Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns

Shams Imam and Vivek Sarkar

Department of Computer Science, Rice University
{shams,vsarkar}@rice.edu

Abstract. In this paper, we address the problem of scheduling parallel tasks with general synchronization patterns using a cooperative runtime. Current implementations for task-parallel programming models provide efficient support for fork-join parallelism, but are unable to efficiently support more general synchronization patterns such as locks, futures, barriers and phasers. We propose a novel approach to addressing this challenge based on *cooperative scheduling* with *one-shot delimited continuations* (OSDeConts) and *event-driven controls* (EDCs). The use of OSDeConts enables the runtime to suspend a task at any point (thereby enabling the task's worker to switch to another task) whereas other runtimes may have forced the task's worker to be blocked. The use of EDCs ensures that identification of suspended tasks that are ready to be resumed can be performed efficiently. Furthermore, our approach is more efficient than schedulers that spawn additional worker threads to compensate for blocked worker threads.
We have implemented our cooperative runtime in Habanero-Java (HJ), an explicitly parallel language with a large variety of synchronization patterns. The OSDeCont and EDC primitives are used to implement a wide range of synchronization constructs, including those where a task may trigger the enablement of multiple suspended tasks (as in futures, barriers and phasers). In contrast, current task-parallel runtimes and schedulers for the fork-join model (including schedulers for the Cilk language) focus on the case where only one continuation is enabled by an event (typically, the termination of the last child/descendant task in a join scope). Our experimental results show that the HJ cooperative runtime delivers significant improvements in performance and memory utilization on various benchmarks using *future* and *phaser* constructs, relative to a thread-blocking runtime system while using the same underlying work-stealing task scheduler.

Keywords: Task Parallelism, Cooperative Scheduling, Delimited Continuations, Async-Finish Parallelism, Habanero-Java

1 Introduction

With the advent of the multicore era, it is clear that future improvements in application performance will primarily come from increased parallelism in software.

A dominant programming model for multicore processors is the Task Parallel Model (TPM), as exemplified by programming models such as Cilk [2], TBB [25], OpenMP 3.0 [24], Java's ForkJoinPool [22], Chapel [4], X10 [5], Habanero-C [31], and Habanero Java (HJ) [3]. Current implementations for the TPM provide efficient support for fork-join parallelism, but are unable to efficiently support more general synchronization patterns that are important for a wide range of applications. In the presence of patterns such as futures [17], barriers, and phasers [26], current TPM implementations revert to thread-blocking scheduling of tasks. Barriers and futures are two common synchronization patterns advocated by many industry multicore programming models that go beyond the fork-join model. But, there is as yet no demonstration of an effective solution to schedule programs with futures and barriers in a scalable fashion when the number of blocked tasks exceeds the number of worker threads.

In this paper, we address the problem of efficient cooperative scheduling of parallel tasks with general synchronization patterns. Our solution is founded on the use of *one-shot delimited continuations* (OSDeConts, Section 4.1) and *single-assignment event-driven controls* (EDCs, Section 4.2) to schedule tasks cooperatively in the presence of different synchronization patterns. The OSDeCont and EDC primitives can be used to support a wide range of *synchronization constructs* (SyncCons) including those where a task/event may trigger the enablement of multiple suspended tasks. This general case is not supported by work-stealing schedulers for Cilk and other fork-join models for task parallelism. While efficient continuation-based scheduling is well established for fork-join parallelism in well structured *tree-like* computations in projects such as Cilk and Manticore [10], we are unaware of any past work that supports more general (and a wide variety of) synchronization patterns in a scalable manner with support for large numbers of suspended tasks. To the best of our knowledge, our paper is the first to support synchronization patterns that represent arbitrary computation graphs through the use of one-shot continuations.

Our cooperative approach of using OSDeConts and EDCs is more performant than schedulers that spawn additional worker threads to compensate for blocked worker threads (as well as approaches that leave worker threads blocked without spawning new worker threads). Transparent use of OSDeConts allows us to leverage the benefits of event-driven programming while the user code remains in standard thread-based structure, thereby avoiding the need to write fragmented difficult to understand event-driven programs where logical units are broken down into multiple callbacks [9]. Section 2 uses simple example programs to illustrate the performance issues with scheduling programs that use blocking SyncCons and the productivity issues with event-driven programming. The contributions of this paper are:

– Use of OSDeConts and EDCs to create a new generic cooperative runtime for task-parallel programs (Section 4). We believe that any task-parallel SyncCon can be supported by this cooperative runtime. To the best of our knowledge, this is the first effort to systematically use OSDeConts to support a task-parallel runtime. A key challenge we address in our runtime is that

the resolution of a synchronization can, in general, trigger the enablement of multiple suspended tasks, a scenario that does not occur in traditional fork-join operations.
- We include recipes for implementing different SyncCons using the API exposed by our cooperative runtime (Section 5). These (and other) SyncCons are all treated uniformly by the runtime and can all be used together without issues in the same program.
- An implementation of our cooperative runtime for the HJ language which supports a large variety of SyncCons.
- Empirical evaluation of the performance of our cooperative runtime relative to a runtime that uses thread-blocking operations (Section 6). Our experiments on various benchmarks show that the cooperative runtime can achieve over 10× speed-up over a runtime that uses thread-blocking operations while implementing SyncCons such as *futures* and *phasers*.

The rest of this paper includes Section 3 which discusses related work and Section 7 which contains our conclusions. For the interested reader, [21] contains additional details on our implementation, including the use of an extended version of the open source Kilim bytecode weaver [28] to support OSDeConts. Our implementation conforms to all the constraints imposed by a standard Java Virtual Machine (JVM).

2 Motivating Examples

In popular task parallel runtimes such as those for HJ, X10, and Chapel, the runtime is usually able to handle synchronization points associated with fork and join operations without blocking the worker. However, other potential synchronization points (such as resolution of future results, point-to-point synchronization points, lock-based implementation of atomic regions) are blocking operations in the runtime. This may result in the worker threads being blocked, effectively resulting in fewer parallel threads that are executing. The runtime can compensate by creating additional worker threads, but this adds to overhead in the runtime as each thread needs its own system resources. In addition, context switching overhead is incurred when the blocked threads become unblocked.

On Intel processors it takes about 1100 ns per thread context switch (without cache effects) [27]. In contrast, object allocation, method call, and setting fields takes around 30 ns, 5 ns and 1 ns respectively [14] on the CLR (timings on the JVM should be similar). Our continuation creation scheme includes one object allocation, setting fields per live variable to be saved, and returning from method calls. Using continuations should cost less than 50 ns per method in the call chain. Besides, the compensation strategy of creating additional threads is contradictory to the goal of the TPM which relies on using comparatively few heavyweight threads to run many lightweight tasks. Finally, these current solutions do not scale as increasing the number of worker threads can eventually cause the runtime to crash due to exhaustion of memory or other system resources.

In addition to the problems mentioned above, presence of synchronization constraints can also lead to starvation situations when all available worker threads become blocked. In such scenarios, the program behavior can change when the parallel program is run with different numbers of worker threads with the starvation scenario not occurring when enough worker threads are provided to compensate for the number of tasks involved in the synchronization constraint.

```
1  public class CyclicProducers {
2    public static void main(final String[] args) {
3      // number of tasks to create
4      final int numTasks = 64;
5      finish {
6        final ItemHolder itemHolder = new ItemHolder(numTasks);
7        for (int i = 0; i < numTasks; i++) {
8          final int myId = i;
9          async {
10           // first produce an item
11           final int myProducedItem = produceItem(...);
12           itemHolder.put(myId, myProducedItem);
13           ...
14           // now consume item produced by neighbor
15           final int neighId = (myId + 1) % numTasks;
16           // wait until neighbor produces item
17           final Object itemToConsume = itemHolder.get(neighId);
18           consumeItem(myId, itemToConsume);
19 } } } } }
```

Fig. 1: An example that can lead to starvation when a thread-blocking runtime runs this program with too few worker threads.

Consider an example program, in Figure 1, which spawns a number of tasks which form a *ring*. Each task is involved in a two stage computation: in the first stage the task produces a value (in lines 11-12) and in the second stage the task consumes the value produced by its immediate right neighbor in the ring (in line 17-18). The synchronization constraint of having to wait for the neighbor to produce the item handled in the `get()` method of the `ItemHolder` data structure is not shown in the example, but one can imagine it being implemented by traditional locks in any of the languages mentioned above (HJ, Cilk, X10, etc.). Locks in these languages have blocking implementations and cause the tasks to block worker threads. Consider this program, which spawns 64 tasks, being run with 64 worker threads. In such a scenario, each of the spawned tasks would potentially be assigned to individual worker threads and each task will have an opportunity to run and produce a value. As a result, all of the blocking calls to `get()` would eventually be satisfied and the computation would complete. Instead, consider the program being run on a runtime with 32 worker threads. If the task scheduler schedules alternate tasks (i.e. tasks with id 0, 2, 4, ..., 62), each of them will produce their value and block in the call to `get()` since their neighbor has not been scheduled to run and never produces the value that these tasks want to consume. Since all available worker threads become blocked, no computational progress can be made and we have a starvation! If a cooperative runtime were used instead, no starvation would occur in this program irrespective of the number of worker threads used.

```
1  public class CyclicProducers {
2    public static void main(final String[] args) {
3      // number of tasks to create
4      final int numTasks = 8;
5      finish {
6        final promise<int>[] items = new promise<>[numTasks]...;
7        for (int i = 0; i < numTasks; i++) {
8          final int myId = i;
9          async {
10           // first produce an item
11           final int myProducedItem = produceItem(...);
12           items[myId].put(myProducedItem);
13           ...
14           // now consume item produced by neighbor
15           final int neighId = (myId + 1) % numTasks;
16           // trigger callback when neighbor produces item
17           asyncAwait(items[neighId]) {
18             final Object itemToConsume = items[neighId].nbGet();
19             consumeItem(myId, itemToConsume);
20 } } } } } }
```

Fig. 2: An event-driven version of Figure 1 where callbacks are used to avoid thread-blocking operations. A `promise` can be viewed as a container with a full/empty state that obeys a dynamic single-assignment rule. The `nbGet()` methods represents a non-blocking `get()` operation. The `nbGet()` can only be performed inside an `asyncAwait` block on any promise registered in its await clause (e.g. `items[neighId]` on line 17).

To avoid blocking, programmers can choose to write their code in an event-driven style with callback registrations. Figure 2 shows an event-driven version of Figure 1 where callbacks are used to avoid thread-blocking operations. In this version, the possible blocking calls to `get()` are replaced by a callback registration (at line 17-19) on the rest of the computation to run when the value from the neighbor is eventually produced. This version requires additional support from the language or runtime to allow callback registrations on the underlying primitive (e.g. `promise`) being used to implement the data structure. Though this version of the program is cumbersome to write, it will never display starvation irrespective of the task scheduler used since there are no blocking operations and worker threads can always be used to make computation progress.

```
1  public class FibCallback {
2    public static void fib (int n, promise<int> f) {
3    if ( n < 2) { f.put(n); return; }
4      promise<int> x = newPromise<>();
5      promise<int> y = newPromise<>();
6      async { fib(n-1, x); };
7      async { fib(n-2, y); };
8      asyncAwait(x, y) { f.put(x.nbGet() + y.nbGet()); }
9    }
10   public static void main (String[] args) {
11     int n = Integer.parseInt(args[0]);
12     promise<int> res = newPromise<>();
13     async { fib(n, res); };
14     asyncAwait(res) {
15       println(res.nbGet());
16 } } }
```

Fig. 3: Version of the Fibonacci numbers program that uses event-driven style with callbacks and asynchronous tasks. Asynchronous tasks are created with `async`; asynchronous callbacks are registered on promises using `asyncAwait`. The `fib()` method needs an extra parameter to store the promise and allow callback registrations. Calling `fib` does not return a result directly, rather an additional callback needs to be registered on line 14 to receive and display the result.

As another example, Figure 3 shows the classic (and inefficient) parallel version of the Fibonacci function written in an event-driven style. This style of programming makes writing and maintaining code somewhat onerous and error-prone. A key difficulty is that the logical unit of work is broken across callbacks and methods are passed extra parameters to help registering on the callbacks. There is no direct return of a value from the callee to the caller. These make the code harder to read and maintain, especially as the method size grows and multiple parameters need to be passed along the call chain.

```
1 public class FibFuture {
2   public static int fib (int n) {
3     if ( n < 2) { return n; }
4     future<int> x = async<int>{ fib(n-1); };
5     future<int> y = async<int>{ fib(n-2); };
6     return x.get() + y.get();
7   }
8   public static void main (String[] args) {
9     int n = Integer.parseInt(args[0]);
10    future<int> res = async<int>{ fib(n); };
11    println(res.get());
12 } }
```

Fig. 4: Version of the Fibonacci numbers program that uses futures for synchronization with asynchronous tasks. Calls to future `get()` wait until the value in the future becomes available. This example is aligned with thread-based code where no extra parameters are required to register callbacks and the function calls return values directly.

Figure 4 shows an example program to compute Fibonacci numbers using futures to asynchronously compute values of the subproblems. The code for this version follows a more standard program structure and is easier to read and maintain compared to Figure 3. The `get()` operations are potential synchronization points where the task may suspend itself if the value of the future has not already been resolved. In many current runtimes, these potential synchronization points could result in thread blocking operations. In our runtime, we handle the synchronization points cooperatively using OSDeConts without blocking the worker thread. This allows us to leverage the benefits of event-driven programming while the user code remains in standard thread-based structure (i.e. the user writes programs similar to Figure 1 and Figure 4). As we see in Section 6.2, the non-blocking version of Fibonacci with futures clearly outperforms a blocking version by a factor that exceeds 100×. Similar performance gains can also be achieved by cooperatively scheduling tasks with other SyncCons, such as phasers (see Section 6.3).

3 Related Work

The general idea of using event-based programming in thread-based code has also been explored by others in the past. In Tasks [9], explicit method annotations provide yield points. These annotations are used to translate the code into event-based style using a form of continuation passing style (CPS) translation. Unlike what its name might suggest, Tasks has nothing to do with task parallelism, instead it is a programming model for writing event-driven programs.

Our implementation requires no explicit method annotations, uses OSDeConts, and runs safely on a parallel scheduler (i.e. the operations are thread-safe).

Use of continuations for task parallelism was popularized by Cilk [2], an extension to C that provides an abstraction of threads in explicit CPS. Our approach uses OSDeConts to achieve the same goal as Cilk where there are no thread blocking operations in the generated code. We support additional SyncCons where a task may trigger the enablement of multiple suspended tasks (as in futures, barriers and phasers) in contrast to Cilk where only one continuation is enabled by an event (the termination of the last child/descendant task in a join scope). Since Cilk relies on serial elision to be equivalent to a sequential program, such programs are not supported in Cilk as there may be no equivalent sequential program which use these SyncCons. Having nonblocking operations allows us to provide proper time guarantees, since some progress is continually made towards the computation. In Cilk, such time guarantees are lost when locks, which are typically blocking, are used. However, supporting the time bound guarantee comes at a cost of space bound with all the additional space for temporary local variables in the heap.

The Intel Threading Building Blocks (TBB) [25] task scheduler is inspired by the early Cilk work-stealing scheduler. TBB deals with possible blocking operations by running other tasks on the same stack, effectively stitching the call stack of the new tasks on top of the blocked task's stack. TBB also allows the parent tasks to specify another "continuation" task that will continue its work when such blocking scenarios arise. This minimizes the load on the scheduler and the uncontrolled overflow of the stack. However, this places the burden on the programmer to detect and schedule tasks to avoid blocking. In our approach, the user does not have to deal with the blocking constructs manually, the runtime implicitly handles the creation of continuations and the scheduler picks the next tasks to execute. Also, since each task has its own stack, we do not have to worry about the stack overflowing due to stitching of frames from multiple tasks. Overall, we go a step further than Cilk and TBB by showing how additional SyncCons such as futures, phasers and isolated blocks can be supported in a nonblocking manner.

Qthreads [30] is a lightweight threading library for C/C++ applications that also uses call stack stitching, it allows spawning and controlling tasks with small (4k) stacks. Our runtime is based on OSDeConts and poses no limits on the stack size of tasks created by the runtime, the stack size for worker threads in our implementation is limited by the JVM thread stack size (default around 1M for 64-bit JVMs) and the limits for OSDeConts is defined by the size of the heap. The qthreads API provides access to full/empty-bit (FEB) semantics (producer-consumer pattern with mutable buffer) and the threads need to be able to interact with the FEB for synchronization. In our runtime, tasks synchronize among themselves using the EDC primitive which is based on the observer pattern.

Li et al. present an alternative approach to implement concurrency in Glasgow Haskell Compiler (GHC) [23]. The runtime offers continuations as a mech-

anism from which concurrency can be built and also supports preemptive concurrency of very lightweight threads. In their implementation of GHC, the list of suspended continuations is periodically polled by the scheduler to see if the cause for blocking has been resolved. We differ in that we use OSDeConts and avoid any polling while deciding to resume suspended tasks by allowing EDCs to add resumed tasks into the scheduler's work queue. Also, we run inside the JVM where we cannot create continuations directly and have to rely on CPS-like transforms to support OSDeConts.

Fluet et al. [10] use full continuations to support fine-grained parallelism in their Manticore project. Manticore, like the GHC, is based on a functional language. It relies on a tree of futures that allows stacking continuations and a comparatively limited set of synchronization patterns (mainly futures). In contrast, our abstractions support a wide variety of synchronization patterns (e.g. futures, phasers, atomic) and arbitrary computation DAGs where continuations may be placed in the work queue without restrictions.

The use of continuations in task parallel programs has also been proposed by the C++ implementation of X10 [29]. The work-stealing scheduler in their implementation supports the work-first policy inspired from Cilk. Their implementation supports distributed `async-finish` programs along with conditional atomic blocks but does not support clocks (a precursor to HJ phasers). In our approach we rely on the help-first policy to have independent stack frames for tasks to enable use of OSDeConts and can use either a work-sharing or work-stealing scheduler. Our cooperative runtime is general enough to support a wide variety of SyncCons as we prove in our implementation.

Continuations are also used in the Continuators construct for an implicitly parallel implementation of Scheme [19]. There continuations are used to invoke the body of a function application (without blocking the interpreter) after the arguments have been evaluated in parallel. We employ delimited continuations with the same goal of avoiding thread blocking operations, additionally, our proposal provides an API to implement SyncCons which subsumes the parallel argument evaluation case. Also, our implementation dynamically discovers suspension points and minimizes overhead by avoiding continuation creations when EDCs have already been resolved.

4 Cooperative Runtime for Task Scheduling

The general TPM allows programmers to represent their computations as directed acyclic graphs with dependences between inter-dependent tasks. As a result, there has been a lot of work done in developing structured synchronization constructs (SyncCons) on the TPM by the community. These constructs include the well structured `async-finish` variant of fork-join style tasks, point-to-point synchronization with futures, localized and group synchronization using phasers, and weak atomicity in critical sections [2,17,26,3]. Such constructs introduce new challenges for the runtime while scheduling and executing tasks.

Synchronization constraints can prevent a currently executing task from making further progress as it *waits* to synchronize with other ready but not executing task(s). Many task parallel runtimes implement such waits by either busy-waiting until the constraint is resolved or by blocking the worker thread. An alternative approach is to use cooperative scheduling of tasks where an executing task, via runtime support, decides to actively suspend itself and yield control back to the runtime. The runtime can then perform book-keeping on the suspended task and use the worker thread to execute other ready tasks. The suspended task can be resumed and scheduled for execution when the synchronization constraint that caused it to suspend is resolved. This approach allows the runtime to continue making progress in the computation and to constantly exploit available parallelism during application execution without spawning additional threads. This nonblocking approach enables us to provide proper time guarantees since each worker is actively making some progress towards the computation. Supporting the time bound guarantee comes at a cost of space bound since many tasks may be *in flight* (either suspended, ready, or executing) with all the additional space for temporary local variables in the heap.

In the rest of the section, we present some background on one-shot OSDeConts and EDCs. Then we describe the API we expose in our runtime to allow language/library developers implement a variety of SyncCons. Finally, we explain how OSDeConts and EDCs are used to build a cooperative task-parallel runtime.

4.1 One-shot Delimited Continuations

Delimited continuations (DeConts) were introduced by Felleisen in 1988 [8] where he referred to them as *prompts*. Continuations represent the rest of a computation from any given point. They refer to the ability to *capture* the state of a computation at that point, the computation can later be *resumed* from that point by resuming the continuation. In contrast, DeConts represent the rest of the computation from a well-defined outer boundary, i.e. a subcomputation. This allows DeConts to return to their caller allowing the program to proceed at the call site. DeConts are hence a good choice when a limited part of the computation needs to be saved/restored [6]. In general, a continuation can be resumed multiple times from the same captured state; however one-shot continuations refer to continuations that are resumed at most once. This guarantee makes them cheaper to implement because they don't require making additional copies of the state.

4.2 Event-Driven Controls

Event-Driven Controls (EDCs) are an extension to Data-Driven Controls (DDCs) which were presented in [20] and used to support event-driven actors in a task parallel runtime. A DDC lazily binds a value and a closure called the execution body (EB), both the value and the EB follow the dynamic single-assignment property ensuring data-race freedom. When the value becomes available, the

EB is executed using the provided value. We generalize DDCs to EDCs in this work to allow multiple EBs to be attached to the EDC as callbacks. We treat the availability of a value in the EDC as an *event* and use the event to trigger the execution of EBs. Due to the single-assignment property, the registered EBs are executed at most once. We also allow multiple values to be added into the EDC as long as the values are *logically equivalent*, this does not violate the dynamic single assignment property and it does not trigger re-executions of the EBs. Attempting to add unequal values into the EDC is reported as a runtime error. Figure 5 shows a simplified implementation of a DDC excluding SyncCons. The EB of the EDC may be executed either asynchronously or synchronously. For example, in a task parallel runtime the EB could store book-keeping data and act as a synchronous callback into the runtime. The EB could trigger possible asynchronous actions, such as scheduling and execution of a task, by interacting with the runtime.

```
1 class EventDrivenControl {
2   ValueType value = ...;
3   List<ExecBody> ebList = ...;
4   /** triggers callback execution **/
5   void setValue(ValueType theValue) {
6     if (!valueAvailable()) {
7       value = theValue;
8       // execute the callbacks/EBs
9       ebList.each().scheduleWith(value);
10    } } else {
11      // check for error
12  } }
13  /** enables callback registration **/
14  void addExecutionBody(ExecBody theBody) {
15    if (valueAvailable()) {
16      // value available, execute immediately
17      theBody.scheduleWith(value);
18    } else {
19      // need to wait for the value
20      ebList.add(theBody);
21 } } }
```

Fig. 5: Simplified representation of an EDC not displaying synchronizations or validations. Both the value and the execution body can be lazily attached. The execution body determines whether it is scheduled asynchronously or synchronously in the `scheduleWith()` method.

4.3 Cooperative Runtime - Design

To allow library/language developers to create their own SyncCons, we expose EDCs as an API in our runtime. The OSDeConts created to manage the book-keeping are not exposed to the developer; this is especially desirable since continuations are notorious for being hard to use and to understand by developers (as opposed to compilers and runtime systems). The API contains the following operations:

– The static `newEDC()` factory method is used to instantiate a new EDC. EDCs are initialized without a resolved value and with an empty EB list. The EDC can be used like a regular object, e.g. stored as a field, passed around as parameters, invoked as receivers for methods, etc.

– The static `suspend(anEdcInstance)` method signals possible creation of a suspension point. If the EDC passed as an argument has not been resolved, the current task is suspended and the runtime handles the book-keeping to register an EB to resume the task when the EDC is resolved.
– The `setValue(someValue)` method resolves the EDC, i.e. it binds a value with the EDC and triggers the execution of any EB registered with the EDC. Suspended tasks registered with the EDC will be resumed and scheduled for execution by the runtime.
– The `isValueAvailable()` can be used to check whether the value in the EDC has been resolved.
– The `getValue()` method retrieves the value associated with the EDC. It is only safe to call this method if the value in the EDC has already been resolved. If execution proceeds past a call to `suspend()`, it is guaranteed that a value is available in the EDC.

With these operations in place, language/library developers can implement their custom SyncCons and synchronization patterns. The same API is used in our implementation of Habanero-Java to support the constructs such as end of `finish`, futures, phasers, etc. For example, Figure 6 shows how simple it is to implement futures using the exposed API. A single EDC is used to suspend all consumers who try to read the value of the future before it has been resolved. When the value of the future is available, the EDC is resolved with a call to `setValue()` and any suspended consumer tasks are resumed by the runtime.

```
1 class Future<T> {
2   EventDrivenControl<T> edc = EventDrivenControl.newEDC<T>();
3   public void put(T item) {
4     edc.setValue(item); // resumes consumer(s)
5   }
6   public T get() {
7     // suspend consumer task till value produced
8     EventDrivenControl.suspend(edc);
9     // return value after it is resolved
10    return edc.getValue();
11 } }
```

Fig. 6: Futures implemented using the EDC API provided by the cooperative runtime. All consumer tasks suspend until the item is produced. Once the item is available, multiple suspended consumers are resumed by the runtime.

4.4 The Cooperative Runtime

In our cooperative runtime, when a potential synchronization point is discovered dynamically, thread blocking operations are avoided by suspending the currently executing task and cooperatively scheduling other ready tasks from the work queue. When the EDC is resolved, the suspended task (and its continuation) is put back into the work queue to eventually be resumed by a worker thread. Task suspensions are implemented by using standard OSDeConts and this guarantees that the runtime never spawns more worker threads than it was initially started with. The trade-off is that the compiler and the runtime now need to support

the overhead of creating the OSDeConts and handling the management of the EDCs in addition to the management of threads and tasks.

A pictorial summary of our runtime is provided in Figure 7. The runtime cooperatively schedules tasks using OSDeConts and EDCs in the presence of arbitrary dependences or synchronization constraints. The runtime places tasks into queues while the pool of worker threads continuously attempt to execute tasks dequeued from these queues. Execution of tasks may result in more tasks being spawned and enqueued into the queues. An application starts with a single *main* task in the work queue which promptly gets executed by one of the worker threads. The application terminates when *a*) the work queues are empty; and *b*) all synchronization constraints in the program have been satisfied (i.e. no deadlocks).

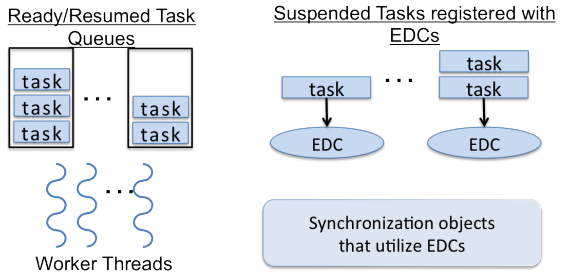

Fig. 7: The cooperative runtime includes worker threads and ready task queues like most other task parallel runtimes. In addition, there are EDCs which maintain a list of suspended tasks to implement higher-level synchronization constructs. Resolving an EDC moves a suspended task into the ready queue.

Our runtime uses a *help-first* policy [15] while scheduling tasks. Under this policy, spawning a child task enqueues it is in the task queue and allows the parent task to continue execution past the spawn operation. The child task hence has a stack of its own and can be executed by any of the worker threads. The independent stack allows us to treat the task as a subcomputation and to have a well-defined outer boundary while forming the OSDeCont. In contrast, using a *work-first* policy [15] does not provide an independent call-stack for a spawned task and requires maintaining fragmented call-stacks to allow helper threads to resume computations. This precludes the use of OSDeConts in a work-first policy (though the work-first policy can be more efficient than help-first for recursive divide-and-conquer parallelism when steals are infrequent, the work-first policy cannot be used to support general SyncCon). In addition, constructs such as phasers are not amenable to work-first scheduling since these constructs do not satisfy the "serial elision" property.

With the help-first policy in effect, we wrap the stack of each task around an OSDeCont which defines an `execute()` method as the continuation boundary. When a worker thread executes a task, it resumes the computation of the OSDeCont which in turn invokes the `execute()` method, as shown in Figure 8. At synchronization points where a task is not allowed to make progress semantically, an OSDeCont is captured and only the state until the `execute()` method

needs to be saved. On returning from a call to `execute()`, the runtime verifies the cause for the return and performs book-keeping if the task was suspended. The worker thread then goes ahead and tries to dequeue other scheduled tasks to execute and continue making progress towards the overall computation.

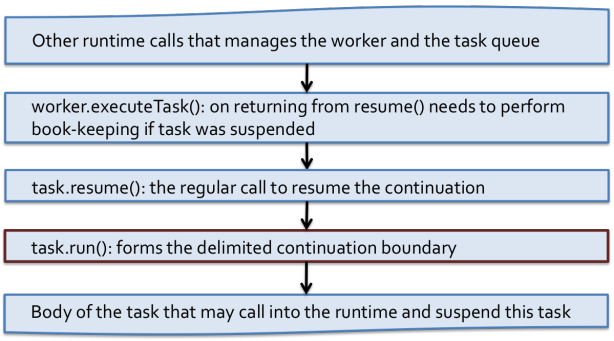

Fig. 8: Representation of the runtime call stack when a task is being executed by a worker thread. The `worker.executeTask()` method is responsible for managing the OSDeConts that may be suspended while executing the body of a task.

In our runtime, the static `suspend` method of the API (Section 4.3) restricts the cause of OSDeCont suspensions to instances of EDCs. On returning to the `worker.executeTask()`, the runtime checks whether an EDC was returned as a cause (i.e. the task was suspended) and registers an EB with that EDC. There is no limit to the number of tasks that can be registered to an EDC (in the form of an EB). When the EDC is resolved, the EBs are executed and the suspended tasks are rescheduled. Note that this approach does not need to use polling to keep track of when suspended tasks can be resumed. After being scheduled, the queued task is picked up by a worker thread and execution is resumed from the previous suspension point. When the execution of the task completes normally, without suspending, the runtime performs any cleanup operations associated with the task and looks for more work from the queue.

The remaining pieces in the runtime are the steps to undertake where synchronization points: *a*) capture continuations, *b*) create EDCs, and *c*) resolve EDCs. The use of OSDeConts and EDCs are abstracted by the implementer of the SyncCons and transparent to an end user of these constructs. We discuss how various SyncCons can be developed in our runtime in Section 5.

Work-Stealing Scheduler The exact policy to retrieve tasks from work queues is unspecified in our cooperative runtime. Recent work has shown that work-stealing policies work very well on multicore architectures. A scheduler using a work-stealing policy maintains a queue of pending tasks per worker thread. When a worker completes a task, it pops a pending task from its own queue. If the queue is empty, it attempts to steal a task from another worker's queue. Our runtime uses the help-first policy and maintains an independent stack for each task, the OSDeCont created is thread independent and can be run by any

thread. Hence, any worker thread may execute a task and we are able to use both the work-stealing or work-sharing scheduling policies in our runtime.

Serializability of Computations Serializability of a group of parallel or concurrent statements refers to the ability to provide a serial ordering of the statements. In our runtime, since a single worker thread can execute the entire computation, that schedule provides a serializable order for the statements. The caveat is that the granularity of the statement blocks is around suspension points rather than user-written tasks. These new statement blocks can be used to form structures to represent the program dependence graph of the computation and reason about parallel portions and simplify, for example, data race analysis. With additional support from a scheduler, the statement blocks from the dependence graph can be scheduled in a deterministic order if so desired or can be used to generate different schedules, both of which can be very useful for debugging programs.

5 Support for Synchronization Constructs

Synchronization constructs (SyncCons) are used to coordinate the parallel execution of tasks. In this section, we describe how various SyncCons can be supported by our cooperative runtime. The key idea is to translate the coordination constraints into producer-consumer constraints on EDCs and to use OSDeConts to suspend consumers when waiting on item(s) from producer(s). We claim that any task-parallel SyncCon can be translated in such a manner and hence be supported by our runtime. The constructs we present include: *a*) termination detection of child tasks, *b*) producer-consumer synchronization, *c*) collective barrier synchronization, *d*) single blocks executed by only one task in a group, and *e*) weak isolation while accessing a shared resource. While constructs a) through d) are typically used for deterministic parallelism, construct e) can be used to support nondeterminism as well.

5.1 Fork-Join Synchronization

In structured fork-join parallelism, a *parent* task can spawn one or more *child* tasks that can logically run in parallel with the parent task. The parent task can then wait, by joining, until all of its transitively spawned children complete execution. An EDC, which wraps a counter[1], is created for each parent task. The counter is atomically incremented each time a child task is forked and atomically decremented as each child task completes execution, either normally or abnormally. When the count reaches zero, the value of the EDC is resolved. The join operation serves as a possible suspension point in our runtime and uses the EDC as its cause for suspending if it is invoked before the count reaches zero. If the count is zero when the join operation is called, execution of the parent task

[1] Distributed counters can be used for increased scalability.

continues without the need for suspension. This model can be easily extended to also support nested fork-join parallelism.

5.2 Producer-Consumer Synchronization

In producer-consumer patterns, producer tasks are responsible for resolving the values inside EDCs while consumer tasks suspend until the value inside an EDC has been resolved. A common case is the single-producer multiple-consumer case, also known as *futures* [17]. A future represents an immutable value, an EDC in our runtime, which will become available at a later point by a producer task. When the producer task completes execution it resolves the value inside the EDC thus resuming any previously suspended consumers. Consumers who read the value of the future after it has already been resolved can continue execution without being suspended. The single-producer single-consumer case can be supported by further wrapping an EDC and ensuring that only one consumer is able to read the value of the EDC, read requests from other consumers report an error.

The general producer-consumer problem with a mutable buffer location can also be modeled using our API. An example of such a construct is the synchronization variable construct available in Chapel [4]. In effect, the buffer location is either empty or full and producers/consumers need to wait when the location is full/empty, respectively. This can be modeled in our runtime by maintaining a doubly-linked list of a pair of EDCs and a pointer to the active pair. The first element in the pair represents whether a producer has produced the item making the location full, while the second element represents whether a consumer has consumed the item making the location empty. A producer suspends until the previous consumer-EDC has been resolved, while a consumer suspends until the producer-EDC in the currently active pair has been resolved. Separate producer and consumer pointers are maintained and they are advanced to the next node in the list when write and read operations are invoked, respectively.

An example implementation of synchronization variables recipe using the cooperative API is provided in Figure 9. There are two pointers being maintained to track progresses made by producers and consumers. A pair of EDCs are maintained to ensure there is the strict alternation of writes and reads by producers and consumers, respectively, while accessing the synchronization variable. If a producer arrives before the previous value has been read by a consumer it is suspended and vice versa.

5.3 Collective Barrier Synchronization

A barrier synchronization provides a means to ensure a group of tasks have all arrived at a particular point before advancing. This is especially useful in phased computations by ensuring each task in the group of tasks has completed one phase before starting the next phase of the computation. It is possible that the group of tasks involved in the barrier remain static or change dynamically

```
class EdcPair<T> {
  EventDrivenControl<T> p = EventDrivenControl.newEDC<>();
  EventDrivenControl<Boolean> c = EventDrivenControl.newEDC<>();
}
class SynchronizationVariable<T> {
  Node<EdcPair<T>> pNode; // producer chain
  Node<EdcPair<T>> cNode; // consumer chain
  Node<EdcPair<T>> nextNode(Node<EdcPair<T>> n) {
    if (n.nextNode == null)
      n.nextNode = new Node<>(n, ...);
    return n.nextNode;
  }
  public SynchronizationVariable() {
    Node<...> item = new Node<>(null, ...);
    item.c.setValue(true);
    cNode = pNode = nextNode(item);
  }
  public void write(T item) { /*suspendable method*/
    Node<EdcPair<T>> n;
    isolated { n = pNode; pNode = nextNode(n); }
    EventDrivenControl.suspend(n.prevNode.c);
    n.p.setValue(item);
  }
  public T read() { /*suspendable method*/
    Node<EdcPair<T>> n;
    isolated { n = cNode; cNode = nextNode(n); }
    EventDrivenControl.suspend(n.p);
    n.c.setValue(true);
    return n.p.getValue();
} }
```

Fig. 9: Synchronization variables implemented using operations provided in the cooperative runtime. Producers suspend until the previous item is consumed, consumers suspend until the current item is produced.

over time, either form of barriers can be supported by our API/runtime. Implementing barriers in a runtime that uses thread-blocking operations is not scalable if the number of tasks registered on the barrier exceeds the number of available worker threads. This can lead to deadlocks if the runtime is not allowed to create additional worker threads to allow all tasks to reach the barrier and release the blocked threads. In the case where the runtime can compensate by creating additional worker threads, scalability and efficiency are affected due to the overhead of having to manage additional worker threads. In our cooperative runtime, since there are no thread blocking operations, the tasks can suspend themselves if they arrive too early at a barrier allowing the worker threads to execute other ready tasks and reach the barrier point. Eventually all tasks will arrive at the barrier and the suspended tasks will be resumed.

To support barriers with dynamic task registration (the static task version is a special case), we maintain a count of registered tasks, a count of arrived tasks, and an EDC for each phase in the barrier computation as shown in Figure 10. When a task dynamically registers on the barrier, it registers on the *next* phase and increments the count of registered tasks for that phase. However, a task always deregisters in the current phase of the barrier and increments the arrived task count. As each task arrives at the barrier it increments the count for arrived tasks in the current phase and the count for registered tasks in the next phase. Additionally, if the task is not the last to arrive at the barrier point it suspends itself using the EDC for the current phase as the cause. The last task to arrive at

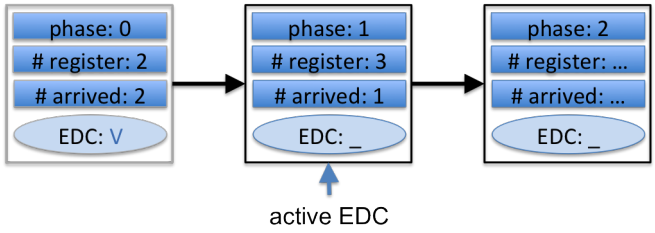

Fig. 10: The barrier represents each phase with two counters to keep track of registered and arrived tasks and an EDC which is used to track *early* arrivers. As each phase completes, the EDC in the current phase is resolved resuming suspended tasks and the active phase pointer is moved to the next item in the linked list.

the current phase of the barrier resolves the EDC of the current phase, advances the phase of the barrier, and continues without suspending. Resolving the EDC resumes all the tasks suspended on the barrier and the tasks now participate in the next phase of the computation when executed.

5.4 Phaser Synchronization

An extension to barrier synchronization is provided by phasers [26]. They unify collective and point-to-point synchronization for phased computations. Unlike traditional barriers where tasks register in *signal-and-wait* mode, tasks can also be registered on a phaser in *signal-only* or *wait-only* modes. Tasks registered on a wait mode (wait-only or signal-and-wait) need to wait for all tasks registered on a signal mode to arrive at the barrier point. The implementation for barriers (Section 5.3) needs to be extended by allowing only signalers to increment the counts of their local phase. Since signaler tasks can be in different phases, care is required to ensure that the correct counters are incremented. Tasks registered in signal-only mode never suspend and continue to make progress. Tasks registered in wait mode need to suspend themselves and wait for the EDC for a given phase to be resolved when all signaler tasks for a given phase arrive at the barrier point. As the EDC for the *oldest* phase is resolved by the last signaler task, it also advances the current phase for use by the waiter tasks.

5.5 Single Blocks

The OpenMP `single` construct specifies that a statement block is executed by only one task among a group of registered tasks [24]. The *wait* version requires that all registered tasks wait until some task has executed the `single` block. This is similar to supporting barriers with a single phase. All tasks, except the last task, that arrive at the `single` suspend themselves. The last task that resolves the EDC executes the statement block before resolving the EDC and causing the tasks registered on the `single` to be resumed. The *nowait* version does not require to suspend tasks, it requires some bookkeeping to ensure that exactly one task to arrive at the `single` executes the statement block.

Phasers also support a variant of `single` blocks when tasks are registered using the *signal-wait-single* mode. The semantics defines that the single block

is executed only after all the signalers and waiters have arrived at the `single` block. Both the signalers and the waiters need to ensure they proceed only after all signalers have arrived and at least one task has executed the `single` block. Supporting such blocks in our runtime requires the use of two EDCs for each phase of the phaser. The first EDC keeps track of whether all signalers have arrived while the second EDC is used to track whether the statements inside the `single` block has been executed by some task. Thus tasks can possibly be suspended twice while executing a single block.

5.6 Weak Isolation

Habanero-Java provides the isolated SyncCon, which can be used to implement critical sections and coordinate the mutation of shared data. The weak isolation guarantee states that the statements inside the critical sections will be executed mutually exclusively with respect to other demarcated critical sections (DCS). In general, weak isolation enforces a serializability bottleneck as only one critical section may be executed by the runtime in the absence of a more sophisticated analysis. Often this serializability is implemented using locks where worker threads block while waiting to attain the lock. Use of locks can limit performance in scenarios where there is moderate or high contention for the lock by the interfering DCS. In the cooperative runtime, blocking of threads while using locks is avoided by maintaining a dynamic linked-list of EDCs. Each task executing the DCS registers itself to an EDC in the list and suspends itself if it does not link to a resolved EDC. The first EDC in the list already resolved by default to allow the first requestor of the lock to make progress in its DCS without suspending. Any task linked to a resolved EDC gets to execute its DCS and resolves the next EDC in the list.

6 Experimental Results

The benchmarks were run on individual nodes in a IBM POWER7 compute cluster. Each node contains 256GB of RAM and four eight-core IBM POWER7 processors running at 3.8GHz each. There is a 32 kB L1 cache and a 256 KB L2 cache per core. The software stack includes IBM Java SDK Version 7 Release 1 and Habanero-Java (HJ) version 1.3.1 (r33926). Each benchmark used the same JVM configuration flags[2] and was run for ten iterations in ten separate JVM invocations, the arithmetic mean of thirty execution times (last three from each invocation) are reported. This method is inspired from [13] and the last three execution times are used to approximate the steady state behavior. In the bar charts, the error bars represent one standard deviation.

We implemented our cooperative runtime in the Habanero-Java (HJ) language supporting all its available constructs without requiring any changes to

[2] Flags: `-Xms6344m -Xmx65536m -XX:MaxPermSize=256m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:-UseGCOverheadLimit`

the syntax of HJ programs, i.e. users are unaware of the use of EDCs and one-shot OSDeConts by the runtime. Our implementation is briefly described in [21]. The implementation of our cooperative runtime for HJ conforms to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, such JVMs do not provide support for continuations or for storing and restoring the stack. We use an extended version of the open source bytecode weaver provided by the Kilim framework [28] to support one-shot OSDeConts that are thread independent and can be restored on threads managed by the HJ runtime.

We focus on benchmarks for the different SyncCons to compare the performance of our implementation of the cooperative runtime with *a*) the existing work-sharing runtime available in HJ which has *blocking* implementations for most of the SyncCons, and *b*) the `ForkJoinPool` and helper classes from the `java.util.concurrent` package in Java like `AtomicInteger`, `CyclicPhaser`, `CountDownLatch`, etc. [3] Both these blocking runtimes have been shown to deliver performance competitive with other runtimes (e.g., OpenMP, X10), for general SyncCons. Both HJ runtimes were configured to use the same work-stealing scheduler (`ForkJoinPool` from the standard JDK) as previous experience has shown the scheduling policy to be more effective than a work-sharing policy. All the benchmarks were run using **thirty-two** worker threads as the starting seed. In the blocking runtimes, additional threads are created around blocking suspension points, while the cooperative runtime never creates more worker threads. All benchmarks use the same algorithm in their implementation. For the HJ file(s) as input, only the runtime is switched from the blocking version to the cooperative version during compilation.

6.1 Fork/Join Benchmarks

Figure 11 shows the result of fork-join benchmarks using the `async-finish` constructs from HJ. The first benchmark is the Java Grande Forum (JGF) Fork-Join (FJ) microbenchmark [7], it measures the time taken to spawn and join asynchronous tasks inside a single `finish` scope. Each task does a minimal amount of work before it terminates. Since there is only a single `finish`, this benchmark effectively measures the relative overhead in the two runtimes to spawn and manage tasks. The cooperative runtime is slower by about 40% as it has the overhead of wrapping the task in a OSDeCont and checking whether the task suspended when it is executed (even though the tasks themselves never suspend in this benchmark). One optimization technique to reduce the overhead is to avoid transforms of tasks that are statically known to be nonblocking. The next benchmark, N-body (Computer Language Benchmarks Game [12]), shows a similar slowdown while using the cooperative runtime as there are few `finish` blocks created compared to `async` blocks (a ratio of 1:7). The effects of blocking are shadowed to some extent since there are fewer join points resulting in

[3] Not all benchmarks have a corresponding Java implementation / data for execution times of the benchmarks. In such cases the performance numbers are not shown in the charts.

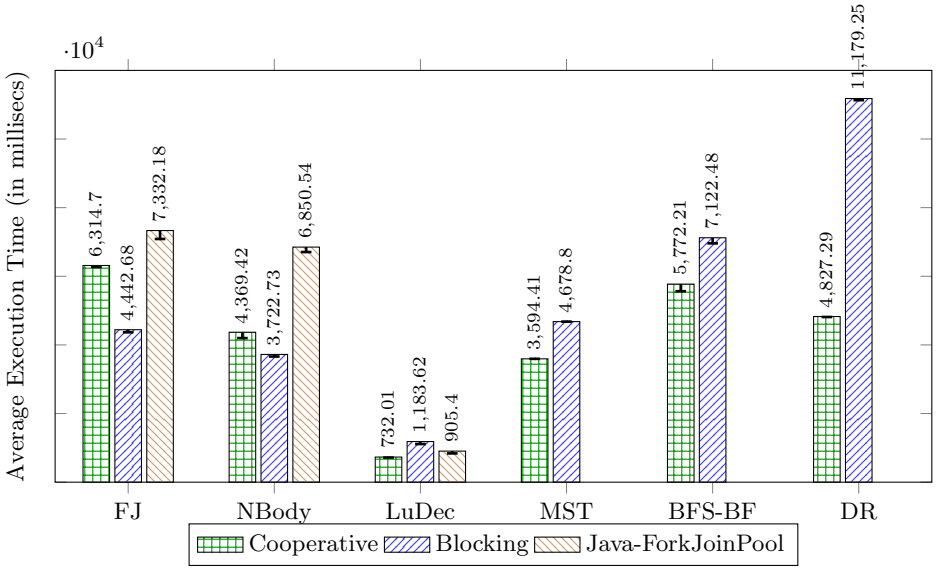

Fig. 11: Results for `async-finish` benchmarks. JGF Fork Join (FJ) with 4 million tasks. NBody with 300K steps. LU-Decomposition (LuDec) with an array size of 2K and block size of 128. MST, BFS-BF and DR with an input graph of size 512 nodes and artificial load values of 500K, 20M, and 8M respectively.

fewer blocked threads. The LU Decomposition (LuDec) benchmark, from the Cilk Benchmark [1], is the first to show speed-up of around 60% as there are relatively more `finish` blocks created compared to `async` blocks. The next set of benchmarks, Minimum Spanning Tree (MST), Breadth First Search using Bellman Ford algorithm (BFS-BF), and Dijkstra Routing (DR), come from the IMSuite benchmark suite [16] for various graph algorithms written in task parallel languages. The relatively larger number of finish blocks allow the cooperative runtime to show speedups of around 20% for MST and BFS-BF, while the DR benchmark executes over 2× faster. The Java versions of the benchmarks have their `finish` implemented using atomic integers and latches which have blocking semantics. The Java versions are slower than the cooperative runtime implementation in all the fork-join benchmarks, for which data is available, due to blocking at the end of `finish` using countdown latches.

6.2 Future Benchmarks

The Fibonacci microbenchmark does almost no computation inside the task thus allowing us to measure the overheads of the future SyncCon in the two runtimes. Two versions of the Fibonacci program were mentioned in Figure 3 and Figure 4. With the cooperative runtime we achieve performance close to the program written in event-driven style while still using the easier to read thread-based style. The cooperative version using futures comfortably outperforms the blocking version, e.g. computing the 20th term of fib resulted over 100× speed-up. In fact, the blocking version runs out of memory for values of `n` larger than 20 as the

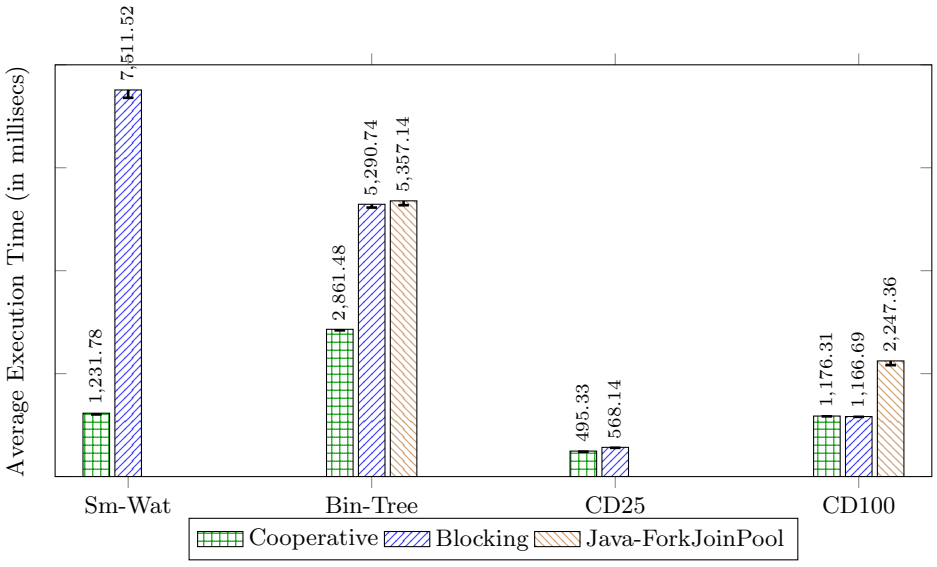

Fig. 12: Smith Waterman on strings of length 960 and 928. Binary Tree operating on a tree with depth of 14. Cholesky Decomposition on an input matrix of size 2000×2000 with tile sizes of 25 and 100.

runtime attempts to create extra threads to compensate for the blocked threads while the cooperative version completes in around 100 milliseconds. The Java future implementation is similar to the HJ blocking runtime's implementation of futures. Hence, the running times for the future benchmarks on these two variants are similar.

In the Smith-Waterman benchmark, futures are used to represent the value at each cell of the dynamic programming *table* and backtracking starts at the highest corner cell. Each cell depends on values from three neighboring cells and thus each cell has exactly three suspension points, once for each attempt to resolve the future of a neighboring cell, while computing its own value. Due to the comparative lack of delay while trying to resolve a future after its creation many blocking operations are performed in the blocking runtime. This degrades performance, as seen in Figure 12, and the cooperative version outperforms the blocking version by a factor of 6×.

The Binary Trees benchmark, from the Computer Language Benchmarks Game [11], involves allocating binary trees, walking the trees bottom-up, and deallocating many nodes after the walk. In this benchmark, there is a relatively larger delay between the creation of the future and the attempt to resolve its value. This nature of the benchmark allows the blocking scheduler to make some progress in executing the futures and thus helps minimize blocking operations due to calls on unresolved futures. Even with this property, the cooperative runtime still outperforms the blocking version by a factor close to 2×.

The next benchmark is Cholesky Decomposition, a dense linear algebra application. We use futures to enforce the data dependences and exploit loop and

pipeline parallelism. With smaller tile sizes, more tasks/futures are created and there is a higher probability of blocking on a future. Hence, the cooperative runtime performs better by about 13% at the smaller tile size of 25. As the tile size increases there are fewer blocked threads on unresolved futures and the blocking runtime performs as well as the cooperative runtime.

6.3 Phaser Benchmarks

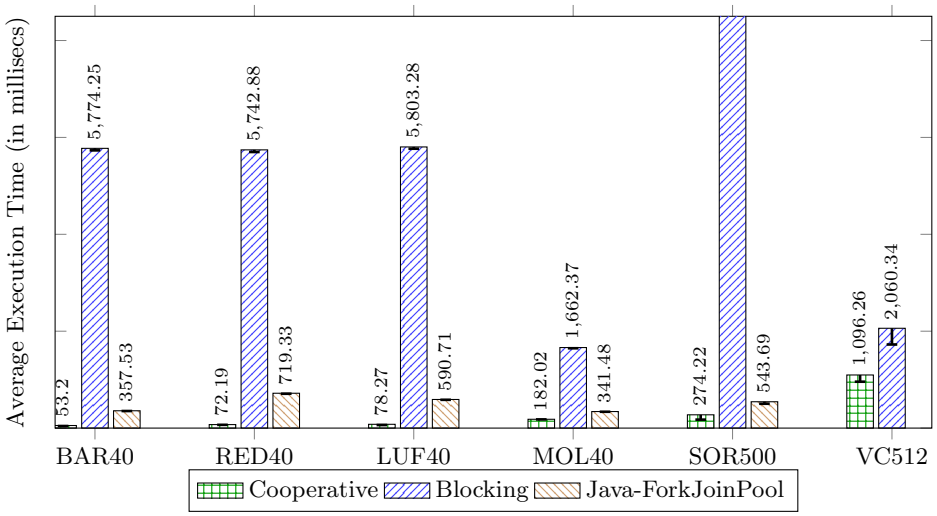

Fig. 13: Phaser benchmark results. BAR, RED, LUF, and MOL with 40 tasks registered on the phaser. SOR benchmark with an input array size of 500. VC coloring with an input graph of 512 nodes and artificial load of 10M.

To compare the performance of phasers we implemented two microbenchmarks: Barrier (BAR) (tasks registered on phasers in *sig-wait* mode) and Reduction (RED) (tasks registered on phasers in *sig-wait-single* mode using sum accumulators). We also implemented two additional benchmarks from the JGF benchmark suite: Moldyn (MOL) and LU-Factorization (LUF). Since our hardware had thirty-two cores, we registered more than forty tasks (i.e. more than thirty-two) on the phasers to stress test the runtimes. This ensures that most of the registered tasks encounter a *forced* suspension point at the *next* operation as only a maximum of thirty-two tasks can be running at any given time. In the blocking runtime, each such suspension point causes the worker thread to block and additional threads are created to run the other tasks. As such the runtime has to deal with the overhead of these additional thread context switches. In contrast, the cooperative runtime avoids such thread context switches and relies on the continuations to perform the relatively lightweight task context switches. We can see that we can achieve more than 100× for BAR, RED, and LUF and close to 10× speed-up for MOL in the cooperative runtime. These speed-ups are even greater when more than forty tasks are registered on the phasers. We

also include results from two other phaser benchmarks: JGF's Successive Over-Relaxation (SOR) with an input size of 500 and IMSuite's Vertex Coloring with an input graph of 512 nodes. In these benchmarks, where more computation is done between phases and overheads from context switching can be amortized to some degree, we get over 50× speed-up for SOR and 2× speed-up for VC.

Java's implementation of `CyclicPhaser` is noticeably more performant than the HJ blocking runtime version. However, `CyclicPhaser`s are still slower than our cooperative implementation which does not block worker threads. On the various phaser benchmarks, the HJ cooperative version ranges from being 2× to 10× faster than the pure Java version.

7 Conclusions and Future Work

In this paper, we address the problem of scheduling parallel tasks with general synchronization patterns using a cooperative runtime for scalability and performance. Our solution is founded on a novel use of one-shot delimited continuations and event-driven controls. We describe recipes for implementing various SyncCons using our cooperative API and provide an implementation of our cooperative runtime for the Habanero-Java language. Experimental results for our implementation for Habanero-Java, on various future and phaser benchmarks, show that the cooperative runtime delivers significant improvements in performance and memory utilization relative to a thread-blocking runtime system while using the same underlying work-stealing task scheduler.

We are working on further extending our cooperative runtime to support preemptive scheduling using the notion of Engines [18] to ensure fairness in the scheduling of tasks. Such a scheme also requires support for runtime generated priorities while scheduling tasks. Engines will enable us to support speculative parallelization, e.g. in the form of Cilk's [2] abort statement, more efficiently. Exploiting the dynamic dependence graph around suspension points to detect dataraces and to help in debugging is also an interesting area of future research which we are looking forward to pursue.

Acknowledgments:

This work was supported in part by NSF award CCF-0964520. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. The results in this paper were obtained on a system that was supported in part by NIH award NCRR S10RR02950 and an IBM Shared University Research (SUR) Award in partnership with CISCO, Qlogic and Adaptive Computing. We are grateful to Vincent Cavé and Jun Shirako for discussions on the Habanero Java runtime system and phasers, respectively. We also thank Akihiro Hayashi, Sağnak Taşırlar, and Jisheng Zhao for sharing their benchmarks with us. We are grateful to Deepak Majeti, Rishi Surendran, Nick Vrvilo, and the anonymous

reviewers whose feedback on earlier drafts helped improve the presentation of this paper.

References

1. Blumofe, R.: LU decomposition - Cilk, http://courses.cs.tau.ac.il/368-4064/cilk-5.3.1/examples/lu.cilk
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. In: Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. pp. 207–216. PPOPP '95, ACM, New York, NY, USA (1995)
3. Cavé, V., Zhao, J., Guo, Y., Sarkar, V.: Habanero-Java: the New Adventures of Old X10. In: PPPJ. pp. 51–61 (2011)
4. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
5. 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. SIGPLAN Not. 40, 519–538 (Oct 2005)
6. Drago, I., Cunei, A., Vitek, J.: Continuations in the Java Virtual Machine. In: International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (2007)
7. EPCC: The Java Grande Forum Multi-threaded Benchmarks, http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html
8. Felleisen, M.: The Theory and Practice of First-Class Prompts. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 180–190. POPL '88, ACM, New York, NY, USA (1988)
9. Fischer, J., Majumdar, R., Millstein, T.: Tasks: Language Support for Event-driven Programming. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 134–143. PEPM '07, ACM, New York, NY, USA (2007)
10. Fluet, M., Rainey, M., Reppy, J., Shaw, A.: Implicitly Threaded Parallelism in Manticore. J. Funct. Program. 20(5-6) (Nov 2010)
11. Fulgham, B.: binary-trees benchmark, http://benchmarksgame.alioth.debian.org/u32/performance.php?test=binarytrees
12. Fulgham, B.: n-body benchmark, http://benchmarksgame.alioth.debian.org/u32/performance.php?test=nbody
13. Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. pp. 57–76. OOPSLA '07, ACM, New York, NY, USA (2007)
14. Gray, J.: Writing Faster Managed Code: Know What Things Cost, http://msdn.microsoft.com/en-us/library/ms973852.aspx
15. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. pp. 1–12. IPDPS '09, IEEE Computer Society, Washington, DC, USA (2009)

16. Gupta, S., Nandivada, V.K.: IMSuite: A Benchmark Suite for Simulating Distributed Algorithms. CoRR abs/1310.2814 (2013)
17. Halstead, R.H.: Multilisp: A Language for Concurrent Symbolic Computation. ACM Transactions on Programming Languages and Systems 7, 501–538 (October 1985)
18. Haynes, C.T., Friedman, D.P.: Engines Build Process Abstractions. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming. pp. 18–24. LFP '84, ACM, New York, NY, USA (1984)
19. Herzeel, C., Costanza, P.: Dynamic Parallelization of Recursive Code Part I: Managing Control Flow Interactions with the Continuator. In: Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications. pp. 377–396. OOPSLA '10, ACM, New York, NY, USA (2010)
20. Imam, S., Sarkar, V.: Integrating Task Parallelism with Actors. In: Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications. pp. 753–772. OOPSLA '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2384616.2384671
21. Imam, S., Sarkar, V.: A Case for Cooperative Scheduling in X10's Managed Runtime. In: The 2014 X10 Workshop (X10'14) (June 2014)
22. Lea, D.: A Java Fork/Join Framework. In: Java Grande. pp. 36–43 (2000)
23. Li, P., Marlow, S., Peyton Jones, S., Tolmach, A.: Lightweight Concurrency Primitives for GHC. In: Proceedings of the ACM SIGPLAN Haskell Workshop. pp. 107–118. Haskell '07, ACM, New York, NY, USA (2007)
24. OpenMP Application Program Interface, Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf (May 2008)
25. Reinders, J.: Intel Threading Building Blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)
26. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In: Proceedings of the 22nd Annual International Conference on Supercomputing. pp. 277–288. ICS '08, ACM, New York, NY, USA (2008)
27. Sigoure, B.: How long does it take to make a context switch, http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
28. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Proceedings of the 22nd European conference on Object-Oriented Programming. pp. 104–128. ECOOP '08, Springer-Verlag, Berlin, Heidelberg (2008)
29. Tardieu, O., Wang, H., Lin, H.: A Work-Stealing Scheduler for X10s Task Parallelism with Suspension. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. pp. 267–276. PPoPP '12, ACM, New York, NY, USA (2012)
30. Wheeler, K., Murphy, R., Thain, D.: Qthreads: An API for programming with millions of lightweight threads. In: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. pp. 1–8 (2008)
31. Yan, Y., Chatterjee, S., Budimlić, Z., Sarkar, V.: Integrating MPI with Asynchronous Task Parallelism. In: Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science, vol. 6960, pp. 333–336. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-24449-0_41