Synchronization for Dynamic Task Parallelism on Manycore Architectures

Yonghong Yan, Sanjay Chatterjee, Daniel Orozco, Elkin Garcia, Jun Shirako, Zoran Budimlic, Vivek Sarkar and Guang Gao

CAPSL Technical Memo 094
February 1st, 2010

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 reality requires a productive programming interface for parallel programming, and an efficient execution and thread coordination runtime. Dynamic task parallelism, introduced recently in several programming languages, promises to be an effective approach to parallel programming. Unlike data parallel and SPMD programming models, concurrent tasks can be dynamically created and joined at any time along the execution. A critical prerequisite for an efficient task parallel runtime is a scalable synchronization mechanism to support task coordination in different level of granularity.

This paper presents a study of task parallel runtime and synchronization issues in manycore architectures, and provides alternative approaches to low-level hardware synchronization primitives. To address these issues, we have implemented a high-level synchronization construct called *phasers* on a Cyclops64 manycore processor. Phasers support both localized and group synchronization of tasks by allowing threads to register and deregister from groups of synchronizing tasks. Phaser interfaces are much more flexible than the architecture-specific hardware primitives available on manycore processors. We have experimented with several implementations of phasers using software support, hardware support or both to explore their portability, usability and performance. Our results show that phasers provide comparable performance to hardware synchronization primitives, and much better performance than other traditional constructs such as OpenMP barrier for a set of commonly used benchmark applications.

1 Introduction

Manycore architectures – hundreds to thousands of cores per processor – is seen by many as a natural evolution of multicore processors. Manycore system workloads could include applications that are massively parallel, require system partitioning, or have dynamically changing parallelism. In reality, without comprehensive enabling software, it is very hard to achieve maximum computation throughput on such systems. 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 the prevalent multicore systems. Unlike the data parallel and the SPMD programming models that normally assume a fixed number of concurrent threads during program execution, in dynamic task parallelism, concurrent tasks can be created and joined at any time during the execution.

The overhead of communication and synchronization between the concurrent tasks typically presents one of the greatest obstacles to getting high performance and scalability on parallel systems. In addition, the code that deals with thread communication and synchronization is also a major source of complexity in parallel programming. To support the diverse workloads in manycore architectures, synchronization mechanisms that provide high-level operations, such as barrier, point-to-point signal/wait, and wavefront synchronization, with 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 one interface, and feature deadlock-freedom and phase-ordering. In this paper, we present our evaluation of phasers and a task parallelism runtime library on the Cyclops64 manycore machine [4], a revolutionary architecture developed by IBM that targets the petaflop supercomputing market. We implemented phasers using different combinations of a portable busy-wait mechanism and the hardware synchronization primitives provided by Cyclops64. The micro-benchmark experimental results show that all our phaser implementations outperforms OpenMP barriers by a large margin (up to two orders of magnitude). For benchmark applications, our phaser implementations deliver performance that is comparable to the hardware barriers provided by the Cyclops64 architecture, while providing a much more flexible and higher-level synchronization mechanism than either OpenMP or Cyclops64 hardware barriers. These experiments validate that phasers can be a highly efficient, flexible and productive programming construct for synchronization on manycore processors.

The main contributions of this paper are:

- An analysis of synchronization issues in manycore architectures (using Cyclops64 processor as an example) and how the phaser synchronization mechanism for task-parallel programming model addresses these issues.
- A study and evaluation of busy-wait phaser implementation for the Cyclops64 manycore architecture. Since the applications we have tested do not stress the (very high) memory bandwidth of the Cyclops64 architecture, the busy-wait phaser implementation performs on par with the variants using non-portable lower-level Cyclops64 hardware mechanisms.
- The evaluation of three other phaser implementations that leverage hardware and system primitives (such as hardware barriers and suspend-awake) for both barrier and point-to-point synchronization. We expect these variants to outperform the busy-wait implementation in bandwidth-intensive applications.

The rest of the paper is organized as follows: Section 2 introduces the Cyclops64 hardware architecture and its threading and synchronization model. Section 3 presents the task parallelism programming model, as well as the phaser construct and its features. We use an example to demonstrate how to create task parallel programs using phasers. Section 4 describes our implementations of phasers and the task parallelism runtime on Cyclops64. Section 5 presents our experimental results. Finally, Section 6 discusses related work and Section 7 contains our conclusions.

2 Cyclops64 Manycore Processors

Cyclops64 is a massively parallel architecture initially developed by IBM as part of the Blue Gene effort. It targets the petaflop supercomputing market with a peak performance in excess of

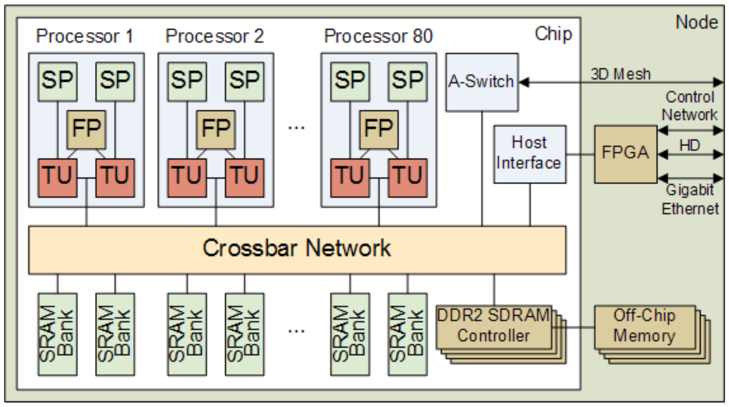

Figure 1: Cyclops64 Processor Architecture

1 PFLOPS. As shown in Figure 1, a Cyclops64 processor features 80 processing cores on a chip, with two thread units per core that share one 64-bit floating point unit. Each core can issue 1 double precision floating point Multiply Add instruction per cycle, for a total performance of 80 GFLOPS per chip when running at 500MHz. Each thread unit has a user-manageable on-chip memory of 32KB that can be used as cache. The processor chip includes a high-bandwidth on-chip crossbar network with a total bandwidth of 384 GBytes/s, and four memory banks for a total off chip memory bandwidth of 16GBytes/s.

Cyclops64 departs from conventional processor architecture in its thread model. Threads in Cyclops64 run in a non-preemptive mode. The non preemptive thread model was designed to provide several performance improvements, such as low thread management overhead, easier and simpler synchronization, and easier access to the physical memory while greatly overcoming the traditional limitations of the non preemptive systems. Creation and termination of threads is simplified because all hardware resources are directly exposed to the application without the virtualization complexity. The low complexity of the Cyclops64 threading architecture allows thread creation and termination in only dozens of cycles.

Cyclops64 chips contain a special signal bus that allows threads to perform very efficient synchronization without any memory bus interference. The signal bus connecting all threads on a chip can be used to broadcast synchronization operations in less than 10 clock cycles, enabling efficient barrier operations and mutual exclusion synchronization. Fast point-to-point signal and wait operations are directly supported by hardware interrupts. For those operations, synchronization between threads can be achieved in tens of cycles.

Cyclops64 toolchain includes a highly efficient threading library, named TNT (or TiNy-Threads) [5], which uses the Cyclops64 hardware support to implement threading primitives. The TNT thread interface is similar to the standard *pthread* API, simplifying porting of pthread-based runtime systems and applications to Cyclops64. Additionally, TNT provides APIs that can be used to access the hardware synchronization primitives, as shown in the Table 1.

Name	Description
tnt_suspend()	Suspends execution of the current thread indefinitely
tnt_awake (const tnt_desc_t)	Awakens a specified thread that was previously suspended
tnt_barrier_include (tnt_barrier_t *)	Add the calling thread to the list of threads that wait on this barrier
tnt_barrier_exclude (tnt_barrier_t *)	Remove the calling thread from the list of threads that wait on this barrier
tnt_barrier_wait (tnt_barrier_t *)	Waits until all threads have reached the call to a specified barrier function

Table 1: Some TNT APIs for synchronization on Cyclops64

3 Task Parallelism and Synchronization using Phasers

While the Cyclops64 instruction set architecture (ISA) primitives and toolchain interfaces described in the previous section allow for very efficient low-level thread creation and synchronization, actually writing parallel applications that correctly use such low-level primitives is a very tedious and error-prone process. A higher-level programming model that would simplify task creation and synchronization will greatly improve the programmability of manycore architectures such as Cyclops64. In this paper, we propose the *dynamic task parallel programming model* as a vehicle for exploiting parallelism and *phasers* as a mechanism for task synchronizations.

3.1 Dynamic task parallelism

Task parallelism, as compared to data parallelism, refers to the explicit creation of multiple threads of control, or tasks, which synchronize and communicate under control of programmers. Conventionally, task parallelism is enabled to programmers through library APIs, notably *pthreads*. Recently, task parallelism has gained more attention in the multicore and manycore systems, and was introduced in several parallel programming languages because of its flexibility and productivity. The three programming languages developed as part of the DARPA HPCS program (Chapel [2], Fortress [6], X10 [1]) identified task parallelism as one of the prerequisites for success. Task parallelism is also being introduced in existing programming models for shared-memory parallelism such as OpenMP 3.0.

We are using two basic primitives borrowed from X10 for the task parallel programming model: async and finish. The async statement, *async* $\langle stmt \rangle$, causes the parent task to fork a new child task that executes $\langle stmt \rangle$. Execution of the async statement returns immediately, i.e., the parent task can proceed to its following statement without waiting for the child task to complete. The finish statement, *finish* $\langle stmt \rangle$, performs a join operation that causes the parent task to execute $\langle stmt \rangle$ and then wait until all the tasks created within $\langle stmt \rangle$ have terminated (including transitively spawned tasks).

The finish statements represents a group synchronization scope, often referred to as a finish scope. Upon entering a finish scope, the master task initializes the scope context to keep track of tasks and other synchronization objects used within the scope. When the master task

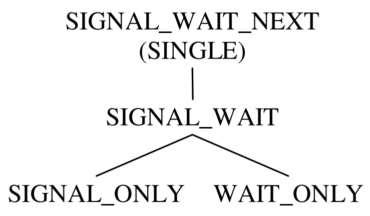

Figure 2: Phaser mode lattice

reaches the end of the scope, it performs a "join" operations on all those tasks spawned within the scope. When the join operation completes, the master task releases those synchronization objects created inside of the scope and enters into the parent scope.

In this execution model, efficiency of the runtime system and task synchronization mechanisms are extremely important for application performance and scalability. Design and implementation of these mechanisms present several challenges, for example, minimizing the overhead and storage cost from task creation/termination, join operation, scheduling and synchronization. In our implementation, presented in Section 4, the runtime system leverages Cyclops64 massive parallelism, fast thread creation and termination, and efficient hardware barrier primitives to meet these challenges. Task synchronization mechanisms use the light-weight thread suspend/awake library calls which are implemented using hardware interrupts.

3.2 Phasers

Phasers [3] are programming constructs that unify collective and point-to-point synchronizations in task parallel programming. It was invented as part of the Habanero-Java programming language [7], which is derived from X10 v1.5. A variant of phasers will be included in the concurrency packages of the upcoming Java Development Kit (JDK) 7. The use of phasers guarantees two safety properties: deadlock-freedom and phase-ordering. 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 [8], and X10 clocks [1]. Phasers are designed to be easy to use and safe at the same time, helping programmers productivity in task parallel programming and debugging.

In a task parallel program, a task registers itself on a phaser in one of the four modes: SIGNAL_ONLY, WAIT_ONLY, SIGNAL_WAIT and SINGLE. For example, in a barrier synchronization scenario, a task would register itself in SIGNAL_WAIT mode. In a producer-consumer scenario, the producer can register in a SIGNAL_ONLY mode while the consumer can register in a WAIT_ONLY mode. When executing a *next* call, the task participates in a barrier or point-to-point operation depending on the registration mode on a phaser. A task can register on multiple phasers, so it can participate in a barrier synchronization on one phaser and a point-to-point synchronization on another phaser while executing a single *next*. In order to guarantee deadlock freedom, a child task can only register in a mode that is at the same level or below the mode of the parent task according to the phaser mode lattice shown on Figure 2.

Name	**Description**
phaser * newPhaser(rMode mode)	Create a new phaser and register it with the current task with *mode*
void next()	Advance each phaser on which current task is registered to its next phase. The semantics depends on the registration mode a task registers with a specific phaser.
void signalAll()	Signal each phaser on which current task registers with a SIGNAL capability.
void doSignal(phaser* ph)	Signal a specific phaser *ph*.
void startFinish()	Enter into a new finish scope
void stopFinish()	Exits from the current finish scope
void async(void (*func)(void * args), void * args)	Spawn a new task that executes function *func* with argument *args*
void asyncPhasedImplicit(void (*func)(void * args), void * args)	Spawn a new task executes function *func* with argument *args*; the new task registers on all the phasers of the parent task.
void asyncPhasedExplicit(phaser * phasers[], rMode modes[], int size, void (*func)(void * args), void * args)	Spawn a new task executes function *func* with argument *args*; the new task registers on those *phasers* with *modes* specified explicitly.

Table 2: Programming APIs for phasers and task parallelism runtime

The other restriction of phasers for providing deadlock freedom is that a task can transmit to its child task only those phasers that have been created in the same immediate enclosing finish scope. When a task exits from a finish scope, it automatically deregisters itself from all the phasers that were created in that scope. So, a child waiting on a barrier can proceed when the parent deregisters. However, if the child was registered on a phaser not from the immediately enclosing finish scope, then there is a potential deadlock situation as the parent would not have deregistered from the phaser and could be waiting on another *next* statement.

3.3 Library APIs for phasers and runtime

Table 2 includes the library APIs of our runtime and phaser implementations. We have decided to use a library approach instead of a language-based approach in order to exploit the flexibility of experimenting with different hardware primitives and system libraries. With our ongoing compiler efforts for task parallelism, a better optimized set of APIs that are tuned for compiler transformation will be developed.

3.4 Parallel breadth-first traversal example

To demonstrate how to program using dynamic task parallelism and phasers, we include a parallel breadth-first traversal (PBFT) algorithm for graph, with pseudo-codes shown in Figure 3. Given a graph $G = (V, E)$, and a root vertex, r, the BFT algorithm explores the edges of G to discover all the vertices reachable from r level by level. During the exploration of each level, the algorithms scans a queue (Q variable as in Figure 3) that contains vertices explored in the previous level. It adds to another queue ($Qnext$) the neighbors of those vertices in Q that were

```
1   /* parallel breadth-first traversal */
2   void pbft ( G g, V root ) {
3        Q[0] = root;
4        marked[root] = true;
5        startFinish( );
6        newPhaser( SIGNAL_WAIT );
7        pbft_part ( Q );
8        stopFinish( );
9   }
10
11  /* partition data and spawn task */
12  void pbft_part ( V[ ] Q ) {
13       next(); /* collect results */
14       V [ ][ ] Qpart = part(Q, BLOCK_SIZE);
15
16       for ( i=0; i<Qpart.size-1; i++ )
17            asyncPhasedImplicit(pbft_worker,
18                 Qpart[i] );
29       pbft_worker ( Qpart[Qpart.size – 1] );
20  }
```

```
21  /* the task function */
22  void pbft_worker ( V[ ] Q) {
23       V [ ] Qnext = BFT( Q );
24       next( );
25       if ( Qnext.size( ) )
26            pbft_part ( Qnext );
27  }
28
29  /* sequential one-level traversal */
30  V [ ] bft ( V[ ] Q ) {
31       V [ ] Qnext;
32       foreach( v : Q )  {
33       /* get the neighbors of v */
34       nbList = v.nbList;
35       foreach ( nb : nbList )
36            if ( cas(!marked[nb], 0, 1) )
37                 Qnext.add( nb );
38       }
39      return Qnext;
40  }
```

Figure 3: Parallel breadth-first traversal algorithm using task parallelism and phaser synchronization

not visited before. At the end of the exploration of a level, the content of *Qnext* is copied to *Q* and *Qnext* is emptied. Then the algorithm proceeds to explore the next level, until there are no more neighbors to visit, i.e., *Q* is empty.

The task parallel version of this algorithm, when exploring each level, partitions the content of *Q* into multiple blocks. For each block, a task is then created to process the block using the *asyncPhasedImplicit* API, as shown in line 17 of the *pbft_part* function in Figure 3. Each task executes the *pbft_worker* function that explores the given level by calling a sequential BFT function *bft*, as in line 23. The *bft* function is similar to the sequential BFT algorithm, except in that it needs to perform an atomic "compare-and-set" operation to check whether a vertex was already visited or not. The barrier synchronization between levels of multiple parallel tasks is performed in a *next* call, as in line 24 of the code. The phaser and finish scope of the whole algorithms are initialized at the beginning of the *pbft* function in line 2.

As shown in Figure 3, the PBFT program includes no explicit thread creation and termination calls or explicit barrier operations, thus reducing the complexity of parallel programming reasoning for programmers. With the deadlock-freedom guarantee of using phasers for synchronization, application programmers are able to write safe parallel programs in a much more productive way than using system and hardware APIs. In Section 5, we discuss the performance factors of using these APIs in different applications.

4 Implementation

This section discusses how our implementation of task parallel runtime and phasers leverages the hardware and system features of Cyclops64.

4.1 Task Parallelism

Our task parallelism runtime implementation takes advantage of the fast thread creation and termination on Cyclops64. Unlike the runtime in Cilk [9] or Habanero-Java [3] that use work-stealing for task scheduling, our runtime uses a much simpler approach that maps task creation to creation of a new Cyclops64 thread that will run on an available thread unit. Our *stopFinish* operation is also greatly simplified in that the master task only performs a join operation on children tasks, without performing other bookkeeping or creating continuations. A significant drawback of this approach is that the number of tasks that can exist at any point in program execution cannot exceed the hardware capabilities of the system. While this would be a very serious limitation on the current multicore architectures, a massively parallel manycore system such as Cyclops64 has support for a large number of hardware threads (160 threads on a single chip), allowing many useful task parallel applications to be executed on the system without exceeding this limit.

4.2 Phasers

We have used a library approach for the phaser implementation. Each phaser has the knowledge of all signaling tasks. Each task has the knowledge of all phasers it is registered on, in both SIGNAL and WAIT mode. Each phaser has two counters that track the current signal phase and wait phase, named *masterWaitPhase* and *masterSigPhase*. A phaser registration with a task is represented by a synchronization object, named *sync* that contains the registration mode and the current phase. Each phaser has two tables, a signal *sync* object table and a wait *sync* object table. A task uses two hash tables to maintain two lists of <phaser, sync> pairs, one for signal *sync* objects and another for wait *sync* objects.

We have four implementation variants of the phasers. They all have the same programming interface, requiring no changes in application code when switching between different phaser implementations. They can be summarized as follows:

- Busy-wait: In this implementation, the *signal* and *wait* operations update corresponding integer counters and then spin in a loop waiting for the synchronization condition to be met. This implementation consumes memory and CPU resources, but it is much portable and simpler to implement since it relies only on a single platform-dependent "compare-and-set" operation.

- Suspend-awake: A task suspends when it is waiting for a signal, so it does not consume memory bandwidth as in a busy-wait implementation. A signal operation wakes up waiting (sleeping) tasks. This implementation uses the *tnt_suspend* and *tnt_awake* functions provided by the Cyclops64 TNT libraries, which use fast hardware interrupts.

- Busy-wait + tnt_barrier: Using this implementation, if a task performs a barrier operation when calling *next*, the runtime calls the hardware-supported *tnt_barrier_wait*. If a task performs a point-to-point signal or wait operation, the runtime uses the busy-wait approach. The runtime decides whether an operation is a barrier or a signal-wait according to the registration mode of the phasers involved. In addition, when a task registers(deregisters) with a phaser, a *tnt_barrier_include(exclude)* call is made to set up the synchronization group.

- Suspend-awake + tnt_barrier: This implementation is similar to the last one, except that point-to-point signal and wait operations use the *tnt_suspend* and *tnt_awake* functions.

5 Experimental Results

In this section, we discuss the experimental results of phaser synchronization using different benchmarks. The Cyclops64 manycore processor (Section 2) was used for this study. The evaluation of phasers was conducted using both micro-benchmarks and two common applications: A two-dimensional finite difference time domain (FDTD2D) and LU domain-decomposition (LU). Phaser synchronization and program partitioning code was inserted manually by the programmer in each case. The programs used for the experiments are linked with an auxiliary library that we call the "Phaser runtime system". That was written by the authors and that implements phaser primitives using thread operations normally available to the user.

Portability, efficiency and ease of implementation of phasers were studied using the above four different approaches. The programs were compiled with gcc for Cyclops64, version 4.3.2, with the *-O3* flag for optimization. The programs were executed using the FAST Simulator [4], which is functionally accurate and it reproduces execution timing with exceptional accuracy. The following sections present a detailed description of each experiment.

5.1 Threadring micro-benchmark for point-to-point synchronization

The *threadring* example is a simple benchmark to evaluate the point-to-point synchronization overhead. In this micro-benchmark, a group of tasks form a signal ring; each task waits on the signal from the previous task and signals the next task after the task receives the signal. In *threadring*, each task registers one phaser with WAIT_ONLY mode and another phaser with SIGNAL_ONLY mode. As shown in Figure 4, the memory consumption of busy-wait has no impact on the time required to complete a round of signals. In fact, the busy-wait implementation performs slightly better than that using hardware interrupts. These results imply that busy-wait is not always the worst option in 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. Although the other techniques in our experiments use hardware support, they still suffer from overhead from the supporting software required to use the hardware primitives. In contrast, *busy-wait* uses a very simple polling mechanism that does not require complex support software.

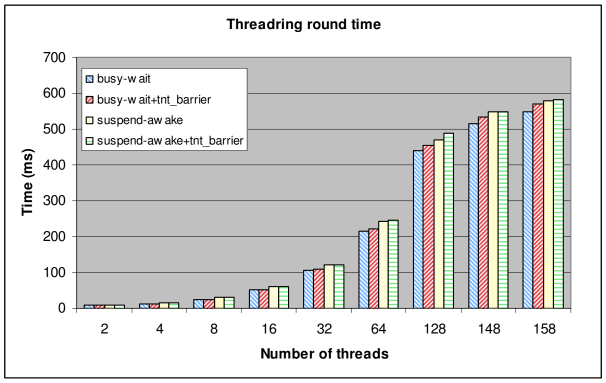

Figure 4: Threadring round time on Cyclops64 simulator

5.2 Barrier micro-benchmarks

We conducted two barrier operation evaluations: static barrier and dynamic barrier. In the static barrier example, the number of tasks participating in the barrier operations are fixed. We use this benchmark to compare barrier operations using phasers with that in other programming model, such as OpenMP that only allows a fixed number of tasks involved in barrier wait operations. The results, as plotted in Figure 5, shows OpenMP barrier (using GNU libgomp runtime) overhead is about 2x to 3x larger than that phaser barrier overhead. This is especially true when the number of tasks involved in barrier operations is large, for example, for more than 32 tasks.

The dynamic barrier benchmark highlights the capability of phasers to handle a variable number of tasks. This capability is key to handling synchronization for dynamic parallelism in a program. In this benchmark, a phaser is used as a barrier synchronization point where the number of threads participating in a barrier varies over time. The phaser is initiated with 2 threads participating in a barrier. After the first barrier, each thread makes a decision over 3 choices. It can either spawn a new thread or terminate itself or continue as is. When a thread spawns, the new thread automatically gets registered on the same phaser to participate in the barrier during the next synchronization point. When a thread decides to terminate itself, then it deregisters from the phaser. The decision is based on a random probability. This way a dynamic number of threads participate in the same barrier in over time.

The results for dynamic barrier evaluation are shown in Figure 6. Interestingly, for 128 threads, the busy-wait phasers outperforms the other three implementations for a large margin, which may imply the setup overhead of TNT barriers and suspend-awake mechanisms become a dominant factor when a large number of tasks are participating in the operations.

5.3 Two-dimensional finite difference time domain (FDTD2D)

The Finite Difference Time Domain (FDTD) technique is a commonly used way to directly solve Maxwell's Equations for electromagnetic wave propagation. The FDTD technique uses

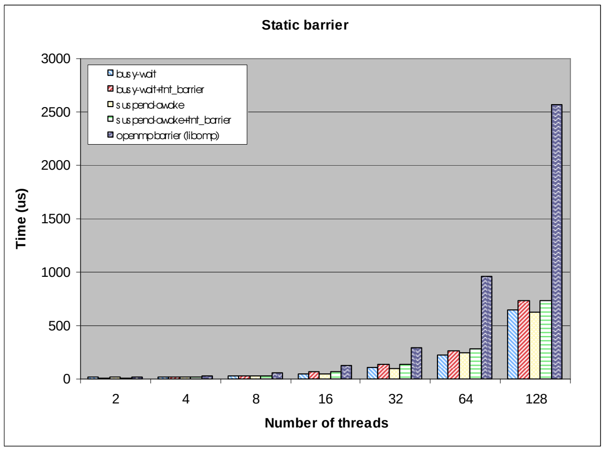

Figure 5: Static barrier overhead

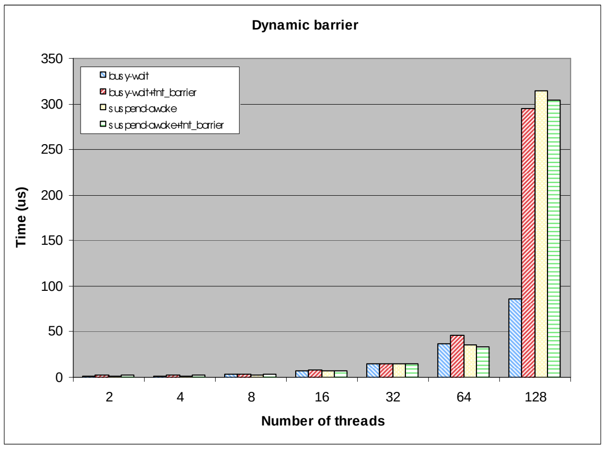

Figure 6: Dynamic barrier overhead

discretization of physical variables to enable approximations of derivatives. An implementation of the Finite Difference Time Domain (FDTD) technique as described in [10] was used to test the effectiveness of phasers for real scientific applications. FDTD is an excellent choice to study synchronization and parallelization techniques for many core architectures: The algorithm is simple, it has abundant parallelism and its complexity depends on the physical phenomena that it models, ranging from a simple array read-modify-write, to numerical integration of physical variables. In two dimensional FDTD implementations, the physical variables are discretized resulting in 2 dimensional arrays that are updated several times.

Two cases were considered for this experiment: A simulation of the propagation of an electromagnetic wave in vacuum (Figure 7) and a simulation of a non-linear material excited by a moving external source (Figure 8). For all cases, the physical environment simulated was discretized to 16 time steps and a grid of size $150x100$ for the spatial dimensions.

The simpler case presented in Figure 7 is characterized by a constant amount of computation

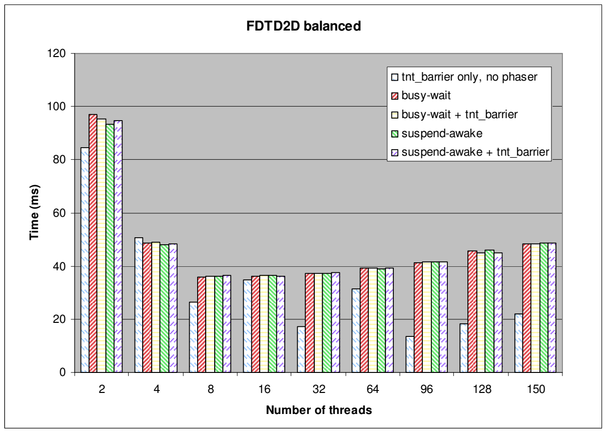

Figure 7: FDTD 2D with point source

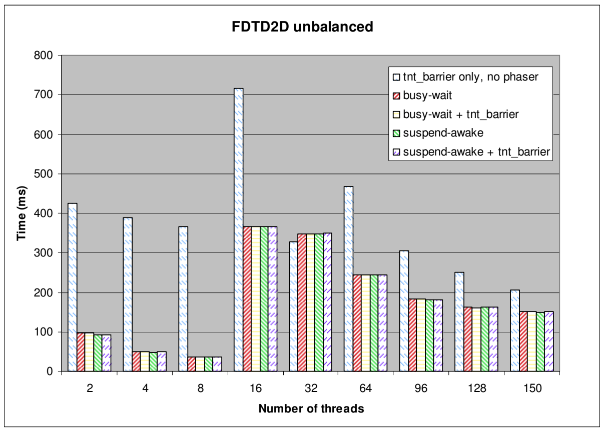

Figure 8: FDTD 2D with nonlinear physical source

per array element. Barrier synchronization has been successfully used to synchronize multiple threads executing the program, since all threads share approximately the same amount of workload. As observed in Figure 7, barriers are a reasonable choice to synchronize threads during each timestep. In the experiments shown in Figure 7, four different phaser implementations were used to synchronize only the threads that required synchronization. The overhead of phasers account for the somewhat lower performance experienced, but they present a reasonable approach to synchronization, even when compared against hardware-supported barriers running a program with uniform workloads.

Phaser synchronization overwhelms the performance of barrier synchronization when the computational load is not so uniform. The physical simulation shown in Figure 8 (source traveling in a nonlinear material) shows that barrier synchronization is inadequate when the load among threads is not uniform. In this case, the point-to-point synchronization provided by phasers presents a significant advantage over barriers, even when such barriers are natively supported by hardware. Figure 8 shows that the overhead of phaser implementations is negli-

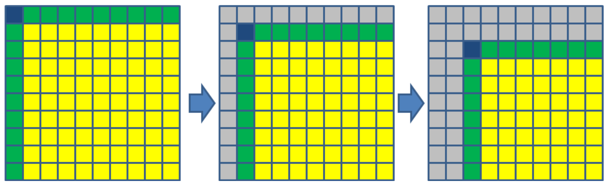
Figure 9: Progress of Blocked LU decomposition per iteration

gible in light of the advantages of point-to-point synchronization for real applications. Phaser synchronization greatly reduces problems associated with barriers such as noise amplification or bad scheduling of resources.

5.4 LU decomposition

LU decomposition is a well studied algorithm for uniprocessor and multiprocessor systems. It is frequently used to characterize the performance of high-end parallel systems and determine their rank in the Top 500 list [11]. We focus on blocking algorithms since they have been extensively used in distributed-memory and shared memory-systems [12, 13]. These kind of algorithms divide application's data into fixed, smaller-sized blocks where each one is of them is processed by one processing unit.

A more detailed explanation of the Blocked LU decomposition is as follows and uses figure 9: A matrix of size $N \times N$ is divided into $M \times M$ blocks, of approximately the same size. M iterations, each one with 3 steps, are done in a complete Blocked LU: (1) The top-left corner block (blue) is updated, then (2) the first row and column blocks (green) are calculated; each block can be processed independently and they use the results of the first step, and finally, (3) the rest of blocks (yellow) are calculated using the results from the second step. In a next iteration, blocks processed in the third step of the previous iteration become the target of the calculation. During the final iteration, only one block located in the lower right corner is calculated.

There are two main challenges in this algorithm: (1) The number of blocks processed are decreasing in each iteration, meaning that many processing units used in the first iteration will not be used in the next one, wasting computational resources. A dynamic partition through iteration can deal with this issue [14]. And (2), synchronization between steps of the same iteration use barriers to avoid data races. The use of barrier synchronization hurts the performance of the algorithm and it is not strictly necessary. Phasers, as a point-to-point synchronization mechanism, are better equipped to address this kind of algorithms.

A dependency graph for one iteration of LU decomposition is shown in Figure 10. Each arrow represents a data dependency being controlled by a phaser that orchestrates synchronization between each pair of blocks.

The results of executing LU are shown in Figure 11. Phasers present a better performance when compared to hardware supported barriers. The distinction is particularly significant when

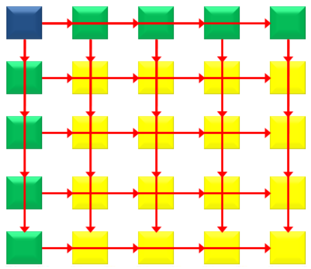

Figure 10: Iteration of LU decomposition synchronized by phasers

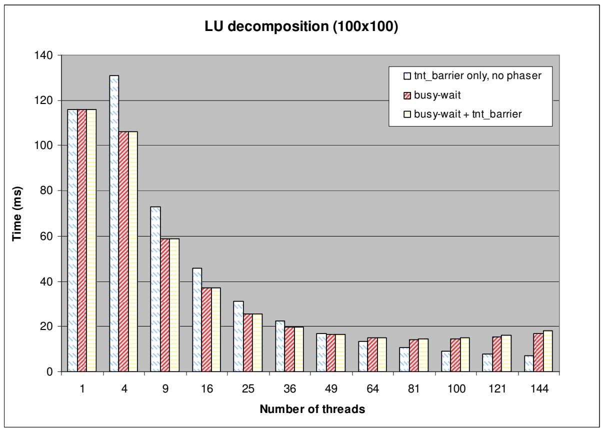

Figure 11: Static Blocked LU Decomposition

a low number of threads (less than 49 in this example) is used. For large number of threads, the granularity of data partition and phaser synchronizations becomes finger as compared to smaller number of threads, phaser overhead shows up in the overall performance. As we observed in the figure, when this number is greater than 64, hardware barrier outperforms phaser-based implementations. We expect that future implementations of phasers will provide competitive performance when compared with hardware barriers even for very fine grain synchronization.

Nevertheless, phasers are a very promising technique that perform reasonably well for uniform scenarios and that provide a significant improvement over barriers when tasks do not have the same amount of work, as is the case of a few threads executing LU or the FDTD 2D case presented in Figure 8. We expect that future, optimized implementations of phasers for a large number of threads will result in performance similar to that of hardware supported barriers.

6 Related Work

Task parallelism and fine grain synchronization have been two separate and active research topics in the past, and have been gaining their popularity as parallel programming for multicore and manycore architectures is gradually becoming programming mainstreams. For examples, approaches such as Cilk [15] and OpenMP [16] have been proposed to express and use available parallelism as tasks in programs. EARTH [17, 18, 19], I-Structures [20] and M-Structures [21] have proposed different solutions to address fine grain synchronization for parallel programming.

Cilk [15, 9] presents a simpler programming model where only fork/join operations are allowed, thus allowing some degree of task parallelism but limited to a join-type of synchronization between tasks. The use of work-stealing runtime in Cilk is one direction we would like to explore for our task parallel runtime.

OpenMP programming and execution model [16] address task parallelism in its recent release, but it is constrained in its ability to handle irregular task synchronization, and dynamic synchronization in general. OpenMP pays great attention to applications where barrier synchronization and single-thread execution regions are useful. These directives must be executed by a fixed number of threads, symmetrically, with no support for dynamic parallelism.

In EARTH (Efficient Architecture for Running Threads) runtime, tasks are associated with threaded function/procedure invocations that can be invoked dynamically through specialized procedure/function calls. Such tasks communicate through a shared memory model, while the code body of an individual task can also be parallelized and managed using a multhreading execution model having its origin in dataflow models [18, 22, 23].

I-Structures [20] are single-assignment constructs that support synchronization by allowing a single producer per memory location. In systems supporting I-Structures, readers are forced to wait (often using hardware support) for the producer to write during memory operations. M-Structures [21] allow multiple assignments, but each value has a single producer. Neither I-structures nor M-structures are general enough to support all the synchronization patterns that can be supported by phasers.

The JUC *CyclicBarrier* class [24] 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.

Titanium is a dialect of Java for SPMD parallelism [25]. The language has a notion of single values that are used to ensure coherence at synchronization points. A set of expression rules enable coherence by inserting conservative checks statically. Phasers, on the other hand, do not require all tasks to reach the same synchronization point except when this is specifically desired, and synchronization is performed dynamically by the runtime.

7 Conclusions and Future Work

In this paper, we present a design and implementation of *phasers*, a high-level synchronization construct for task-parallel programs, on a manycore Cyclops64 architecture. We show that phasers can be implemented very efficiently, providing a flexible and high-level programming unified construct for global and group barriers, point-to-point signal/wait and wavefront synchronization on manycore architectures.

We have designed and implemented four different techniques for phaser synchronization on

Cyclops64 that use a combination of software-based busy-wait approach, hardware barriers, and hardware support for thread suspend/awake.

Our experiments show that phasers, using our implementations on Cyclops64, outperform OpenMP barriers by up to two orders of magnitude, and also deliver performance that is comparable to the hardware barriers on Cyclops64, while providing much more portable, flexible and higher-level synchronization mechanism to the programmer than either OpenMP barriers or the Cyclops64 hardware barriers.

In the future, we will experiment with more bandwidth-limited applications on Cyclops64 to evaluate the limitations of our busy-wait phaser implementation. We will also investigate techniques to allow dynamic creation of a number of tasks that exceeds the number of hardware threads on a Cyclops64 machine, without compromising the very high efficiency of our implementation of the simplified task-parallel programming model.

References

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an Object-Oriented Approach to Non-Uniform Cluster Computing,” in *OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA: ACM, 2005, pp. 519–538.

[2] “Chapel Programming Language,” http://chapel.cray.com/.

[3] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, “Phasers: a unified deadlock-free construct for collective and point-to-point synchronization,” in *ICS ’08: Proceedings of the 22nd annual international conference on Supercomputing.* New York, NY, USA: ACM, 2008, pp. 277–288.

[4] J. D. Cuvillo, W. Zhu, Z. Hu, and G. Gao, “Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture,” in *CAPSL Technical Memo 062*, 2005.

[5] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, “Tiny threads: A thread virtual machine for the cyclops64 cellular architecture,” in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05).* Washington, DC, USA: IEEE Computer Society, 2005, p. 265.2.

[6] S. Microsystems, “Fortress Programming Language,” http://projectfortress.sun.com.

[7] R. Barik, Z. Budimlic, V. Cave, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Tacsirlar, Y. Yan, Y. Zhao, and V. Sarkar, “The habanero multicore software research project,” in *OOPSLA ’09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications.* New York, NY, USA: ACM, 2009, pp. 735–736.

[8] V. Sarkar, "Synchronization using counting semaphores," in *ICS '88: Proceedings of the 2nd international conference on Supercomputing.* New York, NY, USA: ACM, 1988, pp. 627–637.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[10] A. Tavlove, *Computational Electrodynamics*, 1995.

[11] "The Top500 List," http://www.top500.org.

[12] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," http://www.netlib.org/benchmark/hpl, 2004.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, Jun. 1995, pp. 24–36.

[14] I. E. Venetis and G. R. Gao, "Mapping the LU Decomposition on a Many Core Architecture: Challenges and Solutions," in *ACM International Conference on Computing Frontiers (CF2009)*, Ischia, Italy, May 2009.

[15] M. F. et al., "The implementation of the Cilk-5 multithreaded language," in *Proceedings of PLDI'98*, Montreal, Quebec, Canada, Jun. 1998, pp. 212–223, proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[16] "OpenMP specifications," http://www.openmp.org/blog/specifications/.

[17] H. Humy, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryky, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnany, A. Marquez, S. Merali, S. S. Nemawarkarz, P. Panangaden, X. Xue, and Y. Zhu, "A design study of the earth multiprocessor," 1995.

[18] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren, "A study of the earth-manna multithreaded system," *Int. J. Parallel Program.*, vol. 24, no. 4, pp. 319–348, 1996.

[19] K. B. Theobald, "Earth: an efficient architecture for running threads," Ph.D. dissertation, Montreal, Que., Canada, Canada, 1999, adviser-Gao, Guang R.

[20] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 598–632, 1989.

[21] P. S. Barth, R. S. Nikhil, and Arvind, "M-structures: Extending a parallel, non-strict, functional language with state," in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture.* London, UK: Springer-Verlag, 1991, pp. 538–568.

[22] K. B. Theobald, G. Agrawal, R. Kumar, G. Heber, G. R. Gao, P. Stodghill, and K. Pingali, "Landing cg on earth: a case study of fine-grained multithreading on an evolutionary path," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM).* Washington, DC, USA: IEEE Computer Society, 2000, p. 4.

[23] K. B. Theobald, R. Kumar, G. Agrawal, G. Heber, R. K. Thulasiram, and G. R. Gao, "Developing a communication intensive application on the earth multithreaded architecture," in *Euro-Par 2000 Parallel Processing*, 2000.

[24] B. Goetz, *Java Concurrency In Practice.* Addison-Wesley, 2007.

[25] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, "Titanium Language Reference Manual," University of California at Berkeley, Berkeley, Ca, USA, Technical Report CSD-01-1163, 2001.