Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism

Jun Shirako, Vincent Cavé, Jisheng Zhao, and Vivek Sarkar
Department of Computer Science, Rice University
{shirako,vincent.cave,jisheng.zhao,vsarkar}@rice.edu

ABSTRACT

Parallel reductions represent a common pattern for computing the aggregation of an associative and commutative operation, such as summation, across multiple pieces of data supplied by parallel tasks. In this paper, we introduce *finish accumulators*, a unified construct that supports predefined and user-defined deterministic reductions for *dynamic async-finish task parallelism*. Finish accumulators are designed to be integrated into *terminally strict* models of task parallelism as in the X10 and Habanero-Java (HJ) languages, which is more general than *fully strict* models of task parallelism found in Cilk and OpenMP.

In contrast to lower-level reduction constructs such as atomic variables, the high-level semantics of finish accumulators allows for a wide range of implementations with different accumulation policies, *e.g., eager*-computation vs. *lazy*-computation. The best implementation can thus be selected based on a given application and the target platform that it will execute on. We have integrated finish accumulators into the Habanero-Java task parallel language, and used them in both research and teaching. In addition to their higher-level semantics, experimental results demonstrate that our Java-based implementation of finish accumulators delivers comparable or better performance for reductions relative to Java's atomic variables and concurrent collection libraries.

1. INTRODUCTION

A large number of programming models have been recently proposed to address the need for improved productivity and scalability in parallel programming *e.g.,* Intel Threading Building Blocks (TBB) [16], Java Concurrency [7], OpenMP 3.0 tasks [13], Cilk++[11], X10 [4], and Habanero-Java (HJ) [2]. Unlike the SPMD programming models from past work that assume a fixed number of concurrent threads, these models advocate the use of *dynamic task parallelism* to specify lightweight concurrent tasks that can be created at any time and in any amount during program execution.

It is well known that a necessary condition for determinism in a parallel program is that any two operations that are not causally related must commute. *Parallel reductions* represent a common pattern for computing the aggregation of parallel commutative operations across multiple pieces of data supplied by parallel tasks. Typically, the programmer selects a predefined operator for the reduction, but a few programming models also allow programmers to specify a user-defined reduction function instead.

In this paper, we introduce *finish accumulators*, a unified construct that supports predefined and user-defined parallel reductions for *dynamic task parallelism* i.e., for models in which the set of tasks participating in a reduction can increase. Finish accumulators are designed for *terminally strict* task parallelism [10], as in the `async` and `finish` constructs found in X10 and Habanero-Java languages, which is more general than the *map-reduce* model as well as the *fully strict* models of task parallelism found in Cilk and OpenMP. Given a computation dag [1, 10], if every join edge goes from a task to its spawn tree ancestor, the computation is called a *strict* computation. If every join edge goes from a task to its spawn tree parent, the computation is called a *fully-strict* computation [1]. If a computation is strict and every join edge goes from the last instruction of a task to its spawn tree ancestor, the computation is called *terminally-strict* [10].

We have integrated finish accumulators into the HJ task parallel language, and used them in both research and teaching. In contrast to lower-level reduction constructs such as atomic variables, the high-level semantics of finish accumulators allows for a wide range of implementations with different accumulation policies, *e.g., eager*-computation vs. *lazy*-computation. Experimental results demonstrate that our Java-based implementation of finish accumulators delivers comparable or better performance for computing reductions relative to Java's atomic variables and concurrent collections. Specifically, our Java-based implementation of finish accumulators achieved speedups of up to 2.2× on an 8-core Intel Core i7 system and up to 11.4× on a 64-thread Sun UltraSPARC T2 relative to the parallel reduction using standard Java concurrency constructs.

The rest of the paper is organized as follows. Section 2 discusses background related to reductions and dynamic task parallelism. Section 3 describes the proposed programming interface and semantics of finish accumulators. Section 4 discusses implementation details, and Section 5 presents our experimental results. Finally, Section 6 and Section 7 summarize related work and our conclusions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WODET'13, March 17, 2013, Houston, Texas, USA.
Copyright 2013 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

2. BACKGROUND

```
1: int cutoff; // Threshold to stop creating new tasks
2: AtomicInteger atom = new AtomicInteger(0);
3:
4: void fib (int n, int depth) {
5:  if (n < 2) { atom.addAndGet(n); }
6:  else {
7:    async seq (depth >= cutoff) fib(n - 1, depth + 1);
8:    async seq (depth >= cutoff) fib(n - 2, depth + 1);
9: }}
```

Figure 1: Reduction using JUC AtomicInteger

2.1 Habanero-Java

The Habanero Java (HJ) language under development at Rice University [2] proposes an execution model for multi-core processors that builds on four orthogonal constructs:

1. Lightweight *dynamic task creation and termination* using *async* and *finish constructs* [10].
2. *Locality control* with task and data distributions using the *place* construct [3]. Places enable co-location of async tasks and data objects.
3. Mutual exclusion and isolation among tasks using the *isolated* construct [12].
4. Collective and point-to-point synchronization using the *phasers* construct [15] along with their accompanying *phaser accumulators* [14].

The HJ language derives from initial versions of the X10 language (up to v1.5) that used Java as the underlying sequential language [4]. Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [7] is also permitted in HJ programs. We briefly recapitulate the *async* and *finish* constructs for task creation and termination in HJ.

- **async:** The statement "`async` ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩. Execution of the `async` statement returns immediately i.e., the parent task can proceed immediately to its next statement.
- **forasync:** The statement "`forasync (point p :` *region*`)` ⟨*stmt*⟩" creates new child tasks as iterations of a parallel loop. Note that *region* is the iteration space.
- **finish:** The statement "`finish` ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait till all sub-tasks created within ⟨*stmt*⟩ have terminated at the end of the finish scope. Note that the sub-tasks include transitively spawned tasks. There is an implicit finish statement surrounding the main program.

2.2 Reductions in Java

The Java Concurrency Utilities [7] is a standard library in Java, which supports various atomic constructs such as AtomicInteger and ConcurrentHashMap that can be used to implement predefined reductions. As a simple example of a sum reduction using AtomicInteger in an HJ program, Figure 1 computes the n-th Fibonacci number in AtomicInteger `atom` using a parallel variant of the standard recursive formulation. A `seq` clause in HJ's `async` construct is used to spawn tasks until the `depth` of the recursive `fib` invocation reaches a specified `cutoff`. Although this program has a simple structure with large amounts of parallelism, it raises a couple of issues regarding usability and efficiency. From the usability viewpoint, programmers have to ensure that no race conditions or sources of nondeterminism arise from shared data accesses in dynamic tasks. Since the intermediate state of the `atom` object is always visible to a task, incomplete results can be referenced nondeterministically (though that's not the case in Figure 1). From the performance viewpoint, the memory and network contention arising from a single atomic variable can be a scalability bottleneck, especially with a large number of cores.

The finish accumulators introduced in this paper protect programmers from both these issues. Semantic guarantees such as determinism and race freedom follow from the high-level interfaces in an accumulator, which in turn also offers great flexibility in the choice of implementation design for a given application and target platform. In particular, we introduce a *lazy* accumulation policy that avoids the memory and network contention issues mentioned above, when compared with an *eager* policy based on atomic operations.

```
 1: static struct Reducer implements Reducible[Int] {
 2:   public def zero() = 0;
 3:   public operator this(a:Int, b:Int) = a + b;
 4: }
 5: public def run() {
 6:   val result = finish (Reducer()) {
 7:     val d = (0..(N-1))->here;
 8:     for (p in d.region) async { offer 1; }
 9:   };
10: }
```

Figure 2: Reduction using X10 collecting-finish

2.3 X10 Collecting Finish

A more recent example of reductions for terminally strict parallel programs can be found in X10's *collecting-finish* construct [17]. A simple example is shown in Figure 2. As with other user-defined reductions, the reduction semantics is specified by a data type (struct) that implements a `Reducible[T]` interface (lines 1–4). The interface requires the user to specify the reduction `operator` and the `zero` (identity) value as methods.

As shown in lines 6–9, the result of the collecting finish is obtained by treating the finish construct as an expression (lval), thereby allowing at most one reduction per finish construct. In contrast, the finish accumulators introduced in this paper support multiple accumulators per finish construct.

3. PROGRAMMING MODEL

In this section, we introduce the programming interface and semantics of finish accumulators. There are two logical operations, *put*, to remit a datum and *get*, to retrieve the result from a well-defined synchronization (`end-finish`) point. Section 3.1 describes the details of these operations, and Section 3.2 describes how user-defined reductions are supported in finish accumulators.

3.1 Accumulator Constructs

The operations that a task, T_i, can perform on accumulator, *ac*, are defined as follows.

- **new:** When task T_i performs a "`ac = new accumulator(op, dataType);`" statement, it creates a new ac-

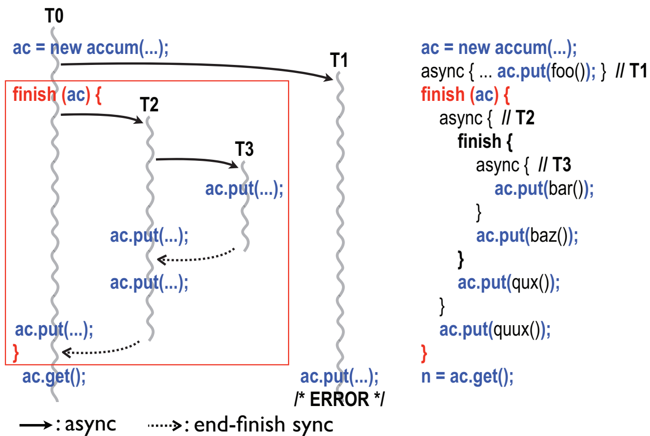

Figure 3: Finish accumulator example with three tasks that perform a correct reduction and one that throws an exception

cumulator, *ac*, on which T_i is registered as the *owner task*. Here, *op* is the reduction operator that the accumulator will perform, and *dataType* is the type of the data upon which the accumulator operates. Currently supported predefined reduction operators include `SUM`, `PROD`, `MIN`, and `MAX`; `CUSTOM` is used to specify user-defined reductions.

- **put:** When task T_i performs an "*ac*.`put`(*datum*);" operation on accumulator *ac*, it sends *datum* to *ac* for the accumulation, and the accumulated value becomes available at a later end-finish point. The runtime system throws an exception if a `put()` operation is attempted by a task that is not the owner and does not belong to a `finish` scope that is associated with the accumulator. When a task performs multiple `put()` operations on the same accumulator, they are treated as separate contributions to the reduction.

- **get:** When task T_i performs an "*ac*.`get()`" operation on accumulator *ac* with predefined reduction operators, it obtains a `Number` object containing the accumulated result. Likewise "*ac*.`customGet()`" on *ac* with a `CUSTOM` operator returns a user-defined `T` object with the accumulated result. When no `put` is performed on the accumulator, `get` returns the identity element for the operator, *e.g.,* `0` for `SUM`, `1` for `PROD`, `MAX_VALUE`/`MIN_VALUE` for `MIN`/`MAX`, and user-defined identity for `CUSTOM`.

- **Summary of access rules:** The owner task of accumulator *ac* is allowed to perform `put`/`get` operations on *ac* and associate *ac* with any `finish` scope in the task. Non-owner tasks are allowed to access *ac* only within `finish` scopes with which *ac* is associated. To ensure determinism, the accumulated result only becomes visible at the `end-finish` synchronization point of an associated `finish`; `get` operations within a `finish` scope return the same value as the result at the beginning of the `finish` scope. Note that `put` operations performed by the owner outside associated `finish` scopes are immediately reflected in any subsequent `get` operations since those results are deterministic.

```
1: void foo() {
2:   accum<Coord> ac = new accum<Coord>(Operation.CUSTOM,
3:                                      reducible.class);
4:   finish(ac) {
5:     forasync (point [j] : [1:n]) {
6:       while(!isEmpty(j)) {
7:         ac.put(getCoordinate(j));
8:   } } }
9:   Coord c = ac.customGet();
10:  System.out.println("Furthest: " + c.x + ", " + c.y);
11: }
12:
13: class Coord implements reducible<Coord> {
14:   public double x, y;
15:   public Coord(double x0, double y0) {
16:     x = x0;   y = y0;
17:   }
18:   public Coord identity(); {
19:     return new Coord(0.0, 0.0);
20:   }
21:   public void reduce(Coord arg) {
22:     if (sq(x) + sq(y) < sq(arg.x) + sq(arg.y)) {
23:       x = arg.x;   y = arg.y;
24:   } }
25:   private double sq(double v) { return v * v; }
26: }
```

Figure 4: User-defined reduction example

To associate a `finish` statement with multiple accumulators, T_{owner} can perform a special `finish` statement of the form, "`finish` $(ac_1, ac_2, \cdots, ac_n)\langle stmt\rangle$". Note that `finish` (*ac*) becomes a no-op if *ac* is already associated with an outer `finish` scope.

Figure 3 shows an example where four tasks T_0, T_1, T_2, and T_3 access a finish accumulator *ac*. As described earlier, the `put` operation by T_1 throws an exception due to nondeterminism since it is not the owner and was created outside the `finish` scope associated with accumulator *ac*. Note that the inner `finish` scope has no impact on the reduction of *ac* since *ac* is associated only with the outer `finish`. All `put` operations by T_0, T_2, and T_3 are reflected in *ac* at the `end-finish` synchronization of the outer `finish`, and the result is obtained by T_0's `get` operation.

3.2 User-defined Reductions

User-defined reductions are also supported in finish accumulators, and its usage consists of these three steps:
1) specify `CUSTOM` and `reducible.class` as the accumulator's operator and type,
2) define a class that implements the `reducible` interface,
3) pass the implementing class to the accumulator as a type parameter.

Figure 4 shows an example of a user-defined reduction. Class `Coord` contains two double fields, `x` and `y`, and the goal of the reduction is to find the furthest point from the origin among all the points submitted to the accumulator. The `reduce` method computes the distance of a given point from the origin, and updates `x` and `y` if `arg` has a further distance than the current point in the accumulator.

In the current implementation, programmers are assumed to make the `reduce` method commutative and associative, and hence it can produce nondeterministic results if the assumption is not satisfied. There are two challenges in order

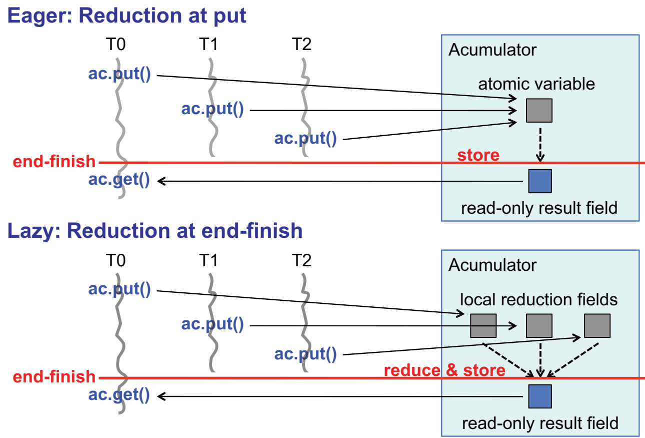

Figure 5: Eager and Lazy Implementation

to avoid such indeterminism in the user-defined reduction: compile-time check for commutativity and associativity and runtime support for ordered reductions. The compile-time checking can be implemented as pattern recognition if the `reduce` method is implemented as combination of collecting operations and/or predefined operations. When commutativity and associativity were not guaranteed at compile-time, the runtime will select the ordered reduction instead of parallel reductions introduced in Section 4. The ordered reduction will preserve all data from the `put` operations with the depth-first order of the `async` tree, and sequentially reduce the preserved data at the `end-finish` synchronization point. These challenges are important future work to be addressed.

4. IMPLEMENTATION

This section describes the implementation of finish accumulators in the Habanero-Java runtime. We provide both *eager* and *lazy* implementation approaches for finish accumulators. The implementation choice can be specified as a command-line argument or an environment variable, and does not require any change in the source code. (Automatic selection of the implementation choice is a subject for future research.) As shown in Figure 5, an eager-compute implementation performs an accumulation immediately when a `put` operation is invoked, while a lazy-compute implementation merely captures each datum in a `put` operation so that it can be accessed later when the reduction computation is performed at an `end-finish` point.

4.1 Eager Implementation Policy

The eager approach uses atomic operations to incrementally perform the reduction as soon as a datum is put by a task. The accumulator contains an atomic variable of the specified type, which is updated at each `put` operation. For predefined reductions (`SUM`, `PROD`, `MIN`, and `MAX`), we used Java's AtomicInteger class for `int` accumulators, and created an AtomicDouble class based on Java's AtomicReference class for `double` accumulators. For user-defined reductions, the eager approach simply uses a standard Java lock implementation (ReentrantLock), so that the user-defined data structures can be updated within a critical section.

Figure 6 shows the pseudocode for eager implementations of predefined reductions (lines 2–11) and user-defined reductions (lines 14–22). The `while` loop in lines 4–9 corresponds to the atomic update for the `SUM` operation. Likewise, the other predefined reductions rely on the `compareAndSet()` operation supported in AtomicInteger. The `put` operation for user-defined reductions simply performs the user's `reduce()` method guarded by a general lock. At the `end-finish` synchronization point, the value stored in `atomI` or `transitState` is copied into the result field of the accumulator and becomes visible to any task registered on the accumulator.

```
 1: AtomicInteger atomI;
 2: public void putEager(int val) { // Predefined int
 3:   if (ope == Operator.SUM) {
 4:     while (true) {
 5:       int cur = atomI.get();
 6:       int neo = cur + val;
 7:       if (atomI.compareAndSet(cur, neo)) break;
 8:       else delay();              // Optional delay
 9:     }
10:   } else ...                     // For PROD, MIN, MAX
11: }
12:
13: T transitState;
14: public void putEager(T arg) {   // User-defined
15:   lockForReduce.lock();         // General lock (JUC)
16:   try {
17:     if (transitState == null)
18:       transitState = arg;
19:     else
20:       transitState.reduce(arg); // Reduce exclusively
21:   } finally { lockForReduce.unlock(); }
22: }
```

Figure 6: Eager implementation of the put method

```
 1: int accumArrayI[];
 2: public void putLazy(int val, int id) { // Predefined
 3:   if (ope == Operator.SUM) {
 4:    accumArrayI[id*strideI] += val;
 5:   } else ...                          // PROD, MIN, MAX
 6: }
 7:
 8: T reducibleArray[];
 9: public void putLazy(T arg, int id) { // User-defined
10:   if (reducibleArray[id*strideI] == null)
11:     reducibleArray[id*strideI] = arg;
12:   else
13:     reducibleArray[id*strideI].reduce(arg);
14: }
```

Figure 7: Lazy implementation of the put method

The eager approach has a straightforward implementation and good portability across different runtimes because it is independent of the underlying task scheduler implementation. On the other hand, the concurrent accesses to the single atomic variable may suffer from significant memory and network contention. To improve the scalability of atomic operations, we employ the idea of adding a delay so as to reduce the contention. There are various choices in the implementation of the delay function, such as random, proportional, exponential, and constant. For the results reported in this paper, we used a random function of the form, `delay * (1.0 + rand.nextDouble())`, where `delay` is a tunable parameter for each platform and `rand` is an instance of java.util.Random whose `nextDouble()` method returns a `double` value between 0 and 1.

4.2 Lazy Implementation Policy

In the lazy approach, each worker, which is a thread assigned to process multiple tasks in the runtime scheduler, has a local reduction field to accumulate data from the assigned tasks. The global reduction across workers is delayed until the `end-finish` synchronization point. We used the Habanero-Java work-stealing scheduler [10], for which the number of workers is fixed at program startup time and hence an accumulator contains a fixed size array (with one entry per worker) of the corresponding data type, `int`, `double`, or `T`.

Figure 7 shows the pseudocode for predefined reductions (lines 2–6) and user-defined reductions (lines 9–14) in the lazy implementation. The `id` of each worker is given by the runtime, and `strideI` (= cache line size divided by array element size) is for array padding to avoid false sharing. As shown in the codes, the `put` operation for both predefined and user-defined reductions locally updates the corresponding array element. The global reduction at the `end-finish` synchronization point reduces all local results and stores into the result field.

The lazy approach allows large amount of parallelism without any inter-thread communication except for the single global reduction. It can be the best implementation for the runtime system, when the number of workers is fixed and relatively small. For large number of workers, the global reduction will need to be replaced by a reduction tree for scalability. Another issue with the current lazy approach is that it is less portable than the eager approach. For instance, we will need to replace the fixed size array by a dynamic collection data structure for runtime systems in which the number of workers can be dynamically changed.

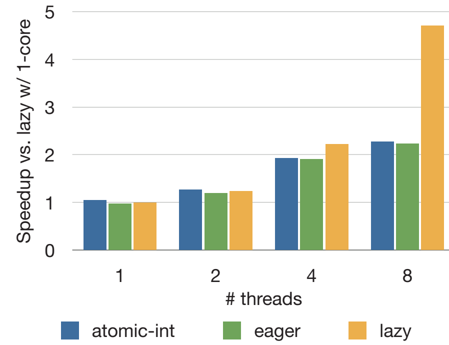

Figure 8: Speedup of Nqueens vs. Lazy with 1-core on Core i7

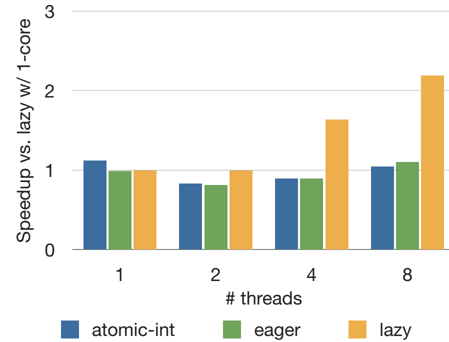

Figure 9: Speedup of Fib vs. Lazy with 1-core on Core i7

5. EXPERIMENTAL RESULTS

In this section, we present experimental results for an HJ-based implementation of finish accumulators on two platforms. The first platform is a 8-core (2 quad-cores) 2.4GHz Intel *Core i7* system with 12 GB main memory running Red Hat Enterprise Linux Server release 5.5. We conducted all experiments on this system by using the Java SE Runtime Environment (build 1.6.0_24-b07) with Java HotSpot Server VM (build 19.1-b02, mixed mode). The second platform is a 64-thread (8 cores $\times$ 8 threads/core) 1.2 GHz Sun *UltraSPARC T2* system with 32 GB main memory running Solaris 10. We used the Java 2 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot Server VM (build 1.5.0_12-b04, mixed mode). All results in this paper were obtained using the Habanero-Java compiler and runtime [2] with the work stealing scheduler [9]. For the purpose of reducing the impact of JIT compilation time and other JVM services in the performance comparisons, the main HJ program was extended with a 10-iteration loop within the same process, and the result with the smallest execution time was reported in each case.

We use the following three benchmarks, two benchmarks for predefined reductions and one for user-defined reductions:

- Nqueens was ported from the Barcelona OpenMP Tasks Suite (BOTS) benchmarks [5] to Habanero-Java (HJ). For both platforms, we ran the benchmark with $n = 13$ and $cutoff = 4$. Since, Nqueens counts the total number of solutions found by parallel tasks and (in our implementation) the number of pruned branches in the search tree, we used predefined `SUM` finish accumulators for both counts.
- Fib was also ported from the BOTS benchmarks. It computes the n-th Fibonacci number using a recursive parallelization strategy. Here, $n = 40$ and $cutoff = 12$ was used as inputs for both platforms. A predefined `SUM` finish accumulator was used for this benchmark, similar to the AtomicInteger version shown earlier in Figure 1.
- WordCount is an HJ program that counts the number of occurrences of each word in a given text document. This implementation divides the input document into chunks of even size. Each parallel task processes its assigned chunk, after which the results are combined with a user-defined finish accumulator. The input document for both platform has 2,000,000 words.

In the following sections, we compare three implementation variants for these benchmarks: 1) finish accumulator with *eager* reduction policy, 2) *lazy* policy, and 3) *JUC-based* variant using `java.util.concurent` libraries (`AtomicInteger` for Nqueens and Fib, and `ConcurrentHashMap` for WordCount). (Note that there are no determinism guarantees when the JUC libraries are used.)

5.1 Speedup on Core i7

Figures 8–10 show the speedup numbers on *Core i7*. Here, the baseline is the single thread execution time of finish accumulator with *lazy* policy, which is the primary focus of this

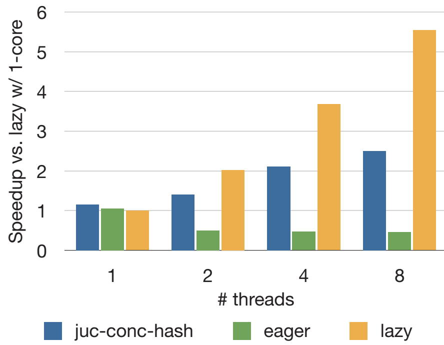

Figure 10: Speedup of WordCount vs. Lazy with 1-core on Core i7

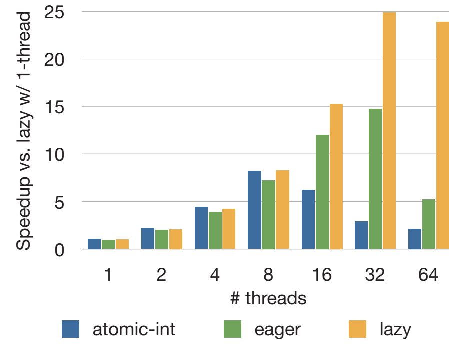

Figure 12: Speedup of Fib vs. Lazy with 1-thread on UltraSPARC T2

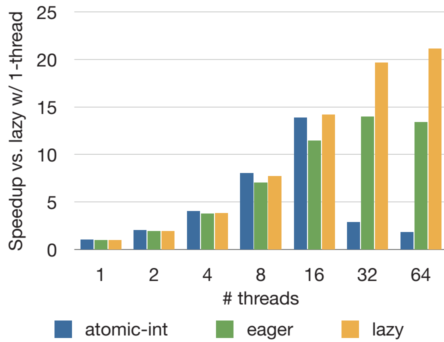

Figure 11: Speedup of Nqueens vs. Lazy with 1-thread on UltraSPARC T2

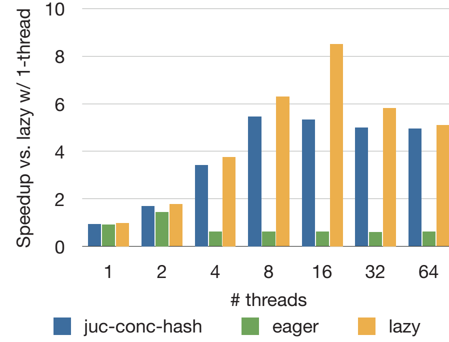

Figure 13: Speedup of WordCount vs. Lazy with 1-thread on UltraSPARC T2

paper[1]. As shown in Figures 8–10, finish accumulator with *lazy* implementation policy shows better scalability than *eager* and *JUC-based* variants for all three benchmarks. As discussed in Section 4.2, the *lazy* policy can be the best approach for a runtime with a fixed number of worker threads such as HJ's work-stealing scheduler. The predefined reduction with *eager* policy uses AtomicInteger internally and shows almost the same performance for Nqueens and Fib compared to an explicit use of JUC's AtomicInteger. Likewise, the *eager* policy for user-defined reductions employs JUC's ReentrantLock so as to support any reduction defined by users. Therefore, it performs worse than the *JUC-based* version using ConcurrentHashMap and *lazy* finish accumulator version of WordCount.

5.2 Speedup on UltraSPARC T2

Figures 11–13 show the speedup numbers on *UltraSPARC T2*. Similar to the results for *Core i7*, the *lazy* policy shows better scalability compared with the *eager* policy and the *JUC-based* variants. Further, the *eager* policy on *UltraSPARC T2* shows better scalability than the *JUC-based* variants for Nqueens and Fib. As discussed in Section 4.1, the predefined operations of *eager* policy employs the delay optimization to reduce memory and network contention for atomic operations. This optimization is especially important on platforms with larger number of hardware threads such as *UltraSPARC T2*. On the other hand, the *eager* implementation relies on a general lock implementation to support user-defined reduction, and trades off flexibility against scalability. A more efficient implementation of the eager policy for user-defined reductions is a subject for future work.

[1]This variant has almost same execution time as the serial versions except for Fib, whose computation cost is smaller than communication cost due to reduction.

6. RELATED WORK

It is well known from past work that reductions and scans with associative operations can be performed in parallel. When reductions are performed with dynamic parallelism (*e.g.,* as in OpenMP [13]), then it is convenient to assume commutativity as well, as in the finish accumulators introduced in this paper. MPI [8] supports both predefined and user-defined reductions in a distributed-memory context.

In OpenMP, a parallel construct can optionally include a clause which specifies a reduction operator and a list of scalar shared locations. For each shared location, a private copy array is allocated sized to the number of implicit/explicit threads created in the parallel construct, initialized to the identity element for the operator. On exit from the reduction, these arrays are populated with the values of the private copies in accordance with the specified operator. The supported reduction operators include sum, product, min, max, logical-or, logical-and, bitwise-or, bitwise-and, and bitwise-xor.

In MPI, reductions are embodied in the following collective routines: `MPI_Reduce`, `MPI_AllReduce`, `MPI_Scan` and `MPI_Reduce_scatter`. MPI supports various type of predefined reduction operators including those in OpenMP, `MINLOC` and `MAXLOC` operations. Furthermore, `MPI_OP_CREATE` enables user-defined reduction which can be used in the collective routines.

In addition to OpenMP and MPI, task-parallel models

such as Cilk++ [11] and TBB [11] also support predefined and user-defined reductions. In Cilk++'s Reducer construct [6], each task has its own local "view" of the reduction variable/object and these local views are reduced at every synchronization point. Therefore, Cilk++ Reducer model does not need to specify a particular synchronization point for reductions. Additionally, programmers are allowed to access intermediate results of Reducer objects, which gives more flexibility to expert users but also increases the possibility of errors due to nondeterminism. On the other hand, finish accumulator model specifies an `end-finish` synchronization point where the reduction is to be completed, and prevents programmers from accessing incomplete intermediate results. As we have seen, this model enables time/space-efficient reduction implementations with flexibility in the choice of implementation design for a given application and target platform.

Finally, finish accumulators can be viewed as an extension of HJ's *phaser accumulators* [14]. Phaser accumulators support per-phase reductions, which are more restrictive than the finish accumulators introduced in this paper. Integration of phaser accumulators and finish accumulators is an interesting topic for future work.

7. CONCLUSIONS

In this paper, we introduced finish accumulators, a unified construct to support predefined and user-defined parallel reduction for dynamic task parallelism. We defined the programming model and semantics of finish accumulators in a manner that guarantees determinism, while also allowing for large flexibility in implementation choices. In this paper, we presented two implementation variants for finish accumulators: eager and lazy. Experimental results obtained on two different platforms demonstrated that our Java-based implementation of finish accumulators delivers comparable or better performance than the reduction implementation using Java's standard concurrent utilities, while also guaranteeing determinism.

Opportunities for future research related to accumulators include support of hierarchical tree-based reductions for further scalability, compiler optimizations for fusing multiple put operations by sequentialized tasks, efficient implementations for user-defined eager reductions, and integration of finish accumulators with phaser accumulators.

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.

We would like to thank members of the Habanero group at Rice for contributions to the Habanero Java infrastructure used in this research, and especially Yi Guo for experiences gained from his early implementation of finish accumulators. We are grateful to the anonymous reviewers for their comments and suggestions. Finally, we would like to thank Keith Cooper for providing access to the Xeon system and Doug Lea for providing access to the UltraSPARC T2 system used to obtain the performance results reported in this paper.

8. REFERENCES

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[2] V. Cavé et al. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[3] S. Chandra et al. Type inference for locality analysis of distributed data structures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, New York, NY, USA, 2008. ACM.

[4] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[5] A. Duran et al. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP'09*, 2009.

[6] M. Frigo et al. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, 2009.

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

[8] W. Gropp et al. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[9] Y. Guo et al. Slaw: a scalable locality-aware adaptive work-stealing scheduler. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS).*

[10] Y. Guo et al. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS '09: International Parallel and Distributed Processing Symposium*, 2009.

[11] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[12] R. Lublinerman et al. Delegated Isolation. In *OOPSLA '11: Proceeding of the 26th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2011.

[13] OpenMP Application Program Interface, version 3.0. http://www.openmp.org/mp-documents/spec30.pdf.

[14] J. Shirako et al. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Proc. of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09.

[15] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proc. of the 22nd international conference on Supercomputing*, ICS '08, 2008.

[16] TBB. http://www.threadingbuildingblocks.org.

[17] C. Zhang et al. Evaluating the performance and scalability of mapreduce applications on x10. In *Proceedings of the 9th international conference on Advanced parallel processing technologies.*