Delegated Isolation

Roberto Lublinerman
Pennsylvania State University
rluble@psu.edu

Jisheng Zhao
Rice University
jisheng.zhao@rice.edu

Zoran Budimlić
Rice University
zoran@rice.edu

Swarat Chaudhuri
Rice University
swarat@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

Isolation—the property that a task can access shared data without interference from other tasks—is one of the most basic concerns in parallel programming. In this paper, we present *Aida*, a new model of isolated execution for parallel programs that perform frequent, irregular accesses to pointer-based shared data structures. The three primary benefits of Aida are *dynamism*, *safety and liveness guarantees*, and *programmability*. First, Aida allows tasks to dynamically select and modify, in an isolated manner, arbitrary fine-grained regions in shared data structures, all the while maintaining a high level of concurrency. Consequently, the model can achieve scalable parallelization of regular as well as irregular shared-memory applications. Second, the model offers freedom from data races, deadlocks, and livelocks. Third, no extra burden is imposed on programmers, who access the model via a simple, declarative isolation construct that is similar to that for transactional memory.

The key new insight in Aida is a notion of *delegation* among concurrent isolated tasks (known in Aida as *assemblies*). Each assembly A is equipped with a region in the shared heap that it owns—the only objects accessed by A are those it owns, guaranteeing race-freedom. The region owned by A can grow or shrink flexibly—however, when A needs to own a datum owned by B, A *delegates* itself, as well as its owned region, to B. From now on, B has the responsibility of re-executing the task A set out to complete. Delegation as above is the only inter-assembly communication primitive in Aida. In addition to reducing contention in a local, data-driven manner, it guarantees freedom from deadlocks and livelocks.

We offer an implementation of Aida on top of the Habanero Java parallel programming language. The implementation employs several novel ideas, including the use of a union-find data structure to represent tasks and the regions that they own. A thorough evaluation using several irregular data-parallel benchmarks demonstrates the low overhead and excellent scalability of Aida, as well as its benefits over existing approaches to declarative isolation. Our results show that Aida performs on par with the state-of-the-art customized implementations of irregular applications and much better than coarse-grained locking and transactional memory approaches.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Languages, Design

Keywords Isolation, Programming abstractions, Irregular parallelism, Contention

1. Introduction

Isolation[1]—the property that a task can access shared data without interference from other tasks—has always been a fundamental concern in shared-memory parallel programming. It is also the topic of much recent research [19, 22, 35]. Much of this research is driven by the observation that the traditional approach to enforcing isolation—lock-based synchronization—has some serious shortcomings, and falls short of the demands made by everyday programmers in an era of multicore computers. On one hand, low-level, fine-grained locking is too complex and error-prone, and requires

[1] In the parallel programming literature, the terms "isolation" and "atomicity" are often conflated. As advocated in [19], we use *isolated* to indicate a statement that executes without interference from other isolated statements, as opposed to *atomic* which implies strong atomicity.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

an effort on the part of the programmer that does not scale well with software size. On the other hand, coarse-grained locking leads to scalability bottlenecks. Consequently, numerous research groups have embarked on quests for declarative and scalable approaches to isolation.

At the language level, a declarative approach to isolation lets the programmer demarcate blocks of code that are to be executed in an isolated manner—the complexity of maintaining isolation is thereby pushed into the compiler and the runtime system. As might be expected, proper design and implementation of such compilers and runtimes is highly nontrivial. A primary concern here is scalable performance on challenging, real-world applications. A second goal is programmability: the programming interface for the construct should be as simple, as general, and as modular as possible. A third goal is the ease of reasoning about programs that the model provides and the availability of high-level correctness guarantees, such as freedom from deadlocks and livelocks.

Meeting the above goals is especially daunting in settings where the goal is to parallelize applications that make frequent dynamic accesses to large, potentially sparse shared-memory data structures [25]. Such *sparse graph applications* appear in a wide range of domains including data mining, scientific computing, compilers, graphics, network science, and system modeling [32]: concrete examples include classic graph algorithms like shortest-paths or minimum spanning trees, algorithms for ray-tracing, Delaunay triangulation, mesh refinement [10], epidemic simulations [4], N-body simulations, and solutions of partial differential equations. These applications operate on large shared data structures, are resource-intensive and contain large amounts of intrinsic parallelism. However, building scalable runtime systems guaranteeing isolation is especially challenging in this context, as the available parallelism is highly data-dependent and prone to changes at runtime, with some pathological instances exhibiting no parallelism at all. Not only does such *irregular parallelism* confound static parallelization, it baffles dynamic approaches based on transactional memory as well [26].

In this paper, we present *Aida*, a new model of isolated concurrent execution targeted at this difficult problem space. Rather than adapting an existing model for isolated execution (such as software transactions), we aimed in this effort for a clean operational semantics informed by a foundational perspective on irregular parallelism. The outcome—Aida—is a minimalistic execution model containing a single primitive for data communication that lets a task A *delegate* its work to a different task B. Pleasantly, the model satisfies each of our design objectives:

- Our implementation of Aida (on top of the Habanero-Java parallel programming language [7, 18]) performs orders of magnitude better than the Java-based transaction library DSTM2 [12] on irregularly parallel benchmarks from the Lonestar benchmark suite [24]. Its scalability is competitive with Galois [25], a custom library and programming model for parallelizing loops with irregular data accesses.
- Like transactions and unlike Galois, Aida offers a highly general and minimalistic programming model. The programmer uses a single keyword to declare certain asynchronous tasks to be isolated, and no further annotation is demanded. Aida can support any computation in an isolated task, and is not restricted to operations on preselected data structures.
- Aida satisfies strong safety and liveness properties, guaranteeing, among others, deadlock- and livelock-freedom.

In the rest of this paper, we elaborate on the main ideas in Aida (Section 2), our implementation of the model (Section 3), and experimental results (Section 4). Related work is discussed in Section 5; we conclude with some discussion in Section 6.

2. The Aida Execution Model

The first basic insight behind Aida is the following:

> In a problem domain where the extent of available parallelism is data-dependent and prone to dynamic changes, granularity of tasks, communication between tasks, and management of contention should be dynamic as well.

In most traditional models of concurrency, the logic for organizing an application into parallel tasks, as well as the communication between these tasks, is decided statically. This is a deficiency in the context of irregular applications, where the optimal granularity of parallelism depends intimately on the problem instance.

For example, it is known [25, 28, 30] that while typical irregular applications require global memory access in the worst case, the effects of their imperative updates are often restricted to small, local regions in shared data structures. A natural strategy for parallelization, then, would be to let each of these regions be processed by a distinct concurrent, isolated task. However, because the regions in question are highly instance-dependent, it is impossible to know statically the number and schedule of tasks that would lead to optimal parallelism, as well as interferences among tasks. A more dynamic approach is needed.

To see why the principled design of such a dynamic approach is not obvious, consider a scenario that is unavoidable in any dynamic approach to isolation. Here, tasks A and B, executing isolated code and each owning parts of the heap, conflict over an object u in the shared heap. (See Figure 1; here nodes represent shared objects, edges represent pointers between objects, the shaded ovals represent the regions of objects owned by the two tasks.) Clearly, only one of the two tasks—say A—should be allowed to access u. But what

happens to B? Letting B block while maintaining ownership of some data does not seem to be a good idea: not only will tasks seeking B's data now have to wait, in the worst case, we will have deadlocks. Another option is to let the task manager retry B. If we retry B too soon, we will have unnecessary contention that will degrade performance; in the worst case, we will have livelocks. If we do not retry B soon enough, do we release all objects held by B? If so, what if a task C acquires some of this data before B is retried?

We note that the "right" strategy for concurrent execution here depends to a large extent on which thread owns which objects, and it is imperative in these settings that task and ownership management are somehow "coupled." Unfortunately, this is not the case in many traditional dynamic approaches to isolation. Rather than using an operational semantics that relates tasks with ownership, they often resolve conflicts using heuristics like exponential back-off [34] that do not offer solid guarantees, are divorced from high-level programming models, and depend on ad hoc parameters.

In contrast, the Aida execution model is based on a single principle that ties together management of parallel, isolated tasks and management of ownership. The fundamental question in task management is "When should two isolated tasks be run in parallel?" The answer provided here is:

> Two isolated tasks A and B should be run in parallel when A and B are not trying to establish ownership over the same object in the shared heap.

Indeed, if two tasks are trying to establish ownership over (i.e., make isolated accesses to) the same objects in heap, then concurrency among them is not serving any useful purpose. We would therefore be better off if A and B were *merged* into a single task.

To make such a mechanism possible, we let each task in Aida have *explicit* ownership of a local region in the shared heap, and only access objects in that region. This task ownership abstraction is borrowed from the Chorus programming model [28] for irregular applications. As in Chorus [28], such a task is called an *(object) assembly*. By definition, an assembly accesses the heap in an isolated manner. However, as discussed in Section 5, the Chorus model is more restrictive than Aida because it is only applicable to parallelization of *cautious* applications [30] and it lacks Aida's notion of delegated isolation.

In case of conflicts among concurrently executing assemblies, the key new notion of *delegation* comes into play. In particular, consider the scenario in Figure 1: A and B are assemblies, and the shaded ovals depict their owned regions. Now suppose B attempts to access the object u via a pointer. In Aida, this conflict scenario is handled by letting B *delegate* itself and its owned region to A. In essence, A and B are now "merged," and from now on, A: (1) owns the region that B passed to A; and (2) has the responsibility of completing the task that B set out to do.

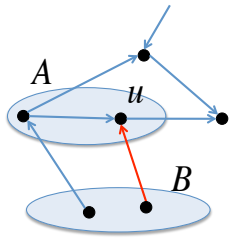


Figure 1. An example of conflicting tasks, A and B, due to accesses to object u. (In the figure, nodes correspond to objects, arrows to pointers/references, and shaded ovals to regions.)

Delegation as above is the only inter-task communication primitive in Aida. The delegation mechanism is accessed by the programmer using a simple programming construct similar to that used in software transactions. The core programming model contains only two parallel constructs: a two-word keyphrase "`async isolated`" that forks an asynchronous task that is to be executed in an isolated manner, and a keyword "`finish`" for synchronization. This framework for scoped fork-join parallelism is derived from the X10 [9] and Habanero Java [7] languages, but applied to '`async isolated`" tasks in Aida. While the "`finish`" construct (akin to "`sync`" in Cilk [3]) is a form of task synchronization, there is no data communication involved in it and it does not introduce issues such as deadlocks or livelocks. Semantically, any two tasks that join at the same program point and operate on different parts of the shared data structure are causally independent, and can be executed in any order.

We observe that the above programming model for delegated isolation is quite general and can be integrated with any imperative parallel language; in particular, we have implemented it on top of the Habanero Java [7, 18] language. Note that the underlying programming language may have other primitives for task creation that offer no guarantees of isolation—for example, in Habanero Java, there is a keyword "`async`" for creating such tasks without the `isolated` qualifier. Such tasks do not fall within the ambit of Aida, and are excluded from discussion in the rest of the paper.

Provided access to shared memory only happens from within blocks declared "`async isolated`," isolation and race-freedom are guaranteed in Aida. This is because the only objects accessed by an assembly A are those it owns. Deadlocks and livelocks are ruled out because when two assemblies conflict, one of them always delegates to the other, and the "merged" assembly continues to progress.

Extended example: Delaunay mesh refinement To see how the Aida programming and execution model work, let us consider a classic irregular application: *2D Delaunay mesh refinement* [10, 26]. Given a set of points in the 2D Euclidean space, a Delaunay triangulation partitions their convex hull into a set of triangles such that no point lies in any triangle's circumcircle. In many applications [10], there

```
1: Mesh m = /* read input mesh */
2: List wl = new List(m.getBad());
3: foreach Triangle t in wl {
4:   Cavity c = new Cavity(t);
5:   c.expand();
6:   c.retriangulate();
7:   m.updateMesh(c);
8:   wl.add(c.getBad());  }
```

Figure 2. Delaunay mesh refinement (sequential algorithm)

```
1: main () {
2:   finish {
3:    for (t in initial set of bad triangles)
4:         processTriangle(t); } }

5: processTriangle(t) {
6:    async isolated {
7:       if (t in mesh) {
8:          Cavity c = new Cavity(t);
9:          c.expand();
10:         c.retriangulate();
11:         for (s in c.badTriangles())
12:              processTriangle(s);  }
      }   }
```

Figure 3. Delaunay Refinement in the Aida programming model

are further qualitative constraints on the resulting triangles. To meet these constraints, a Delaunay triangulation often needs to be *refined*; triangles that do not satisfy this property are "bad." The *mesh refinement* problem is to retriangulate the mesh so that there are no bad triangles left.

It is a property of the application that a Delaunay retriangulation affects a "cavity": a local region in the mesh. Figure 2 shows a popular sequential algorithm that uses this property. Here, in each iteration of a loop, the algorithm builds a cavity `c` consisting of a bad triangle `t` drawn from a work list `wl` of bad triangles, *expands* `c` to the needed extent, and locally *updates* it using a re-triangulation routine. The Delaunay property is guaranteed to hold for the newly created triangles. The qualitative constraint may not hold for all the new triangles, so the size of the work list might increase in certain steps. The algorithm, however, is guaranteed to terminate [10].

In a parallel version of the algorithm, the isolation requirement is that `c.retriangulate()` executes atomically—logically, a cavity is a unit of isolation. In practice, cavities are almost always small, which means that a typical instance of the problem has a significant amount of available parallelism. At the same time, there is no a priori way here to tell if two parallel tasks carrying out retriangulations in parallel will *conflict*—i.e., if during cavity expansion, one task will demand access to a triangle under use by another. Conflict management, therefore, has to be dynamic.

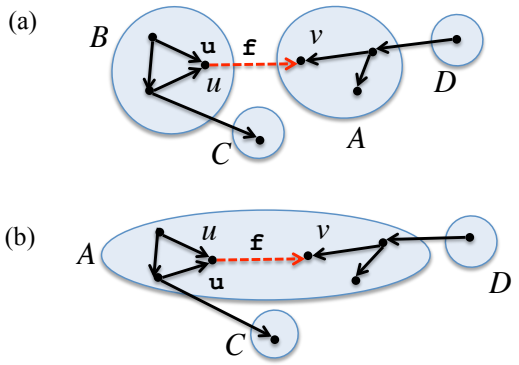


Figure 4. Effect of delegation on ownership

In the Aida approach to the problem, the programmer codes the application as in Figure 3. The main function is a sequential loop that calls the method `processTriangle` on each bad triangle. While executing a call to the method `processTriangle` on input t, the Aida runtime notices a block of code `K` marked as "`async isolated`". Such a task is executed by the Aida runtime in an isolated manner—i.e., as an assembly. Therefore, in the current scenario, a new assembly B is created. This assembly contains:

- An owned region R_B (initially empty)
- A queue Q_B, each element of which is a *closure*—a tuple consisting of a block marked "`async isolated`", as well as the local state of this block when it starts executing. Initially, the queue has a single closure, the code part of which is `K`, the body of the "`async isolated`" block for B.

The closure with code `K` is executed following standard sequential semantics—only, any new triangle accessed by `K` is first added to R_B. In the program on Figure 3, the first triangle added to R_B will be t. *This way, the assembly can only access objects in its owned region.* Unless there is a *conflict* (see below), B completes the execution of `K` and picks up the next closure from Q_B (if there is one)—the closure whose execution just completed is said to have been *committed*. The challenging scenario is when `K` tries to access a triangle in the owned region R_A of some other assembly A (or vice versa). In this *conflict* scenario, B *delegates* its work (all the closures in its queue, as well as the current closure) and region R_B to A. This means that from now on, R_B and Q_B are empty; the items formerly in these now belong respectively to R_A and Q_A (Figure 4). The assembly A must now also execute and commit the closures passed to it by B. Thus, delegation locally *coarsens* the granularity of parallelism in the heap.

Note that the closure that caused a conflict must be executed *from the beginning* at a subsequent time. As our language is imperative, this closure may, in general, already have performed modifications to the heap; therefore, Aida rolls back its effects.

$$
\begin{array}{lcl}
Prog & ::= & Prog; Prog \mid Asgn \mid \texttt{finish}\,\{Stmt\} \\
Asgn & ::= & \texttt{t} := \texttt{x.f}; \mid \texttt{x.f} := \texttt{t}; \mid \texttt{x} := \texttt{t}; \\
Stmt & ::= & [Asgn]^*\ [Async]^* \\
Async & ::= & \texttt{async isolated}\,\{Stmt\};
\end{array}
$$

where $\texttt{x}, \texttt{t} \in Var$, and $\texttt{f} \in F$.

Figure 5. Syntax of Core Aida. (For a nonterminal X, $[X]^*$ denotes zero or more successive occurrences of X.)

It is worth noting that for an important class of irregular applications that includes Delaunay refinement, all object acquisition happens before modification. In these cases, a need for rollback never arises (this class was identified by Pingali et al [30], who referred to such applications as *cautious applications*). In principle, we could use a static analysis [33] that identifies an application or assembly as cautious—given such an analysis, we could eliminate the runtime overhead of maintaining rollback logs in many cases (including every assembly created in Delaunay refinement). The current implementation of Aida does not employ any such optimization.

Also, the retriangulation triangle t may have been removed from the mesh by the time A gets to executing the delegated retriangulation. In this case, A simply skips it, and moves on to the next closure in Q_A (Line 7, Figure 3) — note that the sequential algorithm in Figure 2 does not need to perform the `(t in mesh)` test.

Finally, note that a block marked "`async isolated`" is in a sense an *asynchronously executed* transaction. The asynchrony entails that the recursive call inside the method `processTriangles` is not really a nested transaction; instead, it is a new asynchronously executed assembly.

2.1 Core Aida

Aida is a minimalistic model, and most of its features are covered by the preceding discussion. However, to understand its properties, we need to formalize the model to some degree. We do so using a foundational programming and execution model, henceforth referred to as *Core Aida*.

Core Aida uses the same concurrency constructs as our implementation of Aida; however, the former makes several simplifying assumptions about the sequential language underlying the model. For example, statements in Core Aida do not call methods on objects, create new objects, or declare new local variables. Also, we let objects be untyped and assume that all objects are shared.

2.1.1 Syntax

Aida is implemented on top of a framework for fork-join parallelism. In Core Aida as well as our implementation, the programmer creates assemblies by enclosing imperative code blocks within the construct "`async isolated` $\{\ldots\}$". The construct "`finish` $\{\ldots\}$" defines a scope at the end of which all assemblies created within the scope must join. We do not allow "`finish`"-blocks to be nested within "`async isolated`" blocks, thus following the "flat" model of "`finish`" statements studied in [2]. However, isolated asynchronous blocks can be nested to an arbitrary depth.

Formally, let us assume a universe Var of variable names and a universe F of field names. The syntax of programs in Core Aida is shown in Figure 5. Here, programs are given by the nonterminal $Prog$, and the code executed by an assembly is given by the nonterminal $Stmt$. We assume that all variables appearing in the text of a program P are implicitly declared at the beginning of P.

Note that $Stmt$ is defined such that an assembly must finish all its heap updates before it can create a new assembly. The semantic justification for this restriction will be provided soon.

2.1.2 Semantics

A Core Aida program P executes either in a sequential or a concurrent mode. Initially, it executes in the sequential mode—i.e., exactly like a sequential imperative program. Once control enters a "`finish`" scope in the program, P switches to the concurrent mode. At this point it is able to create new assemblies using the "async isolated" construct. These assemblies modify the heap asynchronously, possibly delegating to each other. Once all the assemblies created within the "finish" scope under consideration have terminated, P resumes execution in the sequential mode, from the control point following the "finish" block.

Heaps The central data structure in Core Aida is the shared-memory *heap*, which maintains the state of all shared mutable data accessed by a program. We abstractly view a heap as a directed graph whose nodes are objects, and edges are pointers labeled with *field names*. A *region* R in a heap G is a subset of the nodes of G.

Assembly states Now we will define a notion of state for an assembly created by a program P. Let us first define a *variable state* of P over a heap G. Such a state is a function μ that maps the variables of P either to objects in G, or to the special symbol `null`. We emphasize that an assembly is not required to own the objects to which its variables point. However, once it tries to read from or write to such an object u, it acquires ownership of u, or alternately delegates to the owner of u.

Consider a code block $\texttt{K} = \texttt{S}_1; \ldots; \texttt{S}_\texttt{m}$ that can be executed by an assembly (nonterminal $Stmt$ in Figure 5). We interpret such a block as a list $[\texttt{S}_1, \ldots, \texttt{S}_\texttt{m}]$. A *closure* of K is a triple $\langle \texttt{K}_1, \texttt{K}_2, \mu \rangle$, where $\texttt{K}_1$ and $\texttt{K}_2$ are the sublists of K that, respectively, have not and have been executed (we have $\texttt{K} = \texttt{K}_2; \texttt{K}_1$), and μ is a variable state of P.

Now, let us assume a universe of assembly IDs. The *assembly state* of an assembly that has ID A and works on a heap G is a tuple $N = \langle A, \langle \texttt{K}_1, \texttt{K}_2, \mu \rangle, R, Q \rangle$, where $\langle \texttt{K}_1, \texttt{K}_2, \mu \rangle$ is the closure currently being executed, R is the region of G currently owned by A, and Q is a list of closures

(DELEGATE)
$$\frac{N_1, N_2 \in \sigma \quad N_1 = \langle A_1, \langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1, Q_1\rangle \quad head(\mathtt{K}_1) \in \{(\mathtt{x.f} := \mathtt{t}), (\mathtt{t} := \mathtt{x.f})\} \quad N_2 = \langle A_2, \langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_2, Q_2\rangle \quad A_1 \neq A_2 \quad \mu_1(\mathtt{x}) = u \quad u \in R_2 \quad N' = \langle A_2, \langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_1 \cup R_2, append(Q_2, Q_1)\rangle}{\langle G, \sigma\rangle \longrightarrow \langle rollback(G, \mathtt{K}_1'), \sigma \setminus \{N_1, N_2\} \cup \{N'\}\rangle}$$

(LOCAL-ACCESS-1)
$$\frac{N \in \sigma \quad N = \langle A, \langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \quad head(\mathtt{K}_1) \in \{(\mathtt{x.f} := \mathtt{t}), (\mathtt{t} := \mathtt{x.f})\} \quad \mu(\mathtt{x}) = u \quad u \in R \text{ or } u \in \mathit{Free}(\sigma) \quad \mu \xrightarrow{head(\mathtt{K}_1)} \mu' \quad N' = \langle A, \langle tail(\mathtt{K}_1), \mathtt{K}_2; head(\mathtt{K}_1), \mu'\rangle, R \cup \{u\}, Q\rangle \quad G \xrightarrow{head(\mathtt{K}_1)} G'}{\langle G, \sigma\rangle \longrightarrow \langle G', \sigma \setminus \{N\} \cup \{N'\}\rangle}$$

(LOCAL-ACCESS-2)
$$\frac{N \in \sigma \quad N = \langle A, \langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \quad head(\mathtt{K}_1) \text{ has form } (\mathtt{x} := \mathtt{t}) \quad \mu \xrightarrow{head(\mathtt{K}_1)} \mu' \quad N' = \langle A, \langle tail(\mathtt{K}_1), (\mathtt{K}_2; head(\mathtt{K}_1)), \mu'\rangle, R, Q\rangle}{\langle G, \sigma\rangle \longrightarrow \langle G', \sigma \setminus \{N\} \cup \{N'\}\rangle}$$

(ASYNC-ISOLATED)
$$\frac{N \in \sigma \quad N = \langle A, \langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \quad head(\mathtt{K}_1) = \mathtt{async\ isolated\ \{\ K_3\ \}} \quad N' = \langle A, \langle tail(\mathtt{K}_1), \mathtt{K}_2; head(\mathtt{K}_1), \mu\rangle, R, Q\rangle \quad A'' \text{ is a fresh assembly ID} \quad \chi = \langle \mathtt{K}_3, \epsilon, \mu\rangle \quad N'' = \langle A'', \chi, \emptyset, \{\chi\}\rangle}{\langle G, \sigma\rangle \longrightarrow \langle G, \sigma \setminus \{N\} \cup \{N', N''\}\rangle}$$

(NEXT-CLOSURE)
$$\frac{N = \langle A, \epsilon, R, Q\rangle \in \sigma \quad N' = \langle A, second(Q), R, tail(Q)\rangle}{\langle G, \sigma\rangle \longrightarrow \langle G, \sigma \setminus \{N\} \cup \{N'\}\rangle}$$

(EMPTY-QUEUE)
$$\frac{N = \langle A, \epsilon, R, Q\rangle \in \sigma \quad Q = \emptyset}{\langle G, \sigma\rangle \longrightarrow \langle G, \sigma \setminus \{N\}\rangle}$$

(FINISH)
$$\frac{A \text{ is a fresh assembly ID} \quad \chi = \langle \mathtt{K}, \epsilon, \mu\rangle \quad head(\mathtt{K}') = \mathtt{finish\{K\}} \quad N = \langle A, \langle \chi, \emptyset, \{\chi\}\rangle\rangle \quad \langle G, \{N\}\rangle \longrightarrow \langle G', \{\emptyset\}\rangle}{\langle G, \mu, \mathtt{K}'\rangle \longrightarrow \langle G', \mu, tail(\mathtt{K}')\rangle}$$

(LOCAL-ACCESS-3)
$$\frac{head(\mathtt{K}) \in Asgn \quad \mu \xrightarrow{head(\mathtt{K})} \mu' \quad G \xrightarrow{head(\mathtt{K})} G'}{\langle G, \mu, \mathtt{K}\rangle \longrightarrow \langle G', \mu', tail(\mathtt{K})\rangle}$$

Figure 6. Operational semantics of Core Aida.

such that $head(Q)$ is the current closure, and $tail(Q)$ is the list of closures that A will execute in sequence once the current closure is committed.

Two assembly states N_1 and N_2 are *disjoint* if (1) they belong to different assemblies A_1 and A_2, and (2) the heap regions R_1 and R_2 referenced in N_1 and N_2 have no nodes in common.

Concurrent state Now we define the states of a Core Aida program during its concurrent phases. Such a state—a *concurrent state*—is a pair $\langle G, \sigma\rangle$, where G is a heap and $\sigma = \{N_1, \ldots, N_k\}$ for disjoint assembly states $N_1, \ldots, N_k$ in G. Thus, an object in our model is owned by at most one assembly. The set of objects in G that do not belong to the regions referenced in $N_1, \ldots, N_k$ are *free* in σ; we denote this set by $\mathit{Free}(\sigma)$.

Sequential state Next we define the states of a program during its sequential phases (in between `finish` scopes). Such a state—a *sequential state*—is a tuple $\langle G, \mu, \mathtt{K}\rangle$, where G is a heap, μ is a variable state, and $\mathtt{K}$ is a block of code whose syntax is given by the nonterminal Prog.

Transitions Finally, we present the transition relation $\longrightarrow$ over states that defines the operational semantics of Core Aida. The rules defining this relation are shown in Figure 6.

Here the rule DELEGATE is used to define delegation. In the premise, A_1 and A_2 are assemblies, respectively at states N_1 and N_2, and Q_1 is the list of closures that A_1 is obligated to execute. Further, the first element $head(Q_1)$ of this list is already under execution. The code for this closure is of the form $\mathtt{K}_1'; \mathtt{K}_1$. Of this code, $\mathtt{K}_1'$ has already been executed; the statement that is to be executed now is $head(\mathtt{K}_1)$.

However, the statement $head(\mathtt{K}_1)$ reads or writes an object u that is currently owned by the assembly A_2 (such an event is defined to be a *conflict*). Consequently, A_1 must now delegate its work and owned region to A_2. After the rule fires, the state of A_2 becomes N'. Note that the queue of A_2 is now $append(Q_2, Q_1)$.

One important issue is that A_1 may have modified certain objects in its region while it was executing $head(Q_1)$. In this case, before delegation, the runtime rolls back the effect of the code $\mathtt{K}'_1$. We denote by $rollback(G, \mathtt{K}'_1)$ the result of atomically applying this rollback on G.

Because an assembly is required to finish all its heap updates before it can create a new assembly, the code $\mathtt{K}'_1$ is solely a collection of heap updates—it does not include any "`async isolated`" statement. Its implementation is therefore straightforward. In the absence of this syntactic restriction, we would gain more parallelism. However, implementation of rollbacks would be much more difficult, as we would now have to "kill" the assemblies spawned by A_1.

The rule LOCAL-ACCESS-1 formalizes read and write accesses by an assembly A to objects not owned by others. The local state of A changes to μ' from μ on executing $head(\mathtt{K}_1)$ (this is denoted by $\mu \overset{head(\mathtt{K}_1)}{\longrightarrow} \mu'$); the heap changes from G to G' (this is denoted by $G \overset{head(\mathtt{K}_1)}{\longrightarrow} G'$). If the object is not already in A's region, it is added. However, there is no delegation.

The rule LOCAL-ACCESS-2 formalizes actions that update an assembly's variable state, but do not modify the heap.

The rule NEXT-CLOSURE shows what happens when an assembly finishes executing a closure (we denote a closure that has executed all its code by ϵ)—i.e., the closure is *committed* . It selects for execution the second item in its closure queue (the first item is the closure that just finished executing); if Q has just one element, then $second(Q)$ is defined to be ϵ.

The rule EMPTY-QUEUE says that when the queue of closures for an assembly becomes empty, the assembly can be removed from the concurrent state of the program. In essence, this rule specifies "garbage collection" of assemblies.

The rule ASYNC-ISOLATED defines the semantics of assembly creation. Intuitively, the rule creates a new assembly A' with an empty set of owned objects. The variables of the new assembly point to the same objects as in the parent assembly.

The rule FINISH defines the semantics of "`finish`"-scopes. Suppose the program enters such a scope from a sequential state $\langle G, \mu, \mathtt{K} \rangle$. The program now moves to a concurrent phase where a single assembly is executing. This assembly has variable state μ, owns no objects, and executes the code enclosed within the "finish" block. Of course, this assembly can subsequently spawn many other assemblies through the "`async isolated`" construct.

Finally, the rule LOCAL-ACCESS-3 gives semantics to the assignments executed by a program during its sequential phases.

Note that in the above semantics, we assume that delegation happens atomically. At the level of the implementation, this atomic delegation step could have been built using straightforward, coarse-grained locking. This would make the atomicity of delegation obvious. However, to achieve greater performance, our implementation of delegation uses a more sophisticated fine-grained locking approach with rollback.

Section 3 describes this implementation and explains in detail how delegation is handled in all conflict scenarios. The key points are that all assemblies in the system are ordered and can be locked, and locks on assemblies are obtained in order. Therefore, if assembly (with ID) A tries to delegate to assembly B just as B is trying to delegate to A, both A and B will attempt to obtain locks on A and B in the same order, and only one will succeed.

Executions and termination A Core Aida execution starts from an initial sequential state $\pi_0 = \langle G, \mu, \mathtt{K} \rangle$. A *terminating state* of a Core Aida program is a sequential state $\pi_n = \langle G, \mu, \emptyset \rangle$ from which no transition is possible. Let $\longrightarrow^*$ be the transitive closure of the relation $\longrightarrow$. A Core Aida program is said to *terminate* if for initial states π_0, there is a terminating state π_n such that $\pi_0 \longrightarrow^* \pi_n$.

Properties of Core Aida

Now we discuss the safety and liveness guarantees that Core Aida offers.

Isolation The property of isolation demands that a concurrent task read or write shared-memory objects without interference from other tasks. In Core Aida, an assembly can only read or write objects in its own region; also, if A_1 delegates work to A_2 (rule DELEGATE in Figure 6), the ability of A_2 to read or write its objects is not compromised. Therefore, Core Aida guarantees isolation. In our implementation of the model, the guarantee of isolation holds for all memory accesses performed from within "`async isolated`" blocks; as in transactional memory systems, we do not offer guarantees on memory accesses made outside blocks declared to be isolated.

Deadlock-freedom There are only two synchronization operations in (Core) Aida: scoped joining of assemblies and delegation. As the former operation does not depend on data at all, it clearly does not introduce deadlocks. As for delegation, the only plausible deadlock scenario involving *two* assemblies is the following: "Assembly A tries to delegate to assembly B, B tries to delegate to A, and neither can progress." This scenario, however, is impossible in Aida. If A and B try to simultaneously delegate to each other, then one of the two requests (let us say the one from B) will be nondeterministically selected and honored. Let u be the object that A needed to access in the statement that triggered the delegation request from A to B. After B delegates to A, A will have all objects that B previously owned. Therefore, the request from A will no longer be a conflict—the request from A to access u will succeed, and A will be able to progress.

It is easy to generalize the above argument to cyclic deadlocks between n assemblies. Of course, while our semantics

for Core Aida is deadlock-free, we have to ensure that our *implementation* of Aida semantics is deadlock-free as well. We do so in Sec. 3.

Livelock-freedom In Core Aida, two assemblies A_1 and A_2 would be livelocked if they constantly try to delegate to each other, none of them progressing. As an analogy, there is always a non-zero probability that such a livelock scenario may occur in a transactional memory system with repeated rollback-retry operations. However, in such a scenario, Aida would destroy one of the two assemblies, delegating its work to the other—the other assembly would then be able to progress.

Bound on conflicts/commit ratio Finally, a key property of Core Aida is that in any execution, the number of conflicts (number of delegation steps, or application of the rule DELEGATE in Figure 6) is bounded by the number of commits (the number of applications of the rule NEXT-CLOSURE). This property works as a sort of performance guarantee in high contention-scenarios, where, in many state-of-the-art systems, there may be too many aborted tasks and too few commits. As we show in Section 4, this property lets us achieve better performance in high-contention scenarios than other approaches to irregular parallelism, such as GALOIS and software transactional memory (with DSTM2 as an example). On the other hand, the growing ownership region created by successive delegations may induce spurious conflicts. Consider three assemblies A_1, A_2 and A_3 in a scenario where A_1 is long running and has some object x in his ownership region, A_2 has object y in his ownership region and requests object x causing A_2 to be delegated to A_1. When A_3, a short running assembly, requests access to y it will be delegated to A_1 as well. Note that A_1 and A_3 could be run concurrently but they are serialized due to the spurious conflict. A possible heuristic solution to this scenario is to terminate a delegation sequence and release the ownership region after some predefined number of assemblies have executed. On the other hand, the growing ownership region created by successive delegations may induce spurious conflicts as exemplified in the following scenario. Consider three assemblies A_1, A_2 and A_3 in a scenario where A_1 is long running and has some object x in its ownership region, A_2 has object y in its ownership region and requests object x causing A_2 to be delegated to A_1. When A_3, a short running assembly, requests access to y it will be delegated to A_1 as well. Note that A_1 and A_3 could be run concurrently but they are serialized due to the spurious conflict. These spurious conflicts would provoke unnecessary retries. To the extent that this phenomenon might happen, it does not seem to affect the current set of benchmarks.

3. Implementation

A key aspect of our model, that presents challenges not present in other concurrency models, is the fact that ownership changes operate on sets of objects rather than on the individual objects themselves. This is illustrated in a conflict scenario, where an assembly will delegate its code as well as transfer its region of ownership to the assembly it has a conflict with. We use a parallel version of a union-find data structure for disjoint sets [1, 16] to implement this ownership transfer efficiently.

```
1  Method Acquire, Input : Object obj
2    objAssembly = obj.owner();
3    currAssembly = this.owner();
4    if objAssembly == currAssembly then go to 13;
5    if acquireObject(obj, currAssembly) then go to 13
     // Conflict! Delegate to other assembly
6    locks = orderedLock (currAssembly, objAssembly);
     // If anything changed, retry
7    if
     (objAssembly ≠ obj.owner() ∨ objAssembly.dead() ∨
     currAssembly ≠ this.owner()) then
8      locks.release(); go to 2;
9    rollback (); // rollback the assembly's state
10   delegateTo (objAssembly);
11   locks.release();
12   die ();
13   if (write operation ∧ obj ∉ writeSet) then
14     rollbackLog.add(obj.copyValues());
15     writeSet = writeSet ∪ {obj}
```

Figure 7. Algorithm for acquiring an object

Figure 8 summarizes the main abstractions in the Aida runtime: shared objects, assemblies, and isolated code.

Here, the "`Assembly`" objects represent a collection of "`IsolatedCode`" blocks that represent all the tasks for whose execution the assembly is responsible. When created using the Aida "`async isolated`" construct, the "`Assembly`" object will contain an empty region and a single task in the queue. As the program progresses, assemblies either accumulate data and tasks, finish their execution and terminate, or delegate their data and tasks to another assembly and terminate.

During the execution of an "`async isolated`" block, the runtime needs to perform ownership checks when accessing shared objects. These checks are performed by calling "`Object.acquire()`" whose pseudocode is shown in Figure 7. Note that a given "`async isolated`" block needs to acquire a particular shared object only once, the first time the object is accessed.

The algorithm for "`Object.acquire`" ensures that the check for object ownership and any delegation is done *atomically* with respect to the other assemblies in the system. Lines 6 and 7, discussed below, are critical for guaranteeing this atomicity.

"`Object.acquire`" first checks if the assembly that is executing the code already owns the object, in which case

```
class hj.lang.Object {
    // owner is only changed using CAS
    private Assembly owner;

    // unowned objects are acquired using this method
    public boolean acquireObject(Assembly newOwner);

    // called on the first access to this object by the current owner
    public acquire();

    // get the owner by navigating the union−find data structure
    public Assembly owner();

    // methods for rollback and restore
    public Copy copyValues();
    public void restoreValues(Copy v);
}

class hj.runtime.wsh.Assembly {
    private Assembly parent;  // parent pointer in the union−find tree
    long rank;      // for union by rank

    private long id; // a unique assembly id

    // a queue of delegated isolated code
    private Queue<IsolatedCode> delegatedCodeQueue;

    public Assembly find(); // find the representative assembly

    // An assembly that finished the execution of all its tasks
    // Free objects are represented by belonging to a dead assembly
    public boolean dead();

    // implements the delegation through union
    public void delegateTo(Assembly to);

    // next IsolatedCode to run
    public IsolatedCode next();

    // acquire assemblies a1 and a2 in order
    static Locks orderedLock(Assembly a1, Assembly a2);
}

class hj.runtime.wsh.IsolatedCode {
    // points to an Assembly in the union find structure
    private Assembly owner;

    // get the owner by navigating the union find data structure
    public Assembly owner();

    // holds information on how to restore objects on roll back
    private List<Copy> rollbackLog;

    // actual code from the user program is run through this method
    public void run();

    // the commit action, done in constant time
    public void commit();

    // restore all accessed object to their pre−execution state
    public void rollback();
}
```

Figure 8. Principal abstractions in the Aida implementation

it can continue execution (adding the object to its write set if necessary) on line 13. The assembly then tries to acquire the object (line 5), which will succeed if the object is not currently owned by any other assembly. If successful, the assembly can continue (line 13).

Otherwise, there is a conflict and the assembly needs to delegate itself to the one that holds the ownership of the object. It first acquires the locks on the current assembly and on the assembly that owns the desired object (line 6). To avoid deadlock, all assemblies that are created in the program have a unique ID and the locks are always acquired in order.

Upon acquiring the locks, the assembly checks on line 7 if anything has changed in the meantime. The assembly that owned the object might have delegated its tasks to another assembly and transferred the ownership of its objects, the assembly that owned the object might have finished the execution of all its activities and died (transferring the ownership of its objects to the unowned object pool) or the representative for the current assembly might also have changed (a consequence of a requirement for the performance guarantees in the disjoint set data structure, delegations can result in the representative for the current assembly to change). If any of that has happened both assemblies are released and the process of acquiring the object is retried.

If nothing has changed, the assembly proceeds with the delegation process. It first rolls back all the changes it has made to the objects in its $writeSet$, then performs the delegation, which copies all the `IsolatedCode` tasks from the currently assembly's queue to the queue of the object's owner. This is done in constant time by concatenating the delegation queues together. The delegation process also transfers the ownership of all the objects owned by the current assembly to the assembly it is delegation to. The assembly then releases the locks and dies (returns the execution to the runtime).

As already mentioned in Section 2, if all the accesses to the shared memory in the program are done only inside the isolated blocks, our implementation of the isolated construct using assembly delegation as described in the algorithm in Figure 7 guarantees isolation and race freedom.

We use a union-find data structure to track object ownerships. This allows us to perform near-constant time object ownership checks, and constant time ownership transfers when an assembly delegates itself to another one, as well as constant time releases of all the objects owned by the assembly when the assembly finishes its execution.

Figure 9 show the union-find data structure we use to track the ownership of objects. Each object has an $owner$ reference, which points to the assembly that owns it. Assemblies merged by delegation are represented as a tree with the root being the representative assembly. Assemblies also have a $parent$ reference that points to their parent in this tree. On Figure 9, object C is owned by assembly 3, while objects A

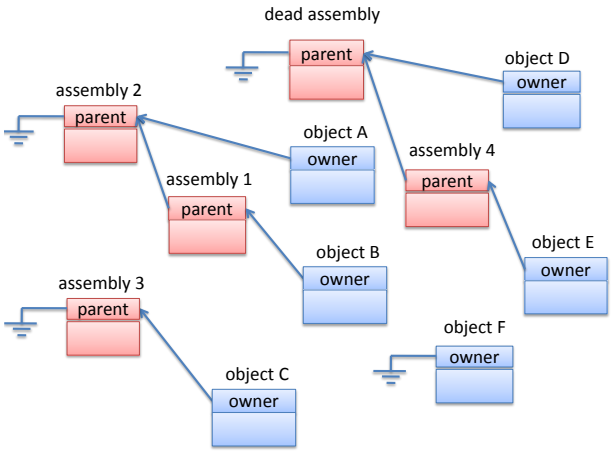


Figure 9. Union find data structure to track object ownerships

and B are owned by assembly 2. When an assembly delegates to another assembly, it transfers the ownership of its objects, which will result in setting either its *parent* reference to point to the assembly it delegated to or vice-versa, depending on the shapes of their union-find trees.

To efficiently keep track of all the objects that have been released, assemblies that have finished execution are marked as dead. An object is free if its *owner* reference is *NULL* (which will be the case if it has never been acquired by anyone), or if the *find* algorithm determines that its owner is dead (which will be the case if it has been acquired by some assembly and then the assembly or its delegate has finished execution). In Figure 9, objects F, D and E are not owned by anyone.

Assemblies that are not at the root of the union-find tree are eventually garbage collected when no objects point to them. References to these assemblies are removed by *find* since it is performed with path compression. Assembly 1 in Figure 9 will become unreferenced when, for example, a *find* is performed on object B.

4. Evaluation

4.1 Experimental Setup

We have used two platforms for our experimental evaluation: (a) a 64-way (single-socket, 8 cores, 8 threads per core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10 and Sun JDK 1.6 64-bit version; (b) 16-way (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory and running Red Hat Linux (RHEL 5) and Sun JDK 1.6 64-bit version. All benchmarks were allocated 3.5 gigabytes in the JVM heap with the exception of DSTM2 that needed a 15G heap to

```
1:  start(nodes){
2:    finish {
3:      for(n in nodes)
4:        mergeMinEdge(n);
5:    }
6:  }

7:  mergeMinEdge(c) {
8:    async isolated {
9:      c' = c.getMinEdge();

10:     // merge c into c'
11:     c'.addEdges(c);

12:     // change x -> c into x -> c'
13:     // change c -> x into c'-> x
14:     // remove c -> c' and c -> c'
15:     c'.fixEdges(c);
16:   }
17: }
```

Figure 10. Boruvka Minimum Spanning Tree pseudocode

```
1:  start(nodes){
2:    finish {
3:      for i = 1 to 40000 // 40.000 concurrent tasks
4:              task()
5:    }
6:  }

7:  task() {
8:    async isolated {
9:      for i = 1 to 20 {//  20 operations per task
10:       v = *;
11:       do *  => insert(v)   //  5%
12:          *  => delete(v)   //  5%
13:          *  => contains(v) // 90%
16:   } } }
```

Figure 11. HashTable and RBTree pseudocode

complete. Each benchmark was run 7 times within the same JVM process for each platform/programming model. The results reported are the average of the later 6 runs, excluding the first run to factor out just-in-time compilation effects.

We evaluate our approach on a set of four different benchmarks described below.

Boruvka minimum spanning tree. We consider a classical algorithm for computing the minimum spanning tree of an undirected graph [23]. The main idea is to proceed by repeatedly fusing nodes through their minimal weight edge. Each node u obtained by such fusion represents a set of nodes S_u in the original graph; we associate with u a minimal spanning tree for the subgraph induced by S_u. As the algorithm progresses it builds larger and larger minimum spanning trees, one for each node remaining in the graph, until only one node remains.

In the parallel version of the algorithm, initially a parallel task is spawned for each node in the graph. As they grow their local minimal spanning trees, the tasks need to merge their local tree with the tree of their minimal weight edge neighbor. This algorithm starts with the maximum parallelism available and the parallelism decreases as the tree is built. For evaluation we used a graph representing the road network for the state of New York from [11]. The road network graph consists of 264,346 nodes and 733,846 edges. A pseudocode for the algorithm is shown in Figure 10. This benchmark is part of the Lonestar Benchmark Suite [27].

Delaunay mesh refinement. As mentioned earlier, this algorithm refines a 2D Delaunay mesh so that each of the triangles satisfy a quality constraint. In our evaluation, we start with a triangle mesh consisting of 100,770 triangles of which 47,768 do not satisfy the quality constraint. This benchmark and the dataset used are part of the Lonestar Benchmark Suite [27].

Hash table and Red-black tree. These benchmarks are based on the red-black tree and hash table benchmark implementations in DSTM2 [12], and are used to evaluate scenarios where several clients access shared data structures concurrently. The pseudocode for the algorithm used in the benchmarks is shown in Figure 11. In these benchmarks, we model 40,000 concurrent tasks performing 20 operations each in an isolated manner. Operations are chosen at random with a 90% chance of an operation being a query, 5% being an insert and 5% being an update. The values are also chosen at random. In the RBTree benchmark operations are performed on a red-black tree, while in HashTable they are performed on a 256-bucket hash table.

Methodology. We compare our approach with several alternate parallel programming models :

- **Chorus**: As mentioned earlier, Aida's object assembly abstraction is borrowed from Chorus [28]. However, the Chorus model is more restrictive than Aida because it is only applicable to parallelization of *cautious* applications [30] and it lacks Aida's notion of delegated isolation. The Chorus results reported in this paper were obtained from the benchmark implementations used in [28].
- **DSTM2:** DSTM2 [12] provides an implementation of dynamic software transactional memory. The red-black tree and hash table benchmarks are provided in the DSTM2 distribution. We also ported the Chorus versions of Delaunay mesh refinement and Boruvka minimum spanning tree to DSTM2.
- **HJ (SEQ)**: The sequential baseline versions of all benchmarks were implemented as single-task Habanero-Java [7, 18] programs, using the Aida data structures and runtime system.
- **HJ (CGL):** The Coarse Grained Locking (CGL) versions of the benchmarks using the same Habanero-Java source code as the Aida versions, but with the use of a single lock to implement HJ's "`isolated`" construct. Coarse-grained locking is the default approach for implementing isolation in the current HJ release [18].
- **Java (FGL)** The Java-based fine-grained locking implementations of the Delaunay and Boruvka benchmarks were the same as those used for comparison with Chorus in [28]. In these implementations, several worker threads obtain work units from a global first-in-first-out work list, and attempt to acquire all needed shared objects for a work unit. If an attempt fails, the work unit is returned to the work list. This version cannot be implemented in HJ because HJ does not support explicit locking.
- **Galois**: In the Galois programming model [17], algorithms are expressed as sequential operations on unordered work lists. The runtime executes the loop bodies in parallel, and through annotations the programmer can specify locking and logging policies as well as many policy options for the implementation of the abstract work list. The Delaunay mesh refinement and Boruvka minimum spanning tree benchmarks used in this evaluation were obtained from the Galois distribution. The red-black tree and hash table benchmarks where implemented by us for this paper, using the graph data structures from the Galois distribution.

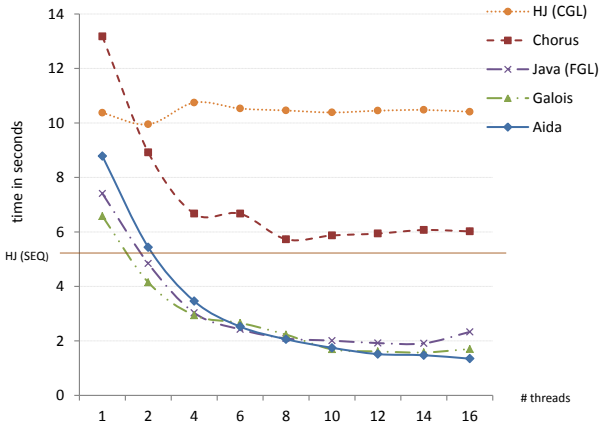


Figure 12. Delaunay mesh refinement on a 16-core Xeon (Note: For reasons of scale DSTM2 is left out of the plot. DSTM2 -1 thread: 962s, 16 threads: 177s)

4.2 Experimental Evaluation

A comprehensive collection of all timing results is shown in Table 1. We extracted some of the interesting trends from the raw data, and plotted them in the graphs discussed below.

Execution times Figure 12 and Figure 13 show the execution times for Delaunay mesh refinement on the 16-core Xeon and the 64-way UltraSPARC T2 respectively. Aida

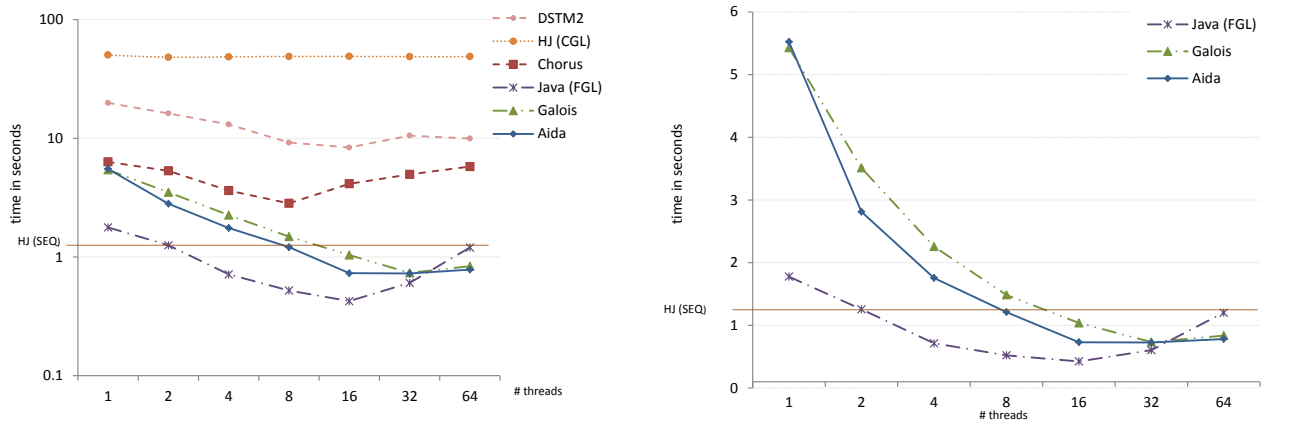
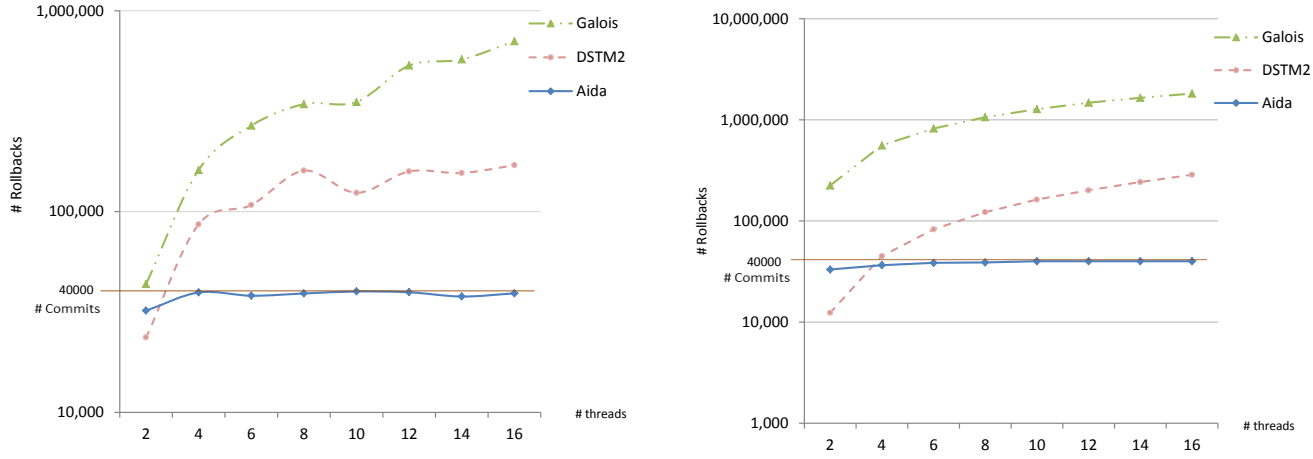


Figure 15. Boruvka minimum spanning tree on a 64-way UltraSPARC T2 (left: logarithmic scale; right: inset of the bottom 6 seconds)



(a) RBTree

(b) HashTable

Figure 16. Impact of speculative execution

shows a speedup of 3.9× over sequential HJ at 16 threads compared to the next best approach which is Galois at 3.3× over sequential HJ at 14 threads (where Galois has a maximum speedup) on the Xeon architecture. The self-relative speedups are 6.5× for Aida and 4.2× for Galois. On the UltraSPARC T2 architecture, Aida peaks at 12.2× speedup against sequential HJ at 64 threads whereas Galois again is the next best and peaks at 12× with 64 threads.

Figure 14 and Figure 15 show the performance numbers for Boruvka minimum spanning tree on the on the 16-core Xeon and the 64-way UltraSPARC T2 respectively. Aida shows a speedup of 1.1× over sequential HJ at 16 threads being the third best approach bettered by Java fine-grained locking at 1.5× (at 10 threads) and also by Galois at 1.4× (at 14 threads). The self-relative speedups are 3.4× for Aida, 2.6× for Java FGL and 2.4× for Galois. On the UltraSPARC T2 architecture, on the other hand, Aida peaks at 1.74× speedup against sequential HJ at 32 threads whereas Galois is at 1.72× at 32 threads and both are bettered by Java FGL at 2.9× at 16 threads. The self-relative speedups are 7.6× for Aida, 7.4× for Galois and 4.23× for Java FGL.

The speedups and performance numbers indicate that Aida is a very competitive approach for parallelizing irregular applications, especially when compared with other JVM based systems. The reduced speedup of Aida and Galois over Java fine-grained locking on Boruvka tree is due to the overhead involved in keeping track of ownerships, conflicts and the maintenance of rollback logs, compared to a much lighter-weight implementation in Java using fine-grained locks. In Boruvka, the transactions perform very

Benchmark	Model	16-core Xeon					64-way UltraSPARC T2						
		1	2	4	8	16	1	2	4	8	16	32	64
RBTree	Aida	0.48	0.48	0.50	0.55	0.51	2.81	4.05	4.65	5.50	5.92	5.208	6.867
	Galois	8.07	7.49	7.84	8.38	10.06	89.24	86.32	87.01	82.89	85.89	87.34	87.85
	DSTM2	4.25	4.64	7.32	11.11	11.40	9.18	18.16	30.78	16.38	35.68	43.87	44.96
HashTable	Aida	1.40	0.64	0.55	0.55	0.53	22.40	19.92	24.27	23.78	24.94	24.93	24.31
	Galois	33.95	33.95	34.46	33.42	35.52	169.97	135.93	125.48	124.15	124.152	123.05	127.04
	DSTM2	0.51	1.042	1.24	1.66	2.23	2.877	2.90	.64	2.78	3.39	5.08	6.26
Delaunay Mesh Refinement	HJ (SEQ)	5.27					23.87						
	Aida	8.78	5.43	3.46	2.06	1.34	25.43	15.77	8.90	4.48	2.72	2.09	1.47
	Galois	6.58	4.15	2.94	1.77	1.70	25.27	15.76	8.38	4.58	2.70	2.11	1.88
	Java (FGL)	7.41	4.84	3.03	2.09	2.33	26.11	17.83	10.44	6.40	4.96	4.88	4.95
	Chorus	13.18	8.92	6.67	5.73	6.02	44.98	35.82	24.65	21.19	20.15	19.87	21.02
	HJ (CGL)	10.37	9.02	10.74	10.46	10.41	50.30	48.23	48.58	48.99	49.00	48.78	48.83
	DSTM2	1058.45	279.96	243.41	205.68	181.60	792.34	425.39	286.07	193.43	136.84	110.54	85.92
Boruvka Minimum Spanning Tree	HJ (SEQ)	0.64					1.26						
	Aida	1.98	1.20	0.91	0.70	0.58	5.52	2.81	1.75	1.21	0.73	0.72	0.78
	Galois	1.09	0.83	0.73	0.69	0.57	5.42	3.51	2.26	1.49	1.04	0.74	0.87
	Java (FGL)	1.18	0.82	0.57	0.47	0.52	1.78	1.26	0.71	0.52	0.42	0.61	01.20
	Chorus	2.16	1.52	1.09	0.82	0.95	6.33	5.32	3.63	2.83	4.14	4.97	5.78
	HJ (CGL)	5.45	5.56	6.49	8.08	10.27	37.74	37.30	37.75	38.14	41.19	46.32	50.25
	DSTM2	4.821	6.453	4.64	3.58	2.73	19.89	16.21	13.09	9.22	8.37	10.53	9.97

Table 1. Benchmark results (time in seconds).

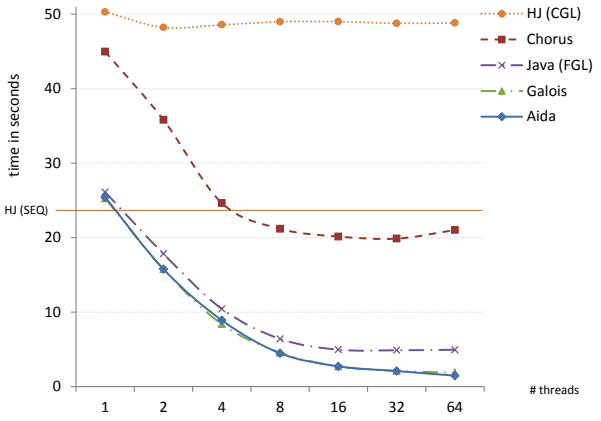


Figure 13. Delaunay mesh refinement on a 64-way UltraSPARC T2 (Note: For reasons of scale DSTM2 is left out of the plot . DSTM2 -1 thread: 795s, 64 threads: 81s)

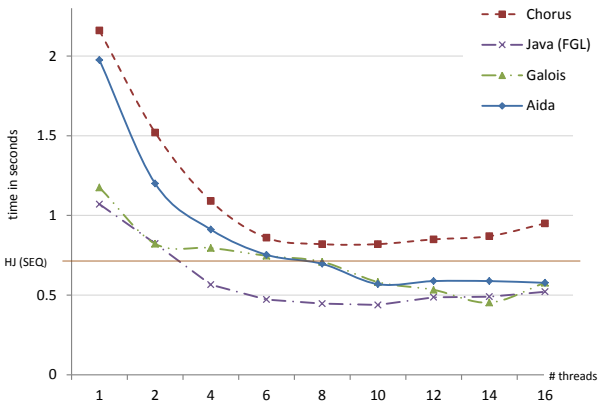


Figure 14. Boruvka minimum spanning tree on a 16-core Xeon (Note: For reasons of scale HJ (CGL) and DSTM2 are left out of the plot HJ (CGL)- 1 thread: 5.5s, 16 threads: 10.3s, DSTM2- 1 thread: 4.8 , 16-threads:2.7.s)

little work, so the relative impact of the bookkeeping overhead is very high, as evidenced by the difference between self-relative and sequential-relative speedups.

Aida is one orders of magnitude faster than both Galois and DSTM2 on the RBTree benchmark and one to two orders of magnitude faster than Galois on the HashTable benchmark. Since the Galois implementation of these two benchmarks was straightforward and was done using minimal programming effort (about the same effort that is required to write these benchmarks in DSTM2 or Aida), we suspect that a more careful implementation using more annotations and done by a Galois expert might yield better results than what we have obtained in Table 1. Either way, we feel that these two benchmarks are much better suited to illustrate the effects of speculation discussed below, than for raw speed comparisons.

Aborted speculative work. DSTM2, Galois and Aida, like many approaches to parallelization and concurrency, rely on speculative execution to improve performance. However, in scenarios with relatively high contention, speculative execution might hinder performance and resource usage. Figure 16 compares the impact of speculative execution for Aida, Galois and DSTM2 in terms of the numbers of commits and aborts for a given workload. Aida offers a lower upper bound on the number of aborted speculative tasks compared to most transactional memory implementations. As discussed in Section 2, Aida guarantees that there will be only as many aborted speculative tasks as there are successful ones,

whereas DSTM2 only bounds the number of aborted tasks in relation to the total number of shared objects. Galois provides a probabilistic progress guarantee based on exponential back-off for conflict resolution. Figure 16(a) and Figure 16(b) show DSTM2 having an order of magnitude and Galois having two orders of magnitude more aborts than Aida.

We have shown the HashTable and RBTree benchmark results with a 90% read-only and 10% write operation mix, which is generally considered to be a realistic medium-to-high contention scenario [14, 34]. We also tested even higher contention scenarios, where as expected Aida fared even better compared to Galois and DSTM2.

Overhead. Our performance numbers indicate that the sequential overhead of our Aida implementation is similar to that of Galois. We note that as the work performed in isolation increases, the relative overhead decreases, evidenced by the different single thread slowdowns with respect to a sequential code.

Programmability Figure 17 and Figure 18 contrast the code for Delaunay Mesh Refinement in Aida and Galois to show the differences in programmability between these two approaches. Galois uses three different types of annotations: annotations to denote the scope of the parallel loop, performance oriented hints and annotations regarding task scheduling policies. Lines 1-4 in In the aforementioned example define the scope of the parallel loop. Galois provides three performance oriented hints to annotate each access to a shared data structure: `NONE`, `CHECK_CONFLICT` and `SAVE_UNDO`. By default in each access to the shared data structure the runtime checks whether the nodes are being in use by a concurrent tasks, and saves state information in a log to provide for an eventual undo. The `NONE` hint instructs the runtime not to perform any check nor save undo data. This flag reduces most of the overhead and is used in lines 13, 18, 19, 28 and 30. The `CHECK_CONFLICT` hint instructs the runtime not to save undo data and is used on line 5. Finally the annotation at line 31-32 instructs the runtime to group activities into chunks and use a global FIFO queue for the chunks, and a local LIFO stack for tasks in a chunk. Aida, in contrast, requires only one declarative annotation with respect to sequential Java code which is the fragment "`async isolated`" in line 2 to denote the scope of the isolated asynchronous task.

5. Related Work

We use Table 2 to guide the discussion in this section. This table qualitatively classifies programming models according to their attributes in the following dimensions:

- *Programmability/expressiveness:* how easy is it to express a wide range of parallel programming patterns in the model?

```
 1: void doCavity(Triangle start) {
 2:   async isolated {
 3:     if (start.isActive()) {
 4:       Cavity c = new Cavity(start);
 5:       c.initialize(start);
 6:       c.retriangulate();

          // launch retriagnulation on new bad triangles.
 7:       Iterator bad = c.getBad().iterator();
 8:       while (bad.hasNext()) {
 9:         final Triangle b = (Triangle)bad.next();
10:         doCavity(b);
          }

          // if original bad triangle was NOT retriangulated,
          // launch its retriangulation again
11:       if (start.isActive())
12:         doCavity(start);
       }
     } // end isolated
   }

13: void main() {
14:   mesh = ... ; // Load from file
15:   initialBadTriangles = mesh.badTriangles();
16:   Iterator it = initialBadTriangles.iterator();
17:   finish {
18:     while (it.hasNext()) {
19:       final Triangle t = (Triangle) it.next();
20:       if (t.isBad())
21:         Cavity.doCavity(t);
22:     }
19:   }
20: }
```

Figure 17. Actual code for Delaunay Refinement in the Aida programming model

- *Correctness guarantees:* does the model provide correctness guarantees such as deadlock-freedom, livelock-freedom, and progress guarantees?
- *Scalability:* how well does performance scale with an increasing the number of processor cores and hardware threads?
- *Overhead:* how much overhead does the model impose relative to a sequential implementation?

Space limitations prevent us from discuss a larger set of parallel programming models in the table due to the vast amount of past work in this area. Instead, we include a few representative examples for each distinct set of attributes, and trust that the reader can extrapolate this discussion to other programming models with similar attributes in these dimensions.

Aida provides a high-level minimalistic programming model similar to *Transactional Memory* [19], with a single construct (`async isolated`) to define blocks of code to be executed concurrently and in isolation. Aida guarantees livelock-freedom, unlike transactional memory where livelock freedom is probabilistic and implementation-dependent. One of the main objections to software implementations of

Parallel prog. model	Expressiveness	Expertise	Safety	Scalability	Overhead
STM [19]	High	Low	implementation-dependent	Poor	High
Java with fine-grained locking	High	High	Very poor	High	Low
OpenMP reductions [8]	Medium	Low	Poor	High	Low
Cilk reducers [3]	Medium	Medium	Poor	High	Low
Chorus [28]	Medium	Low	High	Medium	Medium
Galois [5, 25, 26]	Medium	Low	Medium	High	Low
Aida [this paper]	High	Low	High	High	Low

Table 2. Comparison of several parallel programming models.

```
   // The parallel loop
 1: GaloisRuntime.foreach(badNodes,
 2:   new Lambda2Void<... >() {
 3:   public void call(GNode<Element> item,
 4:     ForeachContext<GNode<Element>> ctx) {

 5:      if (!mesh.contains(item, MethodFlag.CHECK_CONFLICT))
 6:         WorkNotUsefulException.throwException();

 7:      Cavity cavity = new Cavity(mesh);
 8:      cavity.initialize(item);
 9:      cavity.build();
10:      cavity.update();

        //remove the old data
11:      List<...> preNodes = cavity.getPre().getNodes();
12:      for (int i = 0; i < preNodes.size(); i++)
13:        mesh.remove(preNodes.get(i), MethodFlag.NONE);

        //add new data
14:     Subgraph postSubgraph = cavity.getPost();
15:     List<...> postNodes = postSubgraph.getNodes();
16:     for (int i = 0; i < postNodes.size(); i++) {
17:        GNode<Element> node = postNodes.get(i);
18:        mesh.add(node,  MethodFlag.NONE);
19:        Element element = node.getData( MethodFlag.NONE);
20:        if (element.isBad())
21:           ctx.add(node,  MethodFlag.NONE);
        }
24:     List<...> postEdges = postSubgraph.getEdges();
25:     for (int i = 0; i < postEdges.size(); i++) {
26:       ObjectUndirectedEdge<...> edge = postEdges.get(i);
27:       mesh.addEdge(edge.getSrc(), edge.getDst(),
28:                    edge.getData(),  MethodFlag.NONE);
        }
29:     if (mesh.contains(item, MethodFlag.NONE )) {
30:        ctx.add(item,  MethodFlag.NONE);
        }
      }
31: }, Priority. first(ChunkedFIFO.class)
               .thenLocally(LIFO.class)) ;
```

Figure 18. Delaunay Refinement in the Galois programming model

transactional memory has been their inability to deliver competitive performance [6]. Aida on the other hand, performs on par with Galois, perhaps the most successful existing approach to irregular data parallelism.

Our implementation approach has some similarities to *Recursive and Non-Recursive Helping* in wait-free and lock-free implementations of certain nonblocking algorithms [15, 29]. In recursive helping, if a transaction A detects a conflict with transaction B, then A *helps* B by making loads, stores and atomic operations on its behalf, or recursively helps a transaction C that is currently being helped by B, and so on. The Aida approach is far more aggressive: assembly A transfers all the objects it owns to assembly B and delegates its execution to B. This idea is vaguely similar to the "Work Dealing" concept introduced by Hendler and Shavit [21]. While helping requires all transactional objects to be in a total order, Aida only requires the assemblies to be ordered. Helping can also be much more heavyweight than Aida because the bound on the total number of aborts and restarts in the system is much higher.

There is some past work on delegation in the presence of contention that is relevant to the delegated isolation concept introduced in Aida. Oyama et al. [31] protect the data structure with a single lock and have threads form a list of requests on the lock. The thread acquiring the lock services the pending requests of others in LIFO order, and afterwards removes them from the list. Flat combining by Hendler et al. [20] and CC-BSim [13] allow a single thread that acquires a global lock on a shared data structure to learn about all concurrent access request to it, then perform the combined access of all other threads to the structure. None of these approaches provide the simple, general and high-level programming interface of Aida.

Fine-grained Locking has long been the only method for achieving high performance in irregular parallel applications. It is only usable by expert programmers. Programmability is very poor, and there are no correctness guarantees. The inadequacy of non-expert programmers to express complex concurrent algorithms using fine-grained locking is the main reason for the proliferation of higher-level parallel programming models.

Programming models such as OpenMP [8] and Cilk [3] provide efficient support for *reductions* in deterministic parallel programs, but those constructs are not applicable to the

nondeterministic, irregular parallelism supported by Aida and the other models listed in Table 2.

The *assemblies* in the *Chorus* programming model [28] provide a high-level mechanism for irregular data parallelism and have been the main inspiration for assemblies in Aida. The expressiveness of Chorus assemblies, however, is restricted to *cautious applications* [30] due to the fact that objects not belonging to an assembly cannot be requested within an isolated region; a Chorus assembly can always be preempted to abort without a retry during a request for an object. Even though it is a data-driven execution model, Chorus has shown inferior scalability compared to Aida on irregular data parallel applications.

Solaris uses a synchronization scheme similar to delegation in its FireEngine network stack [37]. The operating system has a fixed number of locks that corresponds to the number of processors, and every network connection is assigned to one of these locks using a hash. When a network connection requires processing, e.g., a packet arrives, if the current processor cannot acquire the lock, it queues the packet for processing by the current lock holder. Aida implements a much more general programming model that is applicable to a wide range of applications (not just networking), does not require any operating system support, and the number of tasks is not limited to the number of processors in the system.

6. Conclusions and Future Work

Isolation has long been one of the most basic concerns in parallel programming. Despite the recent attention paid to software and hardware approaches to transactional memory, achieving a practical balance between programmability, correctness guarantees, and scalability in isolated execution has remained a thorny challenge for the community. In this paper, we have taken a step towards meeting this challenge in the context of a notoriously difficult problem domain: imperative parallel computations over large, shared, irregular data structures. Aida's programming model offers several high-level correctness guarantees while requiring minimal programmer expertise— indeed, the only extension needed to a fork-join parallel framework (such as OpenMP 3.0 or Habanero Java) is an isolated statement construct akin to that used in transactional memory. However, Aida's execution model is shown to be orders-of magnitude faster on irregular applications than the DTSM2 software-based transactional memory system, and to yield performance comparable to that of fine-grain locking and the Galois system with a simpler and more general programming model.

Regarding future work, one immediate goal is develop a static analysis toolkit for optimizing Aida programs. For example, some of our runtime overheads are unnecessary for *cautious applications* [30], and we will benefit from static analyses establishing that an assembly is cautious by using approaches such as [33]. On the language design end, we would like greater integration of Aida with existing Habanero Java constructs for locality and directed synchronization, such as hierarchical place trees [38] and phasers [36]. Finally, we would like to extend our formalization of Core Aida and build a process calculus that captures the essence of delegation, and to use it to articulate more precisely the guarantees offered by Aida as well as allow more sophisticated reasoning (e.g., types and invariants). Preliminary investigations on some of these topics are under way.

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 valuable discussions related to this work, and contributions to the Habanero Java infrastructure used in this research. We are grateful to the anonymous reviewers for their comments and suggestions, to Virendra Marathe for his detailed feedback on an earlier draft of this paper, and to Alan Cox for pointing us to the synchronization scheme used in the Solaris FireEngine network stack. 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.

References

[1] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 370–380, New York, NY, USA, 1991. ACM.

[2] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. Efficient, portable implementation of asynchronous multi-place programs. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 271–282, New York, NY, USA, 2009. ACM.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work-stealing. In *Proceedins of the 35th Annual IEEE Conference on Foundations of Computer Science*, 1994.

[4] Donald Burke, Joshua Epstein, Derek Cummings, Jon Parker, Kenneth Cline, Ramesh Singa, and Shubha Charkravarty. Individual-based computational modeling of smallpox epidemic control strategies. *Academic Emergency Medicine*, 13(11):1142–1149, 2006.

[5] Martin Burtscher, Milind Kulkarni, Dimitrios Prountzos, and Keshav Pingali. On the scalability of an automatically parallelized irregular application. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 109–123. Springer-Verlag, Berlin, Heidelberg, 2008.

[6] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee.

Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, September 2008.

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 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.

[8] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

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

[10] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the ninth annual symposium on Computational geometry*, SCG '93, pages 274–280, New York, NY, USA, 1993. ACM.

[11] 9^{th} DIMACS Implemetation Challenge. Available from `http://www.dis.uniroma1.it/~challenge9/`.

[12] DSTM 2.1 beta. Available from `http://www.cs.brown.edu/~mph/`.

[13] Panagiota Fatourou and Nikolaos D. Kallimanis. Blocking Universal Constructions. Technical report, University of Ioannina, 2011. TR-2011-05.

[14] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency control for high contention environments. *ACM Trans. Database Syst.*, 17:304–345, June 1992.

[15] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[16] Z. Galil and G. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[17] Galois. Available from `http://iss.ices.utexas.edu/galois/`.

[18] Habanero java web page. http://habanero.rice.edu/hj.

[19] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool, 2010.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[21] Danny Hendler and Nir Shavit. Work dealing. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 164–172, New York, NY, USA, 2002. ACM.

[22] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 253–262, New York, NY, USA, 2006. ACM.

[23] Dieter Jungnickel. *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 3rd edition, 2007.

[24] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65 –76, april 2009.

[25] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[26] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[27] The Lonestar Benchmark Suite. Available from `http://iss.ices.utexas.edu/lonestar/`.

[28] Roberto Lublinerman, Swarat Chaudhuri, and Pavol Cerny. Parallel programming with object assemblies. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 61–80, New York, NY, USA, 2009. ACM.

[29] Virendra Marathe, William Scherer, and Michael Scott. Design tradeoffs in modern software transactional memory systems. In *Workshop on languages, compilers, and run-time support for scalable systems*, 2004.

[30] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 3–14, New York, NY, USA, 2010. ACM.

[31] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the international workshop on parallel and distributed computing for symbolic and irregular applications*, PDSIA '99, pages 182–204. World Scientific, 1999.

[32] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular applications. Technical Report TR-09-05, University of Texas at Austin, 2009.

[33] Dimitrios Prountzos, Roman Manevich, Keshav Pingali, and Kathryn S. McKinley. A shape analysis for optimizing parallel graph programs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 159–172, New York, NY, USA, 2011. ACM.

[34] William Scherer and Michael Scott. Advanced contention management for dynamic software transactional memory. In

Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[35] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[36] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[37] Sunay Tripathi. FireEngine — A New Networking Architecture for the Solaris Operating System. Technical report, Sun Microsystems, 2004.

[38] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*, 2009.