Data-Driven Tasks and their Implementation

Sağnak Taşırlar, Vivek Sarkar

Department of Computer Science
Rice University
6100 Main Street, Houston TX 77025
{sagnak,vsarkar}@rice.edu

***Abstract*—Dynamic task parallelism has been identified as a prerequisite for improving productivity and performance on future many-core processors. In dynamic task parallelism, computations are created dynamically and the runtime scheduler is responsible for scheduling the computations across processor cores. The sets of *task graphs* that can be supported by a dynamic scheduler depend on the underlying task primitives in the parallel programming model, with various classes of *fork-join* structures used most often in practice. However, many researchers have advocated the benefits of more general task graph structures, and have shown that the use of these task graph structures can lead to improved performance.**

In this paper, we propose an extension to task parallelism called Data-Driven Tasks (DDTs) that can be used to create arbitrary task graph structures. Unlike a normal task that starts execution upon creation, a DDT specifies its input constraints in an `await` clause containing a list of Data-Driven Futures (DDFs). A DDF can be viewed as a container with a *full/empty* state that obeys a dynamic single-assignment rule. The runtime scheduler will then ensure that a task is only scheduled when all the DDFs in its `await` clause become available (*full*). There is no constraint on which task performs a `put()` operation on a DDF.

We describe five scheduling algorithms (Coarse-Grain Blocking, Fine-Grain Blocking, Delayed Async, Rollback & Replay, Data-Driven) that can be used to implement DDTs, and include performance evaluations of these five algorithms on a variety of benchmark programs and multi-core platforms. Our results show that the Data-Driven scheduler is the best approach for implementing DDTs, both from the viewpoints of memory efficiency and scalable parallelism.

I. Introduction

The computer industry is at a major inflection point due to the end of a decades-long trend of exponentially increasing clock frequencies. It is widely agreed that parallelism in the form of multiple power-efficient cores must be exploited to compensate for this lack of frequency scaling. Unlike previous generations of hardware evolution, this shift towards homogeneous and heterogeneous many-core computing will have a profound impact on software. The three programming languages developed as part of the DARPA HPCS program (Chapel [1], Fortress [2], X10 [3]) all identified dynamic task parallelism as a prerequisite for improving productivity and performance on future many-core processors. Dynamic task parallelism has also been introduced as extensions to existing languages such as the OpenMP 3.0 [4] and Cilk [5] extensions to C, and the Java Concurrency [6] and Habanero Java (HJ) [7] extensions to Java.

In dynamic task parallelism, computations are created dynamically and the runtime scheduler is responsible for scheduling the computations across processor cores. Dynamic scheduling is expected to be even more important in future processors, as power management considerations lead to increases in non-uniformity and asymmetry across cores. The sets of *task graphs* that can be supported by a dynamic scheduler depend on the underlying task primitives in the parallel programming model e.g., the task graphs created by Cilk's `spawn` and `sync` primitives are restricted to *fully-strict* computation graphs [8], whereas the task graphs created by `async` and `finish` constructs in X10 and HJ belong to the set of *terminally-strict* computation graphs [9]. A number of researchers have advocated the benefits of more general task graph structures [10], [11], [12], and shown that the use of these task graph structures can lead to improved performance.

In this paper, we propose an extension to task parallelism called Data-Driven Tasks (DDTs) that can be used to create arbitrary task graph structures. Unlike a normal task that starts execution upon creation, a DDT specifies its input constraints in an `await` clause containing a list of Data-Driven Futures (DDFs). A DDF can be viewed as a container with a *full/empty* state. A `put()` operation changes the state of a DDF from *empty* to *full*, and at most one `put()` operation can be performed on a DDF because of the dynamic single-assignment rule. The runtime scheduler will then ensure that a task is only scheduled when all the DDFs in its `await` clause become available (*full*). It is illegal to perform a `get()` operation on an empty DDF; instead, a task can safely perform a `get()` on any DDF in its `await` clause. There is no constraint on which task may perform a `put()` operation on a DDF. A `finish` construct can be used to await completion of a set of `async` task instances as usual, thereby allowing DDTs to be used freely in conjunction with regular `async-finish` task parallelism.

The main contributions of this paper are as follows:

- Definition of Data-Driven Tasks (DDTs), a new extension to task parallelism.
- Specification of five scheduling algorithms (Coarse-Grain Blocking, Fine-Grain Blocking, Delayed Async, Rollback & Replay, Data-Driven) that can be used to implement DDTs.
- A performance evaluation of these five algorithms on a variety of benchmark programs and multicore platforms

that demonstrate that the Data-Driven scheduler is indeed the best approach for implementing DDTs.

While the DDT construct is inspired by past work on dataflow programming models, we are unaware of any past work that integrates dataflow principles with task parallelism as proposed in DDTs or that evaluates the scheduling algorithms considered in our paper.

The rest of the paper is organized as follows. In section II, we introduce DDTs. We discuss a code sample in section III. Section IV describes a Data-Driven runtime scheduler for DDTs. We discuss other scheduling policies from previous work in Section V. Empirical results for our implementation are presented in Section VI. Related work is covered in section VII and, finally Section VIII contains our conclusions. A more detailed version of this work is available in this Master's thesis [13].

II. Overview

A Data-Driven Task (DDT) is a task that synchronizes with other tasks through synchronization constructs named Data-Driven Futures (DDFs). A DDF can be viewed a container with a *full/empty* state that obeys the dynamic single-assignment rule. Therefore, all data accesses performed via DDFs are guaranteed to be race-free and deterministic. In this paper, we define DDTs in the context of Habanero Java but they could just as easily be integrated in other task-parallel languages such as OpenMP 3.0 [4] and Cilk [5].

Our proposed language interface for DDFs is as follows:

Read: `get()` is a non-blocking interface for reading the value of a DDF. If the DDF has already been provided a value via a `put()` operation, a `get()` delivers that value. However, if the producer task has not yet performed its `put()` at the time of the `get()` invocation, the `get()` throws an exception.

Write: `put()` is the interface for writing the value of a DDF. Since a DDF is a reference to a single-assignment value, only one producer may set its value and any successive attempt at setting the value results in an exception.

Creation: `new()` is the interface for creating a DDF object. The producer and consumer tasks use DDF references to perform `put()` and `get()` operations.

Registration: the `await` clause associates a DDT with a set of input DDFs. The syntax we proposed for this registration is as follows: `async await` $(DDF_a, DDF_b, \cdots)$ $\langle Stmt \rangle$.

The sample code snippet in Figure 1 shows the creation of five tasks and the specification of their synchronization pattern through DDFs. Initially, two DDFs are created as containers for data items `left` and `right`. Then a `finish` scope is created with five `async` tasks as in Habanero Java or X10. Task_3, Task_4 and Task_5 use `await` clauses to specify their dependences. The methods suffixed `Reader`, are passed references to perform a `get()` on the DDF instances that they receive.

For instance, Task_4 registers itself on both `left` and `right` DDFs, which establishes data dependences on the first two `async`s that are the producers for those DDFs. Regardless of the underlying scheduler, Task_1 and Task_2 are guaranteed to complete execution before Task_4. This ability for a DDT to wait on two (or more) DDFs before starting is unique to our DDT model, and was not supported by past work such as blocking reads in I-structures [14].

```
// Create two DDFs
DataDrivenFuture left  = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish { // begin parallel region
    async left.Put(leftBuilder());//Task1
    async right.Put(rightBuilder());//Task2
    async await(left) leftReader(left);//Task3
    async await(left, right) {//Task4
        bothReader(left, right);
    }
    async await(right) rightReader(right);//Task5
}  // end parallel region
```

Fig. 1. A Habanero Java code snippet with Data-Driven Tasks and Data-Driven Futures.

In contrast to the future construct introduced in [15] as a 3-tuple of the form *(resolvingProcess, storage, waitingTasks)*, the control and the data aspects of a future are split into DDTs and DDFs in our approach. A DDF can be viewed as a 2-tuple of *(storage, waitingTasks)*, whereas the DDT that performs a `put()` operation on the DDF is the *resolvingProcess*.

A. Discussion

Most task-parallel programming languages enforce fork-join orderings that *unify* control and data dependences. We use the term *control dependence* to express the relationship between a *parent* task and a *child* task, caused by the creation of that child task; each child task thus has a unique *control parent*. A *data parent* of a task is a task that provides data to it, and thus a task can have more than one data parent. In most models, when a task (*control parent*) creates a child computation, it also provides the data the child computation needs. This burdens the programmer by requiring them to ensure that all the data parents of a child task have completed execution before the child task is created. However, in general, the creator of a task need not be the same task that computes the data needed by that child computation i.e., a task's control and data parents may be distinct.

DDFs provide a first-class construct for expressing data-dependences among dynamic tasks, thereby decoupling the roles of the control parent and data parents. Figure 2 demonstrates the expressive power of DDTs by showing how DDFs can be used to synchronize DDTs in a reverse order relative to their creation. DDTs and DDFs can be used to build arbitrary task graphs without the need to couple data dependences with control dependences in any way.

Nested fork-join models are restricted to creating series-parallel task graphs. For example, looking back at Figure 1, we see that the dependences among those five tasks cannot be described by standard nested fork-join constructs without constraining parallelism. The dependence graph for the tasks in Figure 1 is shown on the left side of Figure 3. An

```
DataDrivenFuture[] A = new DataDrivenFuture[n];
for ( int i = 0 ; i < N; ++i ) {
    A[i] = new DataDrivenFuture();
}
for ( int i = N-1 ; i >= 1; --i ) {
    async await(A[i-1]) {
        ...
        A[i].put("");
    }
}
A[0].put("");
```

Fig. 2. An example in which DDFs are used to synchronize DDTs in the reverse order relative to their creation

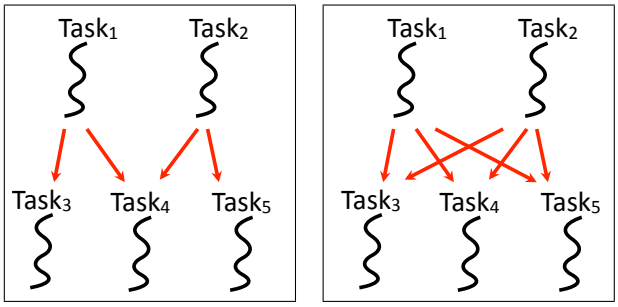

Fig. 3. Dependence graphs for DDT program in Figure 1 (left) fork/join equivalent of the same program (right)

alternate solution in a nested fork-join model would be to hoist $Task_1$ and $Task_2$ to the parent task, thereby creating an implicit barrier between producers and consumers. That approach would result in the dependence graph shown on the right side of Figure 3, which has less parallelism than the graph on the left side. Additionally, the lack of a construct like DDTs burdens the programmer with thinking about the creation of fork-join structures that satisfy a given set of data dependences [16].

Though the DDF construct provides a natural way to enforce data dependences, it can also be used to enforce control dependences. A control dependence can be expressed with a DDF with dummy data. For example, we have used a null string to enable DDTs in Figure 2.

A drawback of hoisting producer tasks, as in Figure 3, is that it not only constrains parallelism but also increases the memory footprint. The hoisting of tasks unnecessarily extends the lifetime of produced data by creating them earlier. However, with DDTs, the creation of DDFs can be delayed as needed thereby shortening their lifetimes.

As with other `async` tasks, DDTs are also registered to their *immediately enclosing finish* (IEF). If all tasks in the same `finish` scope have completed or reached a quiescent state waiting for their `await` clauses to be satisfied, the `finish` scope can exit by discarding DDTs whose dependences are unsatisfied. This is akin to garbage collection of unrequited futures as in [15].

DDTs and DDFs can also be used to implement standard *future* constructs [15], [17]. Consider the Habanero Java `future` construct in Figure 4 as an example. At creation time, an `async` task is bound to the computation of the `future`'s

```
...
final future<int> f = async<int> { return g(); };
...
p.x = f.get();
...
```

Fig. 4. Habanero Java interface for the language construct `future`

value, When the value has to be resolved, a `get()` blocks until the value becomes available. Now, we will show how we can use DDTs and Data-Driven Futures to express the same constraints.

```
...
DataDrivenFuture f = new DataDrivenFuture();
async { f.put(g()); };
...
async await (f) { p.x = f.get(); };
...
```

Fig. 5. Data-Driven Task and Data-Driven Future equivalent of Figure 4

Looking at Figure 5, we see that the proxy object `f` does not have its producer task bound at definition time. Afterwards, a logically parallel `async` task produces the value that `f` refers to. The blocking `get()` call to ensure safety in Figure 4 is now replaced by the non-blocking `await` clause in Figure 5. Thus, the execution of that DDT will be delayed until `f` is provided.

III. Code Sample

In this section, we walk through a code snippet in Figure 6 that is the main computation of the Heart Wall Tracking benchmark [18]. The dependence graph for two iterations over the *j* loop is shown in Figure 7.

The outermost loop, over `numPoints` in line 2, spawns a computation for each pixel in a 2D image using an `async`. As our input is a video, there is also a time dimension, declared over `numFrames` on line 7 for this example. The computation for each pixel has a loop-carried dependence on the *j* loop. We use `nextFrame` to denote the next loop iteration produced by `step10` (line 26).

As we enter the loop over the variable *j*, we start by creating the Data-Driven Futures (lines 9, 12–15) which will be used in `await` clauses of Data-Driven Task declarations (lines 11, 18–23, 26). We then create ten DDTs that are synchronized through those DDFs. If a DDT has no incoming dependence, it can be declared as an `async` without an `await` clause. The convention followed in these declarations is that the last parameter of a DDT's `compute` method produces its output DDF. In this example, each DDT produces exactly one DDF. The entire code in Figure 6 is enclosed in a single `finish` construct.

IV. Runtime Scheduling of Data-Driven Tasks

In this section, we introduce a data-driven runtime scheduler for DDTs that follows asynchronous dataflow semantics. Following the dataflow principle, tasks are only spawned when their input DDFs become available. It is the availability of data

```
finish { //Start a global synchronization scope
  for( int i = 0 ; i < numPoints; ++i ) { //For each pixel on a video
     async { //spawn a computation with loop-carried dependences over the frame loop
       //Create a synchronizer for the first frame
       DataDrivenFuture prevFrame = new DataDrivenFuture();
       prevFrame.Put(new java.lang.Boolean(true)); //Resolve the synchronizer
       for( int j = 0 ; j < numFrames; ++j ) { //for each frame
          //Create a synchronizer for the next frame
          DataDrivenFuture nextFrame = new DataDrivenFuture();
          //When previous frame is produced, start an arbitrary task graph as follows
          async await(prevFrame) {
            DataDrivenFuture[] signals = new DataDrivenFuture[9];
            for (int k = 0; k < 9; ++k) {
              signals[k] = new DataDrivenFuture();
            } // for
            async { step1.compute([i,j], signals[0]); }
            async { step2.compute([i,j], signals[1]); }
            async await(signals[1]) { step3.compute([i,j], signals[2]); }
            async await(signals[0], signals[1]) { step4.compute([i,j], signals[3]); }
            async await(signals[0]) { step5.compute([i,j], signals[4]); }
            async await(signals[2],signals[3],signals[4]){step6.compute([i,j],signals[5]);}
            async await(signals[4]) { step7.compute([i,j], signals[6]); }
            async await(signals[5],signals[6]){ step8.compute([i,j], signals[7]);}
            async { step9.compute([i,j], signals[8]); }
            //Spawn step10 that produces nextFrame when signal 7 and 8 are produced
            async await(signals[7],signals[8]){step10.compute([i,j],numFrames,nextFrame);}
          }
          //production of nextFrame via step 10 described the loop-carried dependence
          prevFrame = nextFrame;
       }
     }
  }
}
```

Fig. 6. Code snippet from Heart Wall Tracking benchmark using Data-Driven Tasks and Data-Driven Futures

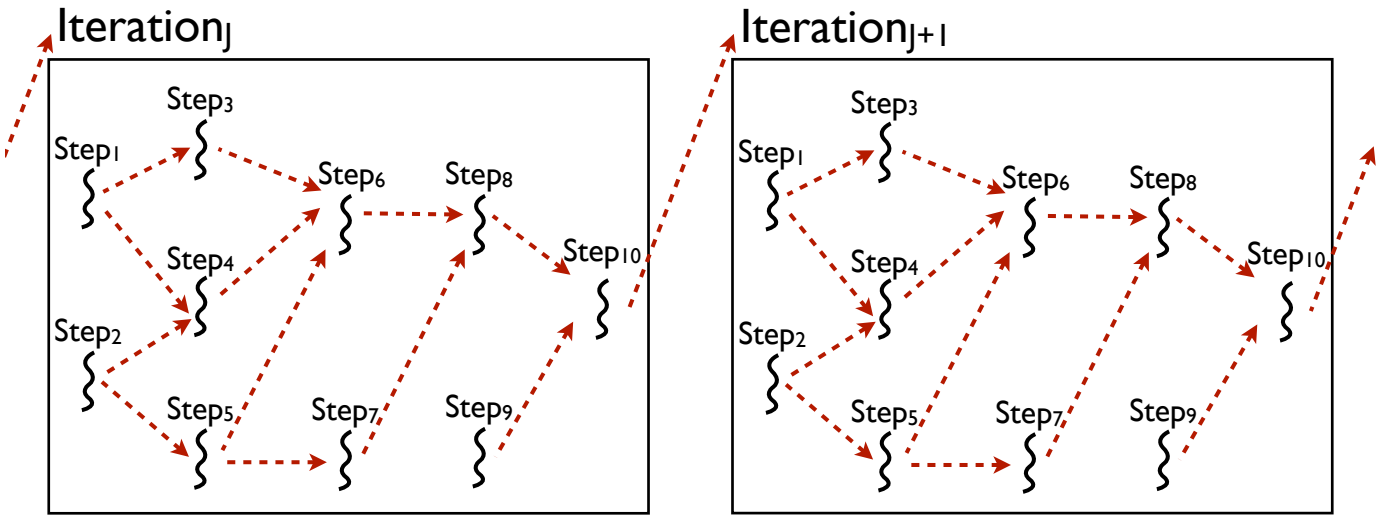

Fig. 7. Unified control and data dependence graph of Heart Wall Tracking

that enables them for execution. Consequently, when a DDF becomes resolved, each DDT registered as its consumer could become ready, if it is the only DDF that the DDT is waiting for. If there are multiple input DDFs for a DDT, one can think of the last DDF to be produced, as the enabling one.

We have implemented a data-driven runtime scheduler for DDTs, whose implementation details are discussed below and performance results are discussed in later section VI. Our implementation of the data-driven runtime scheduler is based on Habanero Java [7] and its work sharing runtime [19], [9]. Accordingly, within this section's scope, it is appropriate to read a *task* as a Habanero Java `async`, an *object* as a Java object and a *list* as linked list of Java objects.

Data-Driven Futures are implemented as objects that hold a single-assignment value and a list of registered data-driven tasks that are consumers of this value. In general, the value will be assigned at runtime by a producer task. Since the value held within a DDF is single-assignment, any attempt to reassign the value results in an exception.

Each DDT holds a list of input DDFs that it is designated to consume. This list is populated during the creation of a DDT at runtime. Readiness of a task can then be checked any time by a traversal over the list of DDFs. Since the readiness of a single DDF monotonically increases from *empty* to *full*, readiness is a stable property. Once a DDF is found to be ready, we can stop checking for its readiness. The state of a DDF list includes a pointer (dashed arrow) to its first non-ready DDF.

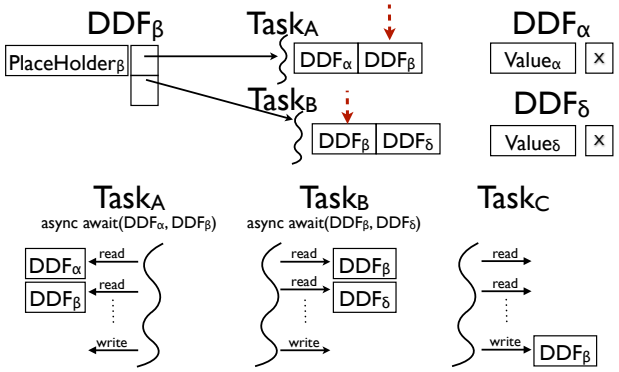

Fig. 8. Snapshot of a subset of Data-Driven Tasks and Data-Driven Future during runtime

We can see a snapshot of some sample DDTs and DDFs in Figure 8. This figure shows the data dependence relationships among data-driven tasks A, B and C through the DDFs α, β and δ. Here are some conclusions we can derive from this snapshot. First of all, we know Task_A consumes data items in DDF_α and DDF_β, whereas Task_B consumes data items in DDF_β and DDF_δ. The task designated as the producer for DDF_β is Task_C. Some producers have already provided the values for DDF_α and DDF_δ. From the upper left corner of the figure, we can see that a DDF has a list of DDTs, which are its consumers and DDTs have a list of DDFs they consume. At the time of this snapshot, Task_A has already passed over DDF_α since the value has been produced. However in Task_B's case, even though DDF_δ is ready, the task is not aware of that fact as it is waiting on DDF_β. In this scenario, let us assume that the very next action is the execution of Task_C, which will lead to the assignment of the value DDF_β. The assignment of that value will induce the traversal of the designated consumer task list. On each task, the waiting frontier, shown by a vertical dashed arrow, is iterated, which is an asynchronous way of moving each task to its next phase when the appropriate `put()` is performed. Once all DDFs a task waits on are provided, that task is deemed ready to execute.

We conform to the semantics for data-driven tasks and data-driven futures described in section II with the following Habanero Java implementation of DDTs and DDFs:

`get()`*:* is implemented as described in section II. If an incorrect program attempts to access a data-driven future that has not been resolved, an exception is thrown.

`put()`*:* A DDF being written may be polled during the write by another thread for readiness, so writing is synchronized. Once the write is complete, `put()` exits the critical region to advance the iterators indicating where data-driven tasks have been waiting, for all the data-driven tasks registered themselves as consumers. Any data-driven task created from then on will observe the DDF to be ready.

Creation: instantiates a DDF object that is a container for the single-assignment value that DDF is a reference to and initializes the list of DDTs registered as consumers to an empty list.

Registration: is supported by the `async await` clause. We require DDTs to declare the all the DDFs they may perform a `get()` on. The creation of a DDT also initializes the list of DDFs that DDT consumes. If all DDFs are resolved prior to the creation of the task, then the task is ready to execute.

V. Task Scheduling Policies

For the four scheduling policies from past work, we implemented synchronization through *collections* of Data-Driven Futures that are globally accessible to all Data-Driven Tasks. It should be noted that these schedulers are implemented for a macro-dataflow parallel programming model, Concurrent Collections [20], where one of its implementations is on top of Habanero Java. The globally accessible *collections* of data is inherent to that model in order to achieve synchronization between producer and consumer tasks. As these *collections* of data are implemented as Java concurrency utilities hashmaps indexed by tags, there is a cost of hash computation for each access. A tuple *(collection$_i$,tag$_j$)* in these four schedulers is analogous to a Data-Driven Future instance in the Data-Driven scheduler. Since collections of DDFs are not amenable to garbage collection, the previous schedulers have the drawback of extending the lifetime of data.

There is also an implicit assumption for the Rollback & Replay scheduler that Data-Driven Tasks that are synchronized through Data-Driven Futures are side-effect free and functional with respect to their inputs to guarantee that their replay is safe.

A. Previous Proposals

Following past work [20], we implemented four possible scheduling policies used in dataflow programs.

Coarse Grain Blocking Scheduler: This scheduler optimistically assumes computations are ready to execute when the `async` is created. However if a task attempts to read a datum that has not been provided by a producer, the task attempting the read blocks on the entire collection harboring that data. The blocked task will be notified each time a data item is added to the collection and unblock when the data items needed becomes available.

Fine Grain Blocking Scheduler: This scheduler is similar to the coarse grain blocking scheduler. However, a task attempting a read an uninitialized data item blocks on the item's DDF rather than the collection. Therefore that task is notified only when that data item is provided.

Delayed Async Scheduler: uses a guarded execution construct, *delayed async*, supported in Habanero Java [21]. This guard is re-executed each time some task completes until it evaluates to true which promotes a delayed async to an `async` that is ready to execute. This scheduler wraps each DDT that needs to synchronize on data into a delayed async, where the guard computation checks if all the input data items are available.

Rollback & Replay Scheduler: This scheduler also optimistically assumes all tasks are ready on creation. It, however, discards a scheduled computation instance if the task attempts to read an uninitialized data item. These "rolled-back" tasks are saved as closures with the uninitialized datum they tried to access. Once the datum is provided, the scheduler checks if it has any associated closures. If so, it "replays" those tasks to be scheduled again. This approach assumes that tasks are side effect free and functional with respect to their inputs so they can be safely re-executed.

Coarse grain blocking scheduling blocks computation instances whose data are not ready. Since a task that blocks on data will block the whole thread harboring that task in a work-sharing runtime, we need to introduce new threads to ensure a progress guarantee. This increases the memory footprint as each blocked task allocates a thread. Additionally, a blocked task is unnecessarily awakened on each data production on that item collection, even if that is not the data items that the task is waiting on.

Fine grain blocking scheduling ameliorates this problem by blocking threads on items rather than collections. However it still does not address the memory concerns introduced by blocking threads.

In delayed async scheduling, since no task is scheduled before it is guaranteed to complete, no threads block on this scheduler and therefore there is no need for creation of new threads. However, the continuous re-evaluation of guards is an overhead introduced in this scheme.

Rollback and replay scheduling ameliorates the two problems discussed above. Since the task eagerly executes, fails and executes again lazily when the prematurely read data is ready, there is no busy wait on the data. However, one downside to this approach is the possibility of eager execution and eager replay. Assume that n reads to data items is performed by a task. In the worst case, it is possible that a computation gets replayed n times.

B. Data-Driven Scheduling with Data-Driven Tasks

Given a program, each task instance can be interpreted as a DDT that reads DDFs as inputs. So if a DDT registers on all its input DDFs, we can use the synchronization and scheduling mechanism described in section IV by describing tasks as `async`s with `await` clauses. A normal `async` is simply assumed to have an empty DDF list.

One may see the natural progression when comparing DDTs with the previous proposals discussed above. This scheduling scheme does not eagerly execute tasks when they are encountered. It instead declares all the registrations on input DDFs at a task's creation. Therefore Data-Driven Scheduling avoids the need to fail repetitively, which rollback and replay scheduling has to do in order to learn data dependences lazily at runtime. As this scheduler does not work with collections of DDFs but rather with instances, it is amenable to garbage collection.

We postulate that the schedulers should perform better in the order they are introduced and data-driven scheduling with DDTs should outperform all.

VI. Experimental Results

We have collected experimental results to show how benchmarks implemented with DDTs perform with the four scheduling policies from previous work, as well as the data-driven policy introduced in this paper.

As we have noted before, the previous schedulers are implemented for Concurrent Collections. The benchmarks used below for the previous schedulers are Concurrent Collections implementations of the benchmarks. In order to obtain results on data-driven scheduling for DDTs, we have written the DDT equivalent of the same benchmarks while retaining the structure of the CnC code.

A. Experimental Method

All the tests presented below report the mean running times of 30 runs of each benchmark from a single JVM invocation to reduce biases from cache effects and effects of just-in-time compilation. Heart Wall Tracking is the exception to the 30 test runs, for which we used 13 runs[1]. We were influenced by [22] in our choice of 30 runs of a program in an invocation and confidence intervals for the mean. The 90% confidence intervals for the means are represented as error bars in the charts in Figures 10–16.

We have dubbed the single worker execution of our parallel benchmarks as the base case for our speedups in Figure 9, where we use the DDT implementation under Data-Driven scheduling as the baseline.

The systems we have collected performance results are:

[1]Due to the use of JNI in Heart Wall Tracking implementation, it can only be run 13 times instead of 30 times.

- **Xeon** contains four quad-core Intel E7730 processors running at 2.4 GHz. Each pair of cores share a L2 cache of size 3MB. The total amount of main memory for this machine is 32 GBs. For our tests on this machine we set the number of workers to be 16, i.e. one worker per core.
- **Niagara** has a Sun UltraSPARC T2 microprocessor with 8 cores, and supports concurrent execution of 8 threads per core. There is only one L2 cache of size 4MB that is shared among all these cores. For our tests on this machine we set the number of workers to be 64, i.e. one worker per hardware thread.

B. Benchmarks

We obtained results for the following benchmarks:

Cholesky decomposition is a dense linear algebra application that exploits loop, data, task and pipeline parallelism. Exploitation of pipeline parallelism helps with performance and scaling as the kernels of this application from different iterations have data dependences in between, namely loop-carried dependences. With a model like nested fork-join it is not possible to achieve a pipelined execution without over-constraining parallelism as in the right side of Figure 3. As shown in [12], a macro-dataflow implementation can match and even outperform state of the art task-parallel parallel programming models and tuned domain specific libraries. That work did not address data-driven task scheduling, which is the focus of this paper.

Black-Scholes is a benchmark adopted from the PARSEC [23] benchmark suite. This benchmark incurs lower scheduling overhead than others.

Heart Wall Tracking is a medical image processing application from the Rodinia benchmark suite [18], that tracks a beating heart video by applying filters to the frames of that video. The application has an outer serial loop with iterations including data parallel computations, where the set of tasks applied to the data has the dependence graph denoted in Figure 7. As can be observed, this graph can not easily be mapped to structured parallel programming models, but the graph can be easily expressed using DDTs.

Rician Denoising [24] is a convergent benchmark, where each iteration applies various computations to an image and some of these steps are stencil computations. The benchmark is memory and bandwidth bound. The premise for the DDT model includes a dynamic single-assignment property, which makes it not a very natural fit for convergent benchmarks as discussed below.

C. Results

Figure 9 shows the speedup results for Data-Driven scheduling, with respect to the single worker execution of the Habanero Java code with DDTs. Minimum of 30 runs in a single JVM invocation are taken into account to calculate the speedup results.

The Heart Wall Tracking benchmark has nested parallelism where the outermost loop is a parallel loop and there also is task parallelism to be exploited in the inner loop. In

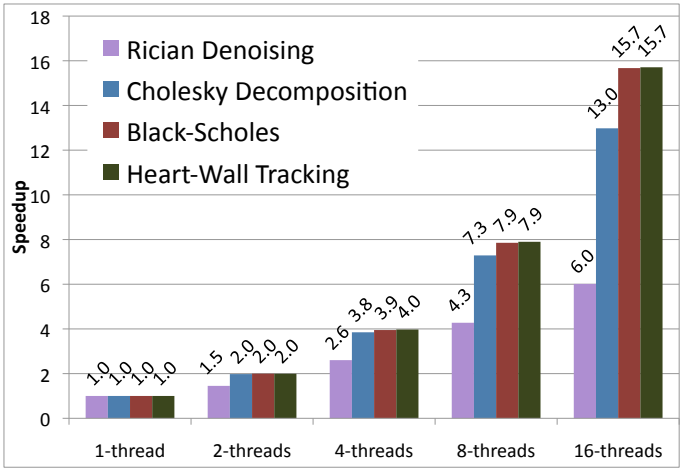

Fig. 9. Speedups for Data-Driven scheduling with respect to single worker execution of the parallel code written in Habanero Java with DDTs. Minimum execution times of 30 runs on a 4-socket 4-way SMP Xeon core are taken into account.

Figure 9, we observe that even though we implemented the benchmark by exposing the fine grain task parallelism within loop iterations, which introduces extra synchronization that the original implementation did not incur, we managed to observe almost perfect scaling on a 16-core Xeon using DDTs.

For Black-Scholes, a data parallel benchmark where there are no inter-task control or data dependences, we observe that the synchronization cost of using a DDT is negligible.

The Cholesky factorization benchmark implemented with DDTs shows almost perfect speedups for 2 and 4 workers. For the 8-worker case, the speedup observed is 7.29 and on 16-worker case, the speedup is 12.98, which can be explained by the non-trivial pipeline parallelism that has to be exploited on a benchmark like Cholesky. We have noted in section VI-B that the fastest known parallel implementation of Cholesky also uses the macro-dataflow parallel programming model [12]. We implemented the equivalent of the program using DDT synchronization instead and we will see in the upcoming figures that the data-driven scheduler outperforms the other schedulers.

Rician Denoising does not have great speedup numbers because the benchmark suffers from heavy memory use and is bandwidth limited. The more data introduced to evaluate this benchmark to provide more wavefront parallelism to the stencil computation kernels, the more frequently the garbage collector has to run. So the parallelism is hampered because of the applications memory footprint but we will see later that the data-driven implementation of this benchmark still outperforms the other schedulers.

The remainder of this section provides comparisons of the benchmarks implemented for DDTs with the four schedulers from past work described earlier. The scheduling schemes on the x-axis of the figures below refer to the scheduling policies described in Section V. We postulated in that section that the schedulers should perform better in the order they

are introduced and data driven scheduling with DDTs should outperform all. Below are the results to validate that claim.

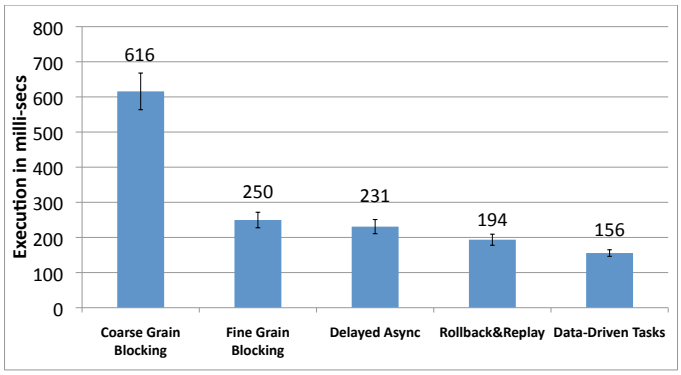

Fig. 10. Average execution times and 90% confidence interval of 30 runs of 16-worker executions for blocked Cholesky factorization application with Habanero Java and Intel MKL steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

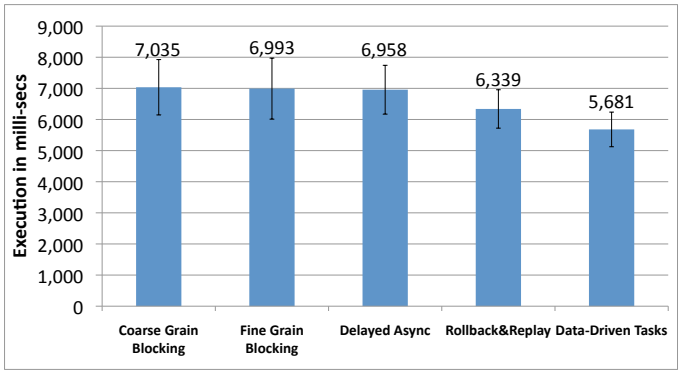

Fig. 11. Average execution times and 90% confidence interval of 30 runs of 64-worker executions for blocked Cholesky factorization application with Habanero-Java and Intel MKL steps on UltraSPARC T2 with input matrix size 2000 × 2000 and with tile size 125 × 125

In Figures 10 and 11, we see how a Cholesky decomposition algorithm from [12] performs on a 16-core Xeon and a 64-thread parallel UltraSPARC T2 with 16 and 64 workers, respectively. As can be seen, the Data-Driven scheduler outperform all known schedulers. Also, coarse grain blocking was relatively worse on Xeon than on the UltraSPARC. These results were obtained using a sequential Intel MKL call in each step for the Intel architecture implementation (Xeon).

Figures 12 and 13 show performance results for Black-Scholes. We see again that the data-driven scheduling outperforms the others. We also observe that the confidence intervals for Rollback and Replay is wide, as Rollback and Replay pays for safety even when tasks are guaranteed not to replay. That cost for safety usually is amortized by replayed tasks, however this benchmark shows doing so may introduce unnecessary overhead.

Rician Denoising is a convergent fixed point algorithm where the convergence check imposes a barrier at the end of each iteration. However, as discussed before, the memory

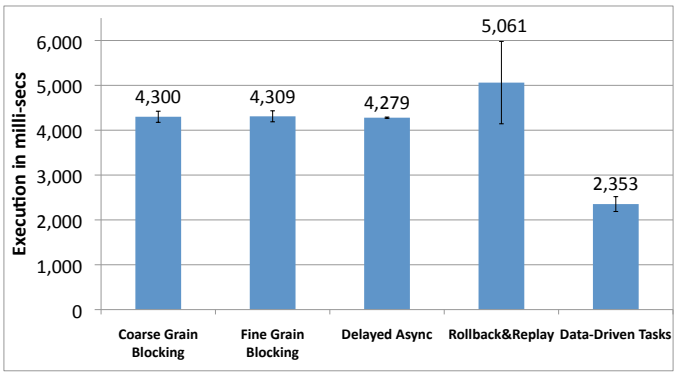

Fig. 12. Average execution times and 90% confidence interval of 30 runs of 16-worker executions for blocked Black-Scholes application with Habanero Java steps on Xeon with input size 1,000,000 and with tile size 62500

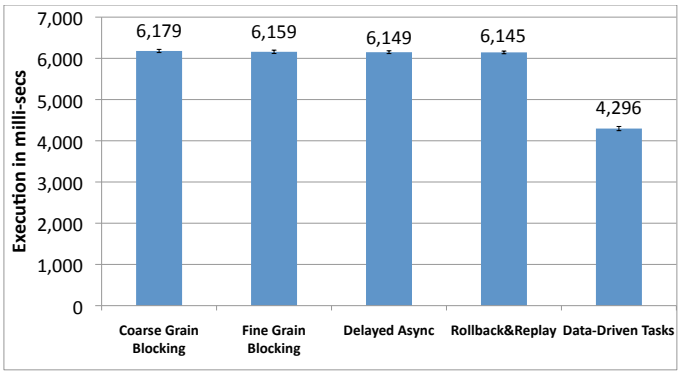

Fig. 13. Average execution times and 90% confidence interval of 30 runs of 64-worker executions for blocked Black-Scholes application with Habanero Java steps on UltraSPARC T2 with input size 1,000,000 and with tile size 15625

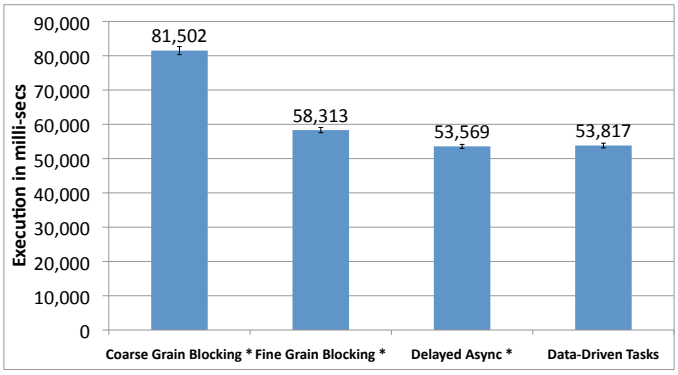

Fig. 14. Average execution times and 90% confidence interval of 30 runs of 16-worker executions for blocked Rician Denoising application with Habanero Java steps on Xeon with input image size 2937 × 3872 pixels and with tile size 267 × 484 (Scheduling algorithms with * required explicit memory management by the programmer to avoid running out of memory)

footprint of this program can grow without bounds since the dynamic single-assignment property in our model leaves each iteration's data alive throughout the computations, which depleted the machine's memory for the four schedulers from past work. So, for our implementation, we have bent the

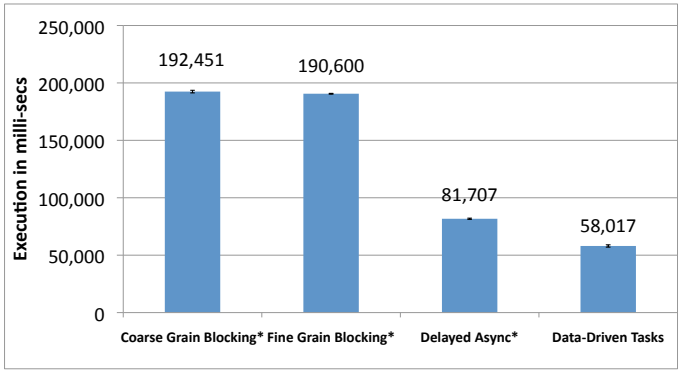

Fig. 15. Average execution times and 90% confidence interval of 30 runs of 16-worker executions for blocked Rician Denoising application with Habanero Java steps on UltraSPARC T2 with input image size 2937 × 3872 pixels and with tile size 267 × 484 (Scheduling algorithms with * required explicit memory management by the programmer to avoid running out of memory)

rules by introducing an interface to explicitly reclaim memory for those schedulers. Schedulers that needed explicit memory management are annotated with a * in Figure 14 and 15. Rollback & Replay scheduling can not be intertwined with explicit memory management and therefore results for that scheduler is lacking from these two charts. It should be noted that data-driven scheduling needed no explicit memory management since the garbage collector automatically frees the DDFs that are no longer needed.

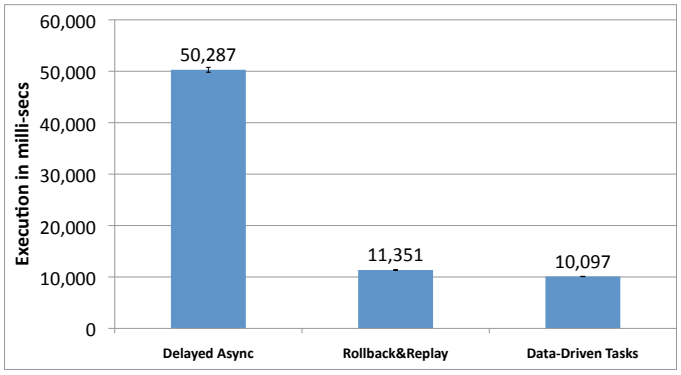

Fig. 16. Average execution times and 90% confidence interval of 13 runs of 16-worker executions for Heart Wall Tracking application with C steps on Xeon with 104 frames

As mentioned earlier, Heart Wall Tracking has an outermost data parallel computation with an intricate task dependence graph for each sub-task. Blocking schedulers for this benchmark are not feasible as the sub-tasks are too fine grained and the blocking schedulers causes the execution to run out of memory with too many blocked threads. In this benchmark, we observe that data-driven scheduling outperforms the other two non-blocking schedulers (Figure 16). We have not as yet obtained results for Heart Wall Tracking on Niagara.

VII. Related Work

While the DDT construct is inspired by past work on the *dataflow* programming model, it also differs from dataflow in some key ways. A unique characteristic of DDTs is the ability to dynamically create a task with an `await` clause that specifies multiple DDFs as inputs, and to freely use such tasks in conjunction with regular `async-finish` task parallelism. For example, we are unaware on any past work on dataflow that could support the task parallelism shown in Figure 6. The combination of dynamic `put()`, `get()` operations on DDFs with dynamic `async` task constructs make it possible to express many kinds of algorithmic patterns that would not be considered to be pure dataflow.

Futures have been proposed by Baker and Hewitt in [15]. Implementations of the concept can be seen in MultiLISP by Halstead [17], and in many other languages since. It is possible to create arbitrary task graphs with futures, but each `get()` operation on a future may be a blocking operation unlike the `await` clause in DDTs. Similarly, *I-Structures* [14] have blocking `get()` semantics, but did not support task creation with an `await` clause as in data-driven tasks. Additionally, futures effectively requires that the DDF and `async` creation be fused, whereas DDTs allow a separation between DDFs and `async`s.

Macro-dataflow languages have been proposed in models like Cedar [25] and more recently in the Concurrent Collections (CnC) coordination language [20]. In this work, we showed how DDTs and DDFs and enable the macro-dataflow model to be integrated with task parallelism. The CnC model was an important motivation for the introduction of DDTs, since CnC can be easily compiled down to DDTs by treating a CnC item collection as a collection of DDFs. TRASGO [26] is an example of another coordination language in which the programmer specifies nested series-parallel synchronization structures and leaves the task of generating more general synchronization structures to the compiler. In contrast, DDTs enable the programmer to specify any directed synchronization structure that they may wish to create.

The Nabbit [11] library extension to Cilk++ allows the user to declare arbitrary task dependences so as to create task graphs that cannot be created with nested fork-join operations. Unlike DDTs, where the task graph is specified implicitly via `await` clauses on `async` statements, it is created explicitly in Nabbit via calls to an `AddDep()` method before the start of graph execution. In addition, DDFs are directly amenable to garbage collection, but that is harder to accomplish with the use of keys in Nabbit.

In section VI-B, we observed that a macro-dataflow implementation for Cholesky decomposition in [12] outperforms the alternate parallel implementations studied in that work. The scheduling algorithm used in [12] is akin to the delayed async [21] scheduling policy described in section V. The results in section VI-C show that the Data-Driven Scheduling policy outperforms its delayed async scheduling counterpart.

In summary, DDTs have benefited from influences from past work, but we are not aware of any other parallel programming model that shares DDTs fundamental properties as a data-driven extension to task parallelism.

VIII. Conclusions

In this paper, we proposed an extension to task parallelism called Data-Driven Tasks (DDTs) that can be used to create arbitrary task graph structures. Unlike a normal task that starts execution upon creation, a DDT specifies its input constraints in an `await` clause containing a list of Data-Driven Futures (DDFs). Another advantage of this approach is that all data accesses performed via DDFs are guaranteed to be race-free and deterministic. We described five scheduling algorithms (Coarse-Grain Blocking, Fine-Grain Blocking, Delayed Async, Rollback & Replay, Data-Driven) that can be used to implement DDTs, and discussed performance results for these five algorithms on a variety of benchmark programs and multicore platforms. Our results showed that the Data-Driven scheduler is the best approach for implementing DDTs, both from the viewpoints of memory efficiency and scalable parallelism.

Acknowledgments

We would like to thank Aparna Chandramowlishwaran and Zoran Budimlić for their help with the Cholesky Decomposition benchmark. We also would like to thank David Peixotto for the Black-Scholes implementation and Alina Simion-Sbîrlea for the Heart Wall Tracking implementation. The Rician Denoising application used in this paper was derived from an earlier version provided by Yu-Ting Chen. The motivation for this paper was strongly influenced by weekly discussions between the Intel CnC and Rice Habanero groups, and we are grateful to everyone who contributed to those discussions. We thank the anonymous reviewers and Kunal Agrawal, Mauricio Breternitz, Jack Dennis, Deepak Majeti, Dragoş Sbîrlea, and Justin Teller for their feedback on the paper. We gratefully acknowledge support from an Intel award during 2009-2010. This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127.

References

[1] "The Chapel Language Specification," Tech. Rep., February 2005.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt, *The Fortress Language Specification*, Sun Microsystems, Inc., May 2006.

[3] 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 *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 3rd ed., May 2008. [Online]. Available: http://www.openmp.org

[5] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[6] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[7] V. Cavé, J. Zhao, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," *9th International Conference on the Principles and Practice of Programming in Java*, August 2011.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Journal of parallel and distributed computing*, 1995, pp. 207–216.

[9] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[10] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," in *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, ser. PARA'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1–10.

[11] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, GA, USA, Apr. 2010.

[12] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010, pp. 1 –12.

[13] S. Taşırlar, "Scheduling Macro-DataFlow Programs on Task-Parallel Runtime Systems," Master's thesis, Rice University, April 2011.

[14] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, pp. 598–632, October 1989.

[15] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," *SIGPLAN Not.*, vol. 12, pp. 55–59, August 1977.

[16] V. Sarkar, "Instruction reordering for fork-join parallelism," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 322–336.

[17] R. H. Halstead, Jr., "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, ser. LFP '84. New York, NY, USA: ACM, 1984, pp. 9–17.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[19] R. Barik, V. Cavé, C. Donawa, A. Kielstra, I. Peshansky, and V. Sarkar, "Experiences with an SMP implementation for X10 based on the Java Concurrency Utilities," in *Workshop on Programming Models for Ubiquitous Parallelism (PMUP 2006)*, September 2006.

[20] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar, "Concurrent Collections," *Sci. Program.*, vol. 18, pp. 203–217, August 2010.

[21] Z. Budimlić, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multicore implementations of the concurrent collections programming model," *Proceedings of the 2009 Workshop on Compilers for Parallel Computing (CPC)*, January 2009.

[22] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 57–76.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.

[24] J. Cong, G. Reinman, A. Bui, and V. Sarkar, "Customizable domain-specific computing," *Design Test of Computers, IEEE*, vol. 28, no. 2, pp. 6 –15, march-april 2011.

[25] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar: a large scale multiprocessor," *SIGARCH Comput. Archit. News*, vol. 11, pp. 7–11, March 1983.

[26] A. Gonzalez-Escribano and D. Llanos, "Trasgo: a nested-parallel programming system," *The Journal of Supercomputing*, pp. 1–9, 2009.