mode as in the master task. The next statement in line 8 is the actual barrier wait; each task waits until all tasks arrive at this point in each iteration of the i-for loop. The first next operation of each task causes itself to wait for the master task to do next operation or to deregister. When the master task reaches the end of the finish scope, it deregisters from the phaser so all child tasks continue and synchronize by themselves in each iteration.

2.2 Software Phasers in Habanero-C

As a synchronization object for dynamic tasks, a phaser has two phases, the signal phase and wait phase, each represented by a counter. Given the mode a task registers with a phaser, a phaser operation could be either or both of a

signal and a wait operation, which advances the corresponding phase counter. A task registration is represented by a unique synchronization object, named *sync*, which contains the registration mode and the current signal and wait phase. In order to guarantee deadlock freedom, a child task can only register in a mode that is the same as or below the mode in the parent task according to the phaser mode lattice shown in Figure 1(a). When signaling on a phaser, a task simply increments the signal phase in the *sync* object. The next operation has the effect of advancing each phaser with which a task registers to its next phase, thereby synchronizing all tasks registering with the same phaser. Details operation semantics are described in [3].

Hierarchical Phaser Implementation: The phaser implementation discussed above has used a single master task to advance to its next phase. While the single master approach provides an effective solution for modest levels of parallelism, it quickly becomes a scalability bottleneck as the number of tasks increases. To address this limitation, we have used an approach based on hierarchical phasers [8] for scalable synchronization.

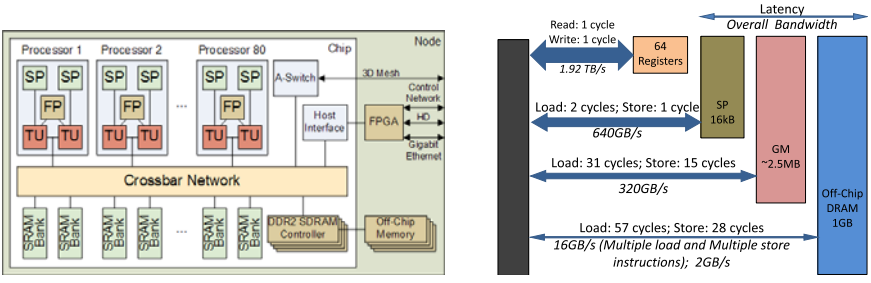
The hierarchical phaser employs a tree of sub-masters, instead of a single master, as in the case of a flat phaser. Tree-based barriers have the advantage that gather operations in the same level (tier) can be executed in parallel by sub-masters. Also, in cases when the hierarchy of sub-masters follows the natural hierarchy in the hardware, each sub-master will leverage data locality among workers in its sub-group. Although the initialization overhead of building a tree is greater than the flat phasers, the runtime of hierarchical phasers outperform the flat phasers heavily on higher number of tasks, as discussed soon in Section 4.

3 Hardware Support in Phasers

The counter-based phaser implementation is a spin-based software approach, also referred to as busy-wait. It consumes both CPU cycles and memory bandwidth, and may quickly become a scalability bottleneck when a large number of tasks are involved in a phaser operations, as in manycores. Recent trends in manycore processor design use tiled architectures to reduce the dependency on the memory bus [9] and to localize synchronizations. In this Section, we explore a phaser implementation that leverages hardware support for synchronization using the IBM Cyclops64 (C64) manycore chip [5] as our evaluation platform.

3.1 Cyclops64 Manycore Architecture

The IBM Cyclops64 is a massively parallel architecture initially developed by IBM as part of the Blue Gene project. As shown in Figure 2, a C64 processor features 80 processing cores on a chip, with two hardware thread units per core that share one 64-bit floating point unit. Each core can issue one double precision floating point Multiply Add instruction per cycle, for a peak performance of 80 GFLOPS per chip when running at 500MHz. The processor chip includes a high-bandwidth on-chip crossbar network with a total bandwidth of 384 GB/s. C64



(a) C64 Chip Architecture (b) C64 Memory Hierarchy

Fig. 2. Cyclops64 Architecture Details

employs three-levels of software-managed memory hierarchy, with the Scratch-Pad (SP) currently used to hold thread-specific data. Each hardware thread unit has a high-speed on-chip SRAM of 32KB that can be used as a cache.

C64 utilizes a dedicated signal bus (SIGB) that allows thread synchronization without any memory bus interference. The SIGB connecting all threads on a chip can be used for broadcast operations taking less than 10 clock cycles, enabling efficient barrier operations and mutual exclusion synchronization. Fast point-to-point signal/wait operations are directly supported by hardware interrupts, with costs on the order of tens of cycles.

The C64 tool chain includes a highly efficient threading library, named TiNy-Threads (TNT) [5], which uses the C64 hardware support to implement threading primitives. Additionally, TNT provides APIs that can be used to access the hardware synchronization primitives to allow for suspension of threads, and including and excluding specific threads from barriers, as shown in Table 1.

Table 1. Cyclops64 TNT APIs for Hardware Synchronization Primitives

Name	Description
tnt_suspend()	Suspend current thread
tnt_awake (const tnt_desc_t)	Awaken a suspended thread
tnt_barrier_include (tnt_barrier_t *)	Join in the next barrier wait operation
tnt_barrier_exclude (tnt_barrier_t *)	Withdraw from the next barrier wait operation
tnt_barrier_wait (tnt_barrier_t *)	Wait until all threads arrive this point

3.2 Optimization Using Hardware Barriers

Barrier operations using phasers can be optimized in manycore architectures that offer direct hardware support for barriers, such as C64. The phaser runtime is able to detect if a phaser operation specified by the user program is equivalent to a barrier operation by checking whether all phasers are registered in SIGNAL_WAIT mode. If so, the underlying hardware support is used directly to perform the barrier operation.

Implementing a hardware barrier in a phaser requires threads to include themselves in the barrier by calling tnt_barrier_include. This requirement is particularly interesting in a workstealing environment due to the fact that the worker that executes the task which is participating in the barrier, has to include itself in the hardware barrier. In workstealing, we cannot include the worker a priori in the barrier. The Habanero-C runtime only includes a worker in the hardware barrier when it is ready to execute a task.

3.3 Optimization Using Thread Suspend and Awake

The TNT API provides functions to suspend a thread and to awake a sleeping thread. A suspend instruction temporarily stops execution in a non-preemptive way, and a signal instruction awakes the sleeping task. Using thread suspend and awake mechanism in place of the busy-wait approach reduces memory bandwidth pressure because all waiting tasks can suspend themselves instead of spinning. The master can collect all the signals from waiting tasks and finally signals the suspended tasks to resume the execution.

The C64 chip provides an interesting hardware feature called the "wake-up bit". When a thread tries to wake up another thread, it sets the "wake-up bit" for that thread. This enables a thread to store a wake-up signal. Hence, if a thread tries to suspend itself after a wake-up signal is sent, it wakes up immediately and the suspend effectively becomes a no-op. This feature is fully utilized by phasers to easily move from phase to phase without worrying about a thread that can execute a suspend after a wake up signal.

3.4 Adaptive Phasers

Adaptability is one of the main features of our phaser implementation. As explained before, the runtime can directly detect the synchronization operation being performed and make a reasonable decision as to how to execute it. A phaser operation can switch to the optimized versions that utilize hardware primitives. These details of how a phaser operation is executed are hidden from the user.

Phaser operations can be implemented in a number of ways to take advantage of the particular characteristics of the underlying hardware. Even when a phaser has all tasks registered in SIGNAL_WAIT mode, it is not guaranteed that a hardware barrier will be used. A task that is registered to support split-phase or fuzzy barriers may signal ahead of its next operation. When a task registers as SIGNAL_ONLY or WAIT_ONLY on a phaser that has been using a hardware barrier, our runtime detects such a scenario and switches to software mode.

The runtime chooses the best mode of operation, depending on the current program state and available features. Each implementation alternately exhibits particular traits: maximum portability and reasonable performance is achieved with a *busy-wait* implementation; low bandwidth and low power usage are featured in the *suspend-awake* implementation.

3.5 Memory Optimizations

Phaser and *sync* objects contain volatile phase counters, and phaser operations involve frequent read and write of those counters in both software based busy-wait approach and hardware-optimized implementations. So low latency and high bandwidth of the memory system are key to the performance of phasers.

The C64's memory hierarchy, as seen in Figure 2, is similar to hardware cache in regular commodity CPUs. The power of using it comes from program manageability as our runtime itself can decide which synchronization objects need to reside on or move to the high-speed SRAM. Yet there is a tradeoff in this software-managed caching approach because the DRAM is limited in its sizes and shared with stack in C64. For a simple DRAM-optimization, the runtime allocates on SRAM, synchronization objects that contain spinning counters. More complex optimizations use heuristic or historical information to identify frequently-accessed data and move them to SRAM. Further memory management by the Habanero-C runtime, such as allocating a list of synchronization objects in a dense array, provide another level of memory optimizations on C64.

4 Implementation and Experiments

Habanero-C includes a workstealing runtime and a compiler for the async and finish task parallel programming constructs. The C64 manycore processor described in Section 3.1 was used as experimental platform for this study. This work is the result of a joint research effort between Rice University and University of Delaware (UDel). Figure 3 shows a description of the infrastructure used for this project as well as the contributions of each institution.

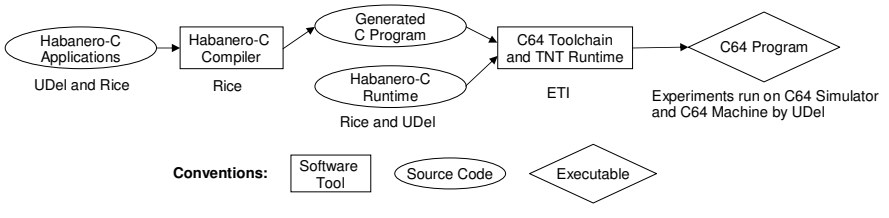


Fig. 3. Collaboration and Software Infrastructure

4.1 Implementation and Experimental Benchmarks

Habanero-C compiler was implemented on top of the ROSE source-to-source compiler framework [10]. The compiler transforms async and finish statements to appropriate library and runtime calls that create and enqueue tasks, and calls to ensure proper task termination within each finish scope.

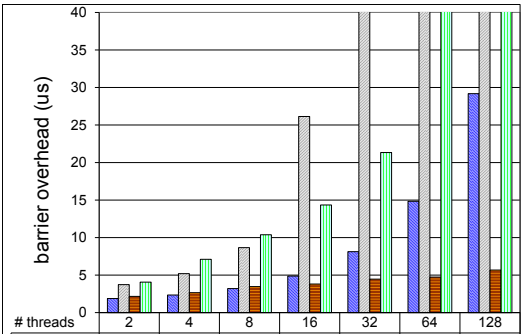
Habanero-C runtime contains a number of worker threads; each worker thread maintains a double-ended queue (deque). A worker enqueues and dequeues tasks

from the tail end of its deque when creating and executing local tasks, respectively. Other workers steal tasks from the head of the deque, when they do not have local tasks to work on. While this approach to the workstealing runtime is similar to the Cilk runtime [6], task creation and enqueuing policy when encountering an async is different from Cilk. In Cilk's "work-first" policy, the code after the async task body (the *continuation*) is pushed onto the deque while the current worker continues the execution of the async body. In our policy, which is referred to as "help first" [4], the async task itself is pushed onto the deque while the current worker continues the execution of the continuation.

The evaluation was conducted using microbenchmarks and common applications. The microbenchmarks include barrier and threadring for evaluating phaser barrier and point-to-point synchronizations. The applications include two-dimensional finite difference time domain (FDTD2D), and Successive Over Relaxation (SOR), to study the performance impact of synchronization overhead using software and hardware approaches, and their tradeoffs.

4.2 Hierarchical Phasers and Memory Optimizations

In Figure 4, we show the barrier overhead of using software flat phasers versus hierarchical phasers, and phasers residing on SRAM versus on DRAM. The dramatic scalability improvements of using hierarchical phasers (4-degree fan-out hierarchy) as compared to flat phasers are obvious. Placing phasers in SRAM results in large (one to two orders of magnitude) overhead reduction for both flat phaser and hierarchical phasers. While this performance does not imply superiority of SRAM over DRAM implementation in general (spin-based solutions may have adverse effects as well), we use the SRAM hierarchical phasers as baseline to compare with other hardware-based implementations in later sections.

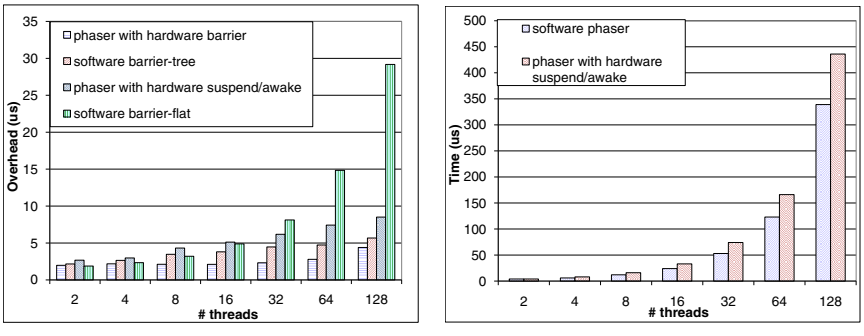


# threads	2	4	8	16	32	64	128
SRAM-flat	1.882	2.343	3.204	4.877	8.122	14.836	29.178
DRAM-flat	3.73	5.19	8.66	26.14	113.56	416.82	1681.59
SRAM-tree	2.164	2.651	3.469	3.808	4.465	4.734	5.678
DRAM-tree	4.06	7.1	10.37	14.35	21.33	41.2	89.85

Fig. 4. Hierarchical Phasers and SRAM Optimization

4.3 Barrier and Point-to-Point Microbenchmarks

The barrier microbenchmark was based on the EPCC OpenMP *syncbench* benchmark that was developed for evaluating OpenMP barrier overhead. When using phasers as barriers, barrier wait operations are performed by phaser next operations. A task can dynamically join and leave a barrier wait operation by registering and deregistering with the phaser that is created (with at least SIGNAL_WAIT capability) for this operation. This is different from OpenMP barrier that only allows a fixed number of threads involved in a barrier from the beginning to the end of a parallel region. OpenMP does not permit the use of barriers within parallel loops, either.



(a) Phaser Barrier (b) Threadring

Fig. 5. Barrier and Point-to-Point Microbenchmarks

Figure 5(a) shows the barrier overheads using four phaser implementations on C64. The implementation that leverages the C64 hardware barrier incurs much lower overhead than that of the software barrier. The reason behind this is the phaser implementation switches to hardware barriers whenever the tasks registering with the phaser are actually perform the barrier wait operations. The implementation that uses suspend/awake performs worse than software phasers because of the sequentially accumulated cost of hardware interrupt in suspend/awake implementation. For software hierarchical phasers, both signal gathering and wait operations are performed in parallel, thus reducing overhead.

The *threadring* microbenchmark evaluates point-to-point signal-wait operation of two tasks. In this program, a group of tasks form a signal ring; each task waits on the signal from the previous task and signals the next task after receiving the signal. As shown in Figure 5(b), the memory consumption of the software busy-wait approach has little impact on the time required to complete a round of the ring. In fact, the implementation using software phasers performs slightly better than the one using hardware interrupts. These imply the effectiveness of using the portable software-based solution for point-to-point synchronizations.

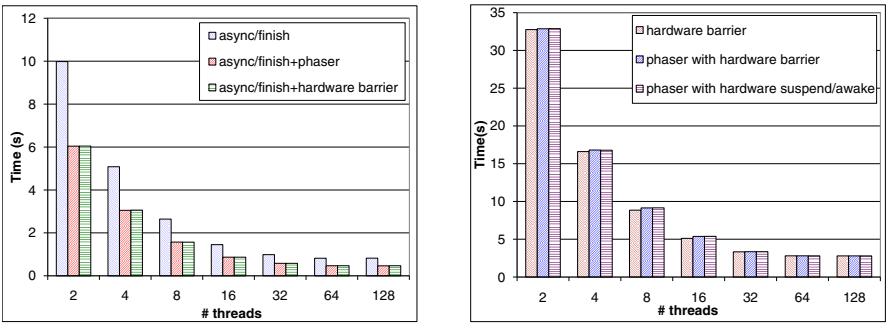
The high performance obtained using the *busy-wait* implementation is due in part to the high bandwidth and low latency of the local on-chip memory in C64.

Although the other techniques in our experiments use hardware support, they still suffer from overhead in the supporting software required to use the hardware primitives. In contrast, *busy-wait* uses a very simple polling mechanism that does not require complex software support.

4.4 Applications

A simulation of propagation of electromagnetic waves that uses the two-dimensional finite difference time domain (FDTD2D) algorithm was used to test the effectiveness of phasers for commonly used scientific applications. The FDTD algorithm used [11] is an excellent choice to study synchronization and parallelization techniques for manycore architectures; the algorithm has abundant parallelism and its complexity depends on the physical phenomena that it models, ranging from a simple read-modify-write of an array to numerical integration of physical variables. The experiments simulate the propagation of a wave in two dimensions, with an implementation that results in a two dimensional array where each element is updated several times using data from the array elements that surround it. A full description of the FDTD algorithm used here can be found in [12].

The case presented in Figure 6(a) is characterized by a constant amount of computation per array element. Barriers have been successfully used to synchronize multiple threads executing the program, since all threads share approximately the same amount of workload.



(a) FDTD 2D (b) SOR

Fig. 6. Applications Performance Using Different Implementations

Figure 6(a) shows FDTD2D performance using following implementations:

1. async/finish: use finish to join tasks as barrier operations; tasks are recreated via async and joined in each iteration. This approach is commonly used in task parallel programming language, such as Cilk.
2. async/finish+phaser: use phaser to perform barrier *wait*; tasks are created once, and then coordinated via phasers in each iteration. Tasks are terminated when the computation completes.

3. async/finish+hardware barrier: similar to async/finish+phaser, but using hardware barrier to perform barrier *wait*.

The implementation using phasers doubles the performance of the one using finish for synchronization. The reason behind this is that finish is a coarse-grained synchronization approach, and it suffers from the runtime overhead for creating and scheduling tasks. Thus algorithms that require fine grained synchronization with large number of iterations should use lower-overhead, finer-grained task coordination mechanism such as phasers. The similar performance between the one using phasers and the one using hardware barriers is because phasers adaptively switch to hardware barrier when it detects a barrier *wait* should be performed.

Another application we used for the evaluation is Red-Black Successive Over-Relaxation (SOR). SOR is a method of solving partial differential equations using a variant of Gauss Seidel method. Task synchronization patterns are similar to FDTD2D, requiring barrier operations to synchronize each iteration. Figure 6(b) shows similar executions time for phasers and hardware barriers, demonstrating the adaptivity of our phaser implementation to the underlying hardware.

5 Related Work

Cilk [6], Cilk++, and OpenMP 3.0 introduced task parallelism at the programming language level. The Cilk's sync and OpenMP's taskwait constructs, related to finish in Habanero-C, are global barrier synchronization points indicating that the execution of current task cannot proceed until all previously spawned tasks have completed. Using this style of synchronization, the runtime efficiency depends heavily on the granularity of parallelism built into the program.

X10 [1] and Chapel [2] provide constructs for dynamic task creation and constructs for task synchronization. X10 allows for the barrier-style phase advancing among all participating tasks using the next operation but it lacks of the point-to-point signal-wait style coordination capability that is available in phasers. Chapel introduce sync variables for programming producer-consumer coordination among tasks. Chapel does not provide direct language construct for barrier operations, or phase-ordered synchronization.

The JUC *CyclicBarrier* class [13] supports periodic barrier synchronization among a set of threads. Unlike phasers, however, *CyclicBarrier* does not support the dynamic addition or removal of threads; nor do they support one-way synchronization or split-phase operations.

6 Conclusions and Future Work

In this paper, we present the design and implementation of *phasers*, a high-level synchronization construct for asynchronous tasks on manycore Cyclops64 processors in the Habanero-C workstealing runtime. We have designed and implemented different techniques for phaser synchronization on C64 that use a

combination of software-based busy-wait approach, hardware barriers, and hardware support for thread suspend/awake. Our experiments show that phasers are able to take advantage of hardware primitives on manycore architectures and optimizations for their memory subsystems to provide superior performance to portable software approaches.

In the future, we will experiment with more bandwidth-limited applications on C64 to evaluate the limitations of our busy-wait phaser implementation. We will also investigate more applications for other phasers operations, such as broadcast and reduction.

Acknowledgments. We wish to thank Vincent Cavé and Joshua Landwehr for their hard work on the correctness, performance and efficiency of the Habanero-C runtime. We wish to express our gratitude to ET International for their advice and their logistics support which ultimately boosted the quality and quantity of our experiments. This work was supported by the National Science Foundation through grants CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534.

References

1. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on OOPSLA, pp. 519–538. ACM, New York (2005)
2. Chapel Programming Language, `http://chapel.cray.com/`
3. 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 ICS, New York, NY, USA, pp. 277–288 (2008)
4. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In: IPDPS 2009 (2009)
5. Cuvillo, J.d., Zhu, W., Hu, Z., Gao, G.R.: TiNy Threads: A Thread Virtual Machine for the Cyclops64 Cellular Architecture. In: IPDPS 2005, 265.2 (2005)
6. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN Conference on PLDI. Ser. PLDI 1998, pp. 212–223. ACM Press, New York (1998)
7. Sarkar, V.: Synchronization using counting semaphores. In: Proceedings of the 2nd International Conference on Supercomputing, pp. 627–637. ACM, New York (1988)
8. Shirako, J., Sarkar, V.: Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism. In: IPDPS 2010 (2010)
9. Wentzlaff, D., et al.: On-chip interconnection architecture of the tile processor. IEEE Micro. 27(5), 15–31 (2007)
10. ROSE compiler framework, `http://www.rosecompiler.org`
11. Taflove, A., Hagness, S.: Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd edn. Artech House Publishers, Boston (2005)
12. Orozco, D., Gao, G.: Diamond tiling: A tiling framework for time-iterated scientific applications. In: CAPSL Technical Memo 091 (December 2009)
13. Goetz, B.: Java Concurrency In Practice. Addison-Wesley, Reading (2007)