Synthesizing Concurrent Schedulers for Irregular Algorithms *

Donald Nguyen
Department of Computer Science
The University of Texas at Austin
ddn@cs.utexas.edu

Keshav Pingali
Department of Computer Science
Institute for Computational Engineering and Sciences
The University of Texas at Austin
pingali@cs.utexas.edu

Abstract

Scheduling is the assignment of tasks or activities to processors for execution, and it is an important concern in parallel programming. Most prior work on scheduling has focused either on static scheduling of applications in which the dependence graph is known at compile-time or on dynamic scheduling of independent loop iterations such as in OpenMP.

In irregular algorithms, dependences between activities are complex functions of runtime values so these algorithms are not amenable to compile-time analysis nor do they consist of independent activities. Moreover, the amount of work can vary dramatically with the scheduling policy. To handle these complexities, implementations of irregular algorithms employ carefully handcrafted, algorithm-specific schedulers but these schedulers are themselves parallel programs, complicating the parallel programming problem further.

In this paper, we present a flexible and efficient approach for specifying and synthesizing scheduling policies for irregular algorithms. We develop a simple compositional specification language and show how it can concisely encode scheduling policies in the literature. Then, we show how to synthesize efficient parallel schedulers from these specifications. We evaluate our approach for five irregular algorithms on three multicore architectures and show that (1) the performance of some algorithms can improve by orders of magnitude with the right scheduling policy, and (2) for the same policy, the overheads of our synthesized schedulers are comparable to those of fixed-function schedulers.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Concurrent programming structures

General Terms Algorithms, Performance

Keywords Amorphous data-parallelism, Irregular programs, Multicore processors, Optimistic parallelization, Program synthesis, Scheduling

* This work was supported by NSF grants 0833162, 0719966, 0702353, 0724966, and 0739601, as well as grants from the IBM and Intel Corporations.

1. Introduction

Given a decomposition of a program into activities or tasks, the *scheduling problem* is the *assignment of these activities to processors and the specification of an order in which each processor should execute the activities assigned to it*. Scheduling is important for both sequential and parallel implementations of algorithms since it may affect locality and load balance; it may even effect the total amount of work performed by some programs, as we show in this paper.

Scheduling can be done either statically by a compiler or dynamically by a runtime or operating system. Static scheduling can be used when dependences between activities are known statically and the execution time of each activity can be estimated accurately at compile-time. Stencil computations are the classic examples of algorithms amenable to static scheduling. Dynamic scheduling is useful for problems in which (1) dependences between activities cannot be elucidated statically, or (2) new work is created dynamically so the number of activities is not known statically, or (3) accurate estimates of the time required to execute each activity are not available. The vast majority of algorithms, including almost all *irregular* algorithms (in which the key data structures are sparse graphs) require dynamic scheduling.

For the most part, prior work on dynamic scheduling has focused on problems in which there are no dependences between activities and new work is not created dynamically, so the only problem is that time required to execute an activity cannot be determined accurately. Self-scheduling of DO-ALL loops in OpenMP is the classic example [10]. Activities in this case correspond to iterations of the DO-ALL loop; the number of iterations is known before the loop is executed, and it is assumed that there are no dependences between iterations. However, different iterations may take different and unpredictable amounts of time to execute, so to ensure good load balance, OpenMP provides scheduling policies such as chunked dynamic self-scheduling, in which a processor gets a chunk of k iterations every time it needs work, and guided self-scheduling, in which the chunk size decreases steadily as the loop nears completion. Chunking reduces the overheads of scheduling and may improve locality.

More recently, attention has shifted to "task-parallelism" in which new activities are created dynamically, although it is still assumed that all activities are independent except for fork-join control dependences. In OpenMP 3.0, there is support for different dynamic scheduling policies such as breadth-first and work-first policies [11]. Another popular technique is *work-stealing*. In work-stealing, each thread maintains a local deque that it initially adds and removes activities from. When a thread's local deque is empty, it selects the local deque of another thread, the victim, and

tries to steal activities from it. Work-stealing is parameterized by the order maintained in the local deque (usually LIFO) and how a thread selects a victim (usually at random). Work-stealing was implemented in MultiLisp [13] and was later popularized by the Cilk language [4], where it is used to implement fork-join parallelism. It is now available in many programming environments [17, 21, 22].

The key assumptions in this work on task-parallelism are that any dependences between activities are captured by fork-join control dependences and are known statically. While these assumptions are reasonable for regular (dense-array) algorithms and divide-and-conquer algorithms, they do not hold for most irregular graph algorithms because dependences in these algorithms are complex functions of runtime values (*e.g.*, the shape of the graph and the values on nodes and edges), which may themselves change during execution. An abstract description of parallelism in irregular algorithms is the following. At each step of the algorithm, there are certain *active nodes* in the graph where computation needs to be performed. Performing the computation at an active node may require reading or writing other graph nodes and edges, known collectively as the *neighborhood* of that activity (the neighborhood is usually distinct from the neighbors of the active node). In general, there are many active nodes in a graph, so a sequential implementation must pick one of them and perform the appropriate computation. In the *unordered algorithms* considered in this paper, the implementation is allowed to pick *any* active node for execution.[1] The final output may be different for different orders of executing active nodes, but all such outputs are acceptable, a feature known as *don't-care non-determinism*. A parallel implementation of such an algorithm can process active nodes simultaneously, provided their neighborhoods do not overlap (this condition can be relaxed but is sufficient for correct execution). However, the neighborhood of an activity is generally not known until the activity has finished execution. This parallelism is known as *amorphous data-parallelism* [30].

Efficient exploitation of amorphous data-parallelism requires far more sophisticated runtime support than fork-join parallelism or DO-ALL parallelism for the following reasons.

1. In most irregular algorithms, nodes become active dynamically, so the number of activities is not known statically.
2. In general, the neighborhood of an activity may be known only after that activity completes execution. Therefore, it may be necessary to use optimistic or speculative parallelization.
3. Most importantly, the number of activities that are executed by an algorithm may be dramatically different for different schedules.

For these reasons, even *sequential* implementations of irregular algorithms often use handcrafted, algorithm-specific scheduling policies; for example, some mesh refinement algorithms process triangles or tetrahedra in decreasing size order since this can reduce the total amount of refinement work [27]. Section 2 describes some of the more important policies in the literature. However, following these orders strictly can dramatically reduce parallelism, so parallel implementations of irregular algorithms often use more complex scheduling policies that trade off extra work for increased parallelism. These schedulers are themselves concurrent data structures and add to the complexity of parallel programming. In addition, they cannot easily be reused for other applications.

In this paper, we present a flexible and efficient approach for specifying and synthesizing schedulers for sequential and parallel implementations of irregular algorithms. We distinguish between *scheduling policies*, which are descriptions (possibly informal) of the order in which activities should be processed (*e.g.*, LIFO, FIFO, etc.), *scheduling specifications*, which are formal descriptions of scheduling policies, and *schedulers*, which are concrete implementations of scheduling specifications. A *schedule* is a specific mapping of activities to processors in time.

The rest of this paper is organized as follows. In Section 2, we describe several irregular algorithms and scheduling policies studied in the literature. Section 3 describes a language that concisely encodes these policies as scheduling specifications. Section 4 describes how concurrent schedulers are synthesized from these specifications. Our modular and compositional approach can produce efficient schedulers without requiring users to write complex, concurrent code. In Section 5, we evaluate our synthesized schedulers against *fixed-function schedulers* that use work-stealing with local LIFOs, work-stealing with local FIFOs and bulk-synchronous worklists.[2] We show that (1) for the same scheduling policy, fixed-function schedulers and synthesized schedulers have similar performance, and (2) for some algorithms, algorithm-specific schedulers synthesized by our approach outperform fixed-function schedulers. Section 6 compares our approach with previous work in scheduling and synthesis. Finally, Section 7 summarizes our contributions.

2. Algorithms

In this section, we describe five algorithms that exhibit amorphous data-parallelism. All the algorithms are unordered, which means that activities can execute in any order, and each algorithm has at least one algorithm-specific scheduling policy that has been proposed in the literature. Pseudocode for these algorithms is written using the Galois programming model [19], which is a sequential, object-oriented programming model augmented with a *Galois unordered-set iterator*, which is similar to set iterators in C++ or Java but permits new items to be added to a set while it is being iterated over.

- **foreach** (e : Set S) {B(e)} — The loop body B(e) is executed for each item e of set S. The order in which iterations execute is indeterminate and can be chosen by the implementation. There may be dependences between the iterations. An iteration may add items to S during execution.

2.1 Delaunay Triangulation

Finding the Delaunay triangulation (DT) of a set of points is a classic computational geometry problem. There are many algorithms for finding the triangulation; here, we describe the incremental algorithm of Bowyer and Watson [5, 35]. Each point is associated with the triangle that contains it. We assume points are in 2D. Initially, there is one large triangle that covers all the points. Each point p calculates all triangles whose circumcircles include p. This is the cavity of p. The cavity is re-triangulated, p is removed and the remaining points of the cavity are redistributed among their corresponding triangles. Figure 1 shows the pseudocode. G is the graph representing the triangulation. Figure 2 shows an example of processing one point.

All orders of processing points lead to the same Delaunay triangulation. Clarkson and Shor have shown that selecting points at random is optimal [8]. Amenta *et al.* present an algorithm called biased randomized insertion order (BRIO) that takes advantage of locality while still maintaining the optimality of randomness [1]. Briefly, let n be the number of points to triangulate. Points are processed in $\log n$ rounds. The probability that a point is processed in the final round is $\frac{1}{2}$. For the remaining points, the probability that

[1] In contrast, *ordered algorithms* have a specific order in which active nodes must be executed.

[2] Fixed-function schedulers and synthesized schedulers are not mutually exclusive. In particular, work-stealing with local LIFOs can be seen as one implementation of a LIFO policy. However, in this paper, we separate the two to clearly delineate our contribution.

```
1  Workset ws = new Workset(points);
2  foreach (Point p : ws) {
3    Cavity c = new Cavity(p);
4    c.expand();
5    c.retriangulate();
6    G.update(c);
7    c.dispatchPoints();
8  }
```

Figure 1. Pseudocode for Delaunay triangulation.

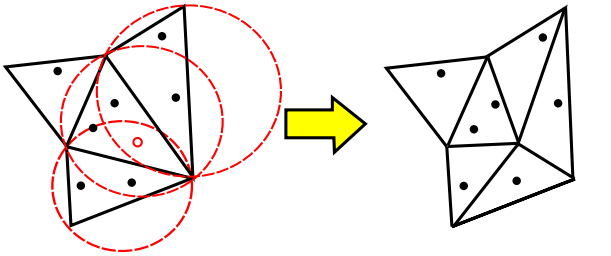

Figure 2. Example of processing an active point (hollow and red) for Delaunay triangulation. Circles are circumcircles of triangles containing the active point.

they will be processed in the next-to-last round is $\frac{1}{2}$, and so on until the first round. For the first round, all remaining points are processed with probability one. Within a round, points are processed according to the spatial divisions of an oct-tree.

2.2 Delaunay Mesh Refinement

Delaunay mesh refinement (DMR) [7] is an algorithm related to Delaunay triangulation. Given a Delaunay triangulation, triangles may have to satisfy additional quality constraints beyond that guaranteed by triangulation. To improve the quality of a triangulation, Delaunay mesh refinement iteratively fixes "bad" triangles, which do not satisfy the quality constraints, by adding new points to the mesh and re-triangulating. Refining a bad triangle may itself introduce new bad triangles, but it can be shown that, at least in 2D, this iterative refinement process will terminate and produce a guaranteed-quality mesh.

Figure 3 shows the pseudocode for this algorithm. It is similar to Delaunay triangulation, except that activities are centered on triangles rather than points. In both cases, a cavity is expanded and re-triangulated. However, in DMR, new bad triangles can be created that must be processed as well. They are tracked in a workset.[3] Additionally, different orders of processing bad triangles lead to different meshes, but all such meshes satisfy the quality constraints and are acceptable outcomes of the refinement process [7]. In contrast, for Delaunay triangulation, different orders still produce the same triangulation.

Naive implementations of DMR have quadratic worst-case running times [31] although they perform well in practice. Miller proved sub-quadratic worst-case time of a modification of DMR that processes triangles in decreasing circumcircle diameter together with other changes [27]. In Shewchuk's Triangle program, bad triangles are placed into buckets according to their minimum angle, each bucket stores triangles in FIFO order, and buckets are processed in increasing angle order [32]. Kulkarni *et al.* showed that a parallel implementation of DMR that distributes the initial

[3] Throughout this paper, we use the colloquial term workset, although more formally, these objects behave as bags or multisets because they may contain duplicate items.

```
1  Workset ws = new Workset(G.badTriangles());
2  foreach (Triangle t : ws) {
3    if (!G.contains(t)) continue;
4    Cavity c = new Cavity(t);
5    c.expand();
6    c.retriangulate();
7    G.update(c);
8    ws.addAll(c.badTriangles());
9  }
```

Figure 3. Pseudocode for Delaunay mesh refinement.

bad triangles among threads and uses thread-local stacks for newly created bad triangles performs well in practice [18].

2.3 Inclusion-based Points-to Analysis

Inclusion-based points-to analysis (PTA), also known as Andersen's algorithm [2], is a flow and context-insensitive static analysis that determines the points-to relation for program variables. PTA is a fixpoint algorithm that computes the least solution to a system of set constraints. The basic algorithm maintains a workset of program variables whose points-to relations need to be computed. For each variable in the workset, the algorithm examines the system of constraints to see if the current variable satisfies the constraints. If so, the algorithm continues processing the remaining variables. If not, some set of program variables are modified to satisfy the constraints. These modified variables are then added to the workset, and the algorithm continues until the workset is empty. Hardekopf and Lin showed how the basic fixpoint algorithm augmented with sophisticated cycle-detection can scale to large problem sizes [14]. From this algorithm, Mendez-Lojo *et al.* produced the first parallel implementation of this algorithm [24]. Our results in Section 5 are based on this implementation.

Since this is a fixpoint algorithm, all orders of processing variables will produce the same solution. Many heuristics have been proposed for organizing the workset, such as processing variables in least recently fired (LRF) order [29] or dividing the workset into current and next parts [28]. Variables are processed from the current part, but newly active variables are enqueued onto the next part. When the current part is empty, the roles of the current and next parts are swapped. Hardekopf and Lin report that the divided workset approach performs better in practice [14].

2.4 Single-source Shortest-path

Given a weighted, directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights, and a source node $s \in V$, the single-source shortest-path (SSSP) problem is to find the least weight path from s to all other nodes in the graph. The Bellman-Ford algorithm solves this problem on graphs with arbitrary weights. In the case of graphs with no negative weight edges, there are specialized algorithms such as Dijkstra's algorithm. In this paper, we consider the case where graphs have no negative weight edges. The Bellman-Ford algorithm can also be optimized for this case by reformulating it as a fixpoint algorithm. Instead of processing each node $|V|$ times, there is a workset that maintains nodes whose distances have changed; only those nodes are processed. One parallel SSSP algorithm is delta-stepping [25].

Each one of these algorithms can be viewed as an instantiation of a generic algorithm with different scheduling policies (see Figure 4). Each node n has a tentative distance estimate $n.dist$ (initialized to ∞), and there is a set ws of distance requests $(u, d, light)$ indicating that node u can be updated with distance d. The additional $light$ value is only used by the delta-stepping algorithm. If the requested distance d is less than the current distance $u.dist$ at node u, the current distance is updated. This generates a set of new

```
1  Workset ws = new Workset({(u, w(s,n), w(s,n) ≤ Δ) : (s,u) ∈ E})
2  s.dist = 0
3  foreach ((u, d, light) : ws) {
4    if (d < u.dist) {
5      u.dist = d;
6      for (Node v : u.neighbors()) {
7        float w = G.edgeData(u, v);
8        ws.add((v, d + w, w <= Δ))
9      }
10   }
11 }
```

Figure 4. Pseudocode for generic single-source shortest path algorithm. Δ parameter and light value are only used by delta-stepping algorithm.

requests based on the new value of $u.dist$ that are then added to ws.

The (optimized) Bellman-Ford algorithm processes requests in arbitrary order. Dijkstra's algorithm is an *ordered* algorithm which processes requests in increasing distance order. The delta-stepping algorithm orders requests as follows. The algorithm creates a sequence of buckets, $b_1, b_2, \ldots, b_N$. Bucket b_i contains all requests with distances in the range $[\Delta(i-1), \Delta i)$ where Δ is an input to the algorithm. Furthermore, requests are divided into light requests, whose distances come from edges with weight less than Δ, and heavy requests otherwise. Requests are processed in increasing bucket order. Within a bucket, light requests are processed before heavy ones.

2.5 Preflow-push

Given a directed graph $G = (V, E)$, a capacity function $c : E \to \mathbb{R}^+$ mapping edges to non-negative values and source and sink nodes $s, t \in V$, the preflow-push algorithm computes the maximal flow from source to sink. Unlike in maxflow algorithms based on augmenting paths, nodes in preflow-push can temporarily have more flow coming into them than going out. Each node n maintains its excess inflow $n.excess$. Each node n also has a label called height, which is an estimate of the distance from n to t in the residual graph induced by unsaturated edges. Nodes with non-zero excess that are not the source or sink are contained in a workset. These nodes are called active nodes. The preflow-push algorithm repeatedly selects a node from the workset. Each node tries to eliminate its excess by pushing flow to a neighbor (see Figure 5). Pushing flow may cause a neighbor to become active. A node can only push flow to a neighbor at a lower height. If a node is active but no neighbors are eligible to receive flow, the node relabels itself, increasing its height to one more than its lowest height neighbor.

Cherkassy and Goldberg show the importance of two heuristics named global relabeling and gap relabeling [6]. Global relabeling is a technique that periodically reassigns heights by performing a breadth-first traversal from the sink. The frequency of global relabeling is determined empirically. Gap relabeling is a technique that preemptively removes from the workset any nodes that cannot push flow to the sink. The key insight is that if no node has height h, all nodes with height greater than h cannot push flow to the sink. Cherkassy and Goldberg also consider two orders for processing active nodes: HL order, where nodes are processed in decreasing height order, and FIFO order. The parallel implementation evaluated in Section 5 includes all these heuristics.

3. Scheduling Policies

Section 2 demonstrates that many algorithms benefit from scheduling strategies that are more complex than simple strategies like LIFO and FIFO. In this section, we show how the informal policies

```
1  Workset ws = new Workset({s});
2  foreach (Node u : ws) {
3    L1:
4    while (u.excess > 0) {
5      for (Node v : u.neighbors()) {
6        float cap = G.edgeData(u, v);
7        if (cap > 0 && u.height == v.height + 1) {
8          pushFlow(u, v, min(cap, u.excess));
9          if (v != s && v != t)
10           ws.add(v);
11         if (u.excess == 0)
12           break L1;
13       }
14     }
15     relabel(u);
16   }
17   if (*)
18     globalRelabel()
19 }
```

Figure 5. Pseudocode for preflow-push.

described in the previous section can be encoded in a simple but flexible specification language.

Scheduling specifications are built from *ordering rules*, where an ordering rule specifies a total order on activities or items. Given two items a and b, an ordering rule R may specify that a should be processed before b (written as $a <_R b$) or vice versa ($b >_R a$), or it may leave the order unspecified ($a =_R b$). For example, if items have integer priorities, an ordering rule A_1 may order them in ascending priority order; the relative order of items with the same priority is left unspecified by A_1. Ordering rules can be composed sequentially in a manner similar to lexicographic ordering: a sequence $D = R_1 R_2 R_3 \ldots$ is itself an ordering rule that first orders items according to R_1; if two items are not strictly ordered by R_1, they are ordered according to R_2, etc. For example, if F_1 orders items in FIFO order, then $A_1 F_1$ denotes the order in which items are processed in increasing priority order and items with the same priority are processed in FIFO order. For the synthesis procedure described in Section 4, it is convenient to distinguish between *final rules*, which are rules that appear last in an ordering sequence, and *non-final rules*.

As discussed in Section 2, some implementations of irregular algorithms maintain separate global and thread-local worksets, so it is natural to use different ordering rules for them. A scheduling specification can have both a global ordering rule and a local ordering rule. The global rule is applied to the initial set of items, and the local rule is applied to each thread-local workset, which holds items created dynamically by the corresponding thread. A thread accesses the global workset only when its local workset is empty. Continuing the previous example, if L_1 is last-in-first-out (LIFO) order, then global order $A_1 F_1$ and local order L_1 specify that global items are processed as before, and local items are processed in LIFO order.

Figure 6 gives the syntax rules for scheduler specifications. $\mathbf{T}$ is the type of items. Figure 7 gives the semantics. The meaning of an ordering rule R is given as a function over items a and b that is true iff $a <_R b$ ($<$ is the standard order on integers and reals). For the FIFO and LIFO rules, we define an auxiliary function $\mathbf{time}$ that maps an item to an integer according to when the item is added to the scheduler. For the first item x_1 added to the scheduler, $\mathbf{time}(x_1) = 0$; for the second item x_2, $\mathbf{time}(x_2) = 1$, and so on. f_{U} is an injective function mapping items to integers; this function essentially encodes a random permutation of items. The function f_{D} should be consistent with a total order.

Figure 8 shows the specifications of the scheduling policies discussed in Section 2. When a specification has an empty local ordering, we omit the global and local tags. The BRIO specification

P	::=	Global: D Local: D	*Specification*
D	::=	$R_{\text{NF}}{}^{*}$ $R_{\text{F}}?$	*Ordering rule*
R_{F}	::=	FIFO	*Final rule*
	|	LIFO	
	|	Random	
R_{NF}	::=	ChunkedFIFO(k)	*Non-final rule*
	|	ChunkedLIFO(k)	
	|	Ordered(f_{D})	
	|	OrderedByMetric(f_{M})	
k			*Integer*
f_{D}			$\mathbf{T} \times \mathbf{T} \to \mathbf{bool}$
f_{M}			$\mathbf{T} \to \mathbb{R}$

Figure 6. Scheduler specification syntax.

$$
\begin{array}{rcl}
[\![\text{FIFO}]\!](a,b) & = & \textbf{time}(a) < \textbf{time}(b) \\
[\![\text{LIFO}]\!](a,b) & = & \textbf{time}(a) > \textbf{time}(b) \\
[\![\text{Random}]\!](a,b) & = & f_{\text{U}}(a) < f_{\text{U}}(b) \\
[\![\text{ChunkedFIFO}(k)]\!](a,b) & = & \lfloor \textbf{time}(a)/k \rfloor < \lfloor \textbf{time}(b)/k \rfloor \\
[\![\text{ChunkedLIFO}(k)]\!](a,b) & = & \lfloor \textbf{time}(a)/k \rfloor > \lfloor \textbf{time}(b)/k \rfloor \\
[\![\text{Ordered}(f_{\text{D}})]\!](a,b) & = & f_{\text{D}}(a,b) \\
[\![\text{OrderedByMetric}(f_{\text{M}})]\!](a,b) & = & f_{\text{M}}(a) < f_{\text{M}}(b) \\
[\![R_1\ R_2\ \ldots]\!](a,b) & = & \begin{cases} [\![R_2\ \ldots]\!](a,b) & \text{if } a =_{R_1} b \\ [\![R_1]\!](a,b) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 7. Rule semantics.

assumes that items have already been assigned rounds according to the random distribution described previously. To process light requests before heavy ones, the delta-stepping specification splits each bucket into two: one part for the light requests and one part for the heavy requests.

4. Synthesis

It is straightforward to implement sequential schedulers for policies specified in the language described in Section 3. Each rule can be implemented by a *workset*, which is an object with the following methods:

- **void** add(**T** t) — adds an item to the workset
- **T** poll() — removes and returns the next item to execute; if there are no items left, returns a **null** value distinct from all items added to the workset

Items are added to the workset by invoking the add method, which returns the value **void** when it completes. To get items from the workset, the poll method is invoked; if this method invocation does not find any items in the workset, it returns a unique **null** value.

The goal of this section is to synthesize *concurrent* worksets that implement this functionality. One approach is to compose a set of library components, each of which is a workset by itself. There is a workset for each final rule; non-final rules are implemented by worksets parameterized by a function that constructs instances of the next workset in the ordering sequence, the *inner* workset.

However, a naive implementation along these lines can be incorrect in a concurrent setting, as we argue in Section 4.1, and the result may not satisfy any intuitive notion of correctness such as linearizability [16]. To address this problem, we propose a relaxed correctness condition in Section 4.2 that requires modifications to the semantics of worksets and to how they are used by clients.

In Section 4.3, we discuss two important consequences of this relaxed condition: (1) all final scheduling policy rules in Section 3 have implementations that satisfy the relaxed condition, and (2) non-final worksets satisfy the relaxed condition assuming only that their inner worksets satisfy the relaxed condition. This permits compositional construction of worksets.

In Section 4.4, we discuss the implementation of scheduling policies in Java and several optimizations to improve the performance of the synthesized worksets.

4.1 Problems with Naive Composition

To understand the issues that arise in composing worksets, consider the implementation of the OrderedByMetric rule in lines 1-23 of Figure 9, and its client, a simple *runtime system*, in lines 24-30. The runtime system manages threads and assigns them work. The workset creates an array of inner worksets and processes each inner workset in ascending order. This workset is essentially an implementation of a priority queue in which the range of keys is known *a priori*; there is one inner workset (bucket) for each key value. Similar worksets have been used in a variety of sequential [1, 6, 32] and parallel implementations [25] of irregular algorithms.

Unfortunately, the workset in Figure 9 can exhibit incorrect behavior because it is possible for items to be inserted into the workset but never retrieved. Consider two threads T_1 and T_2, where T_1 is executing the poll method and T_2 is executing the add method. The following sequence of events may take place:

1. T_1 executes line 14. The cursor value is i, and the the poll method on buckets[i] returns **null**.
2. T_2 executes lines 6-8. The value of index $= i$, so an item is added to buckets[i].
3. T_1 executes lines 15 and 16 of the poll method, incrementing cursor.

Clearly, the item added by T_2 is now lost. The race exists even if each line of the implementation is atomic, so that reading and updating the cursor on line 8 or incrementing its value on line 16 is performed atomically. To use this implementation correctly in a concurrent context, we must ensure that when poll moves to the next bucket, no thread is adding an element to a higher priority bucket. One solution is to use transactional memory [15], but the overhead of software transactional memory systems is high, especially for a performance-critical component like a scheduler. Hardware transactional memory systems exist, but they are not widely available.

We summarize these observations.

- In general, simple compositions of worksets do not produce correct concurrent worksets.
- Composition of concurrent worksets is problematic even in absence of having to maintain a particular order of items.

4.2 Relaxed Concurrent Semantics

In this section, we describe a solution to the problem of composing worksets that takes advantage of the fact that scheduling specifications for unordered algorithms are inherently "fuzzy" and are intended as suggestions to the runtime system rather than as commands that must be followed exactly. At a high level, the idea is the following.

- We relax the behavior of the poll method so that in a parallel setting, it may return a different item than the one it would have returned in a sequential setting. In addition, we allow poll to return **null** even when there are still items in the workset. These modifications permit us to implement poll with low overhead.
- To compensate for the relaxed behavior of poll, we introduce a new method poll-s that is similar to poll but is never executed concurrently with other invocations. It returns **null** only when the workset is truly empty.

Algorithm	Order	Specification	Used by
DMR	Bucketed triangle angle (AS2)	OrderedByMetric($\lambda t.$ minangle(t)) FIFO	[32]
	Local stack (AS1)	Global: ChunkedFIFO(k) Local: LIFO	[18]
DT	BRIO (AS1)	OrderedByMetric($\lambda p.\ p.round$) ChunkedFIFO(k)	[1]
	Random	Random	[8]
PFP	FIFO	FIFO	[12]
	HL order (AS1)	OrderedByMetric($\lambda n.\ -n.height$) FIFO	[6]
PTA	LRF	FIFO	[29]
	Split worklists (BS-F)		[14, 28]
SSSP	Bellman-Ford	FIFO	[9]
	Delta-stepping (AS1)	OrderedByMetric($\lambda n.\ \lfloor 2 * n.w/\Delta \rfloor + (n.light)\ ?\ 0 : 1$) FIFO	[25]
	Dijkstra (AS2)	Ordered($\lambda a, b.\ a.w \leq b.w$)	[9]

Figure 8. Application-specific scheduling specifications.

```
1  class OrderedByMetricWorkset implements Workset {
2    Workset[] buckets;
3    int cursor;
4
5    void add(T t) {
6      int index = floatToInt(fM(t));
7      buckets[index].add(t);
8      if (index < cursor) cursor = index;
9    }
10
11   T poll() {
12     T retval = null;
13     while (cursor < buckets.length) {
14       retval = buckets[cursor].poll();
15       if (retval == null)
16         cursor++;
17       else
18         break;
19     }
20     return retval;
21   }
22 }
23
24 OrderedByMetricWorkset scheduler;
25 ThreadPool.fork(N); // spawn N threads
26 T item;
27 while ((item = scheduler.poll()) != null) {
28   operator.call(item, scheduler);
29 }
30 ThreadPool.join(N);
```

Figure 9. Naive bucketed scheduler and its use in a parallel runtime system.

Intuitively, if most items are retrieved from the workset using the poll method, and poll-s is used infrequently to determine if the workset is truly empty, we get a solution that is both correct and efficient. In this section, we formalize this behavior, and in Section 4.3, we show how this behavior is closed under composition.

We model a workset by its history H, which is a finite sequence of *events*, where an event is either (1) a method invocation, (2) a response to a method invocation, or (3) a special termination event, $\langle \text{term} \rangle$. We write $\langle o.m(a, b, \dots)\ T \rangle$ for an invocation on object o of method m with arguments $a, b, \dots$ by thread T, $\langle o.m(a, b, \dots)/r\ T \rangle$ for a response to method m with return value r, and **void** for the unit return value. An invocation $\langle o_1.m_1(a_1, b_1, \dots)\ T_1 \rangle$ *matches* a response $\langle o_2.m_2(a_2, b_2, \dots)/r\ T_2 \rangle$ if $o_1 = o_2, m_1 = m_2, a_1 = a_2, b_1 = b_2$ and so on, and $T_1 = T_2$. We omit the object and/or thread if it is clear from context. We use $x_1, x_2, \dots$ as variables over arguments or return values.

An invocation is *pending* in H if it has no matching response. A history is *whole* if it has no pending invocations and it contains exactly one termination event and that is the last event in the history. A history *restricted to* object o or thread T is the subsequence with only events on o or by T respectively and possibly a termination event. Without loss of generality, we assume items are unique. The notation $a \rightarrow_H b$ denotes that event a precedes event b in history H; we write $a \rightarrow b$ when the history is clear from context.

Property 1 is a formal description of the behavior of poll and poll-s. Condition B1 states that (1) items returned by poll and poll-s must have been added earlier by the add method, and (2) a given item can only be returned once. Both requirements are captured by the injective function M. Condition B2 states that poll-s cannot be invoked when there are pending method invocations. Condition B3 states that if poll-s returns **null**, all previously added items have been retrieved by poll or poll-s, and the workset is truly empty. This is captured by requiring M to be a bijective function.

Property 1 (Weak Bag)**.** *A history H models a* weak bag *if the following are true:*

*B1. There is an injective function M from non-***null** *response events $e_1 = \langle poll()/x_1\ T_1 \rangle$ or $e_1 = \langle poll\text{-}s()/x_1\ T_1 \rangle$ to invocation events $e_2 = \langle add(x_2)\ T_2 \rangle$ such that (1) $M(e_1) = e_2$, (2) $x_1 = x_2$, and (3) $e_2 \rightarrow e_1$.*

B2. For each invocation event $e = \langle poll\text{-}s()\ T \rangle$ of poll-s in $H = H_1, e, H_2$, there are no pending invocations in H_1.

B3. For each **null** *response event $e = \langle poll\text{-}s()/\mathbf{null}\ T \rangle$ of poll-s in $H = H_1, e, H_2$, H_1 satisfies condition B1 and M is a bijective function.*

A workset is *correct* if it only generates whole histories satisfying Property 1. It is the responsibility of the client to use the workset properly by never invoking poll-s concurrently with other methods. This form of correctness may seem particularly weak since it does not refer to the sequential ordering semantics. However, we have found it useful because it includes many natural compositions of worksets as well as most hand-written schedulers. A linearizable bag is a correct workset if one considers poll-s the same as poll. Likewise, a bag with a single lock guarding all its methods is also a correct workset.[4]

One correct workset and its proper use by a runtime system is the following modification of Figure 9. After line 30, the runtime system should call poll-s; if the returned value is a non-**null** item, the system should process that item and go to line 25 to continue execution. The poll-s method should walk the bucket array calling poll-s on each inner workset and return the first non-**null** item if it exists.

The implementations in the Galois system of the final rules in Figure 6 satisfy Property 1 since the LIFO and FIFO rules are implemented by a linearizable stack and queue respectively, and the

[4] It is possible to introduce deadlock when arbitrarily composing worksets with locks. However, compositions based on Theorem 2 whose implementations are themselves wait-free do not introduce deadlocks.

Random rule is implemented with a resizable array and all method invocations are protected by a single lock.

4.3 Workset Composition

We now show how the implementations of the non-final rules in Figure 6 satisfy Property 1, assuming they are parameterized by correct worksets. We give a detailed description for the OrderedByMetric workset, and for lack of space, summarize the reasoning for the other non-final worksets.

Theorem 1 models the behavior of the OrderedByMetric workset by its history with respect to its inner worksets. Theorem 2 shows how objects that generate such histories are correct worksets.

Theorem 1 (OrderedByMetric)**.** *Let o be an instance of the OrderedByMetric workset in Figure 9 modified so that each thread maintains a thread-local cursor variable and poll-s walks all the buckets calling poll-s on each inner workset. Let $W = \{w_1, w_2, \ldots, w_n\}$ be the set of correct worksets contained in the buckets. The workset o only generates whole histories containing the following non-overlapping sequences when restricted to thread T for all threads:*

D1. $\langle o.add(x)\rangle$,
$\langle w.add(x)\rangle$, $\langle w.add(x)/\mathbf{void}\rangle$,
$\langle o.add()/\mathbf{void}\rangle$.

D2. $\langle o.poll()\rangle$,
$(\langle w_*.poll()\rangle, \langle w_*.poll()/\mathbf{null}\rangle)^*$,
$\langle w.poll()\rangle$, $\langle w.poll()/x\rangle$,
$\langle o.poll()/x\rangle$ *where* $x \neq \mathbf{null}$.

D3. The above with poll() *replaced with poll-s*().

D4. $\langle o.poll()\rangle$,
$(\langle w_*.poll()\rangle, \langle w_*.poll()/\mathbf{null}\rangle)^*$,
$\langle o.poll()/\mathbf{null}\rangle$.

D5. $\langle o.poll\text{-}s()\rangle$,
$\langle w_1.poll\text{-}s()\rangle, \langle w_1.poll\text{-}s()/\mathbf{null}\rangle, \ldots,$
$\langle w_n.poll\text{-}s()\rangle, \langle w_n.poll\text{-}s()/\mathbf{null}\rangle$,
$\langle o.poll\text{-}s()/\mathbf{null}\rangle$.

Proof. Note that the only objects shared between threads executing methods of o are the inner worksets in W, represented in Figure 9 as the variable buckets. Thus, it is sufficient to only consider sequential executions of o.

The add method clearly satisfies clause D1.

The poll method may either return **null** or a non-**null** value. In the case of **null**, each inner workset visited returns **null**, and o satisfies clause D4. In the case of a non-**null** value, there are some number of inner worksets that return **null** and exactly one that returns a non-**null** value. This satisfies clause D2.

The poll-s method is the same as the poll method except that it visits all the inner worksets in W. By reasoning similar to the poll method, the poll-s method must satisfy clauses D3 or D5. □

Theorem 2 (OrderedByMetric is correct)**.** *Whole histories H satisfying Theorem 1 model a weak bag.*

Proof. We consider each condition of Property 1 in turn.

First, H satisfies condition B1. Without loss of generality, we discuss events of poll; events of poll-s behave similarly. From clauses D2 and D4, the only time the workset o produces a non-**null** response $e_1 = \langle o.\text{poll}()/x_1\rangle$ is when one of its correct worksets $w \in W$ produces a response $f_1 = \langle w.\text{poll}()/x_1\rangle$. There is exactly one event e_1 for each event f_1 and vice versa. Let M_1 be the bijective function from events of the form e_1 to events of the form f_1. From clause D1, there is exactly one event $f_2 = \langle w.\text{add}(x_2)\rangle$ for each event $e_2 = \langle o.\text{add}(x_2)\rangle$ and vice versa.

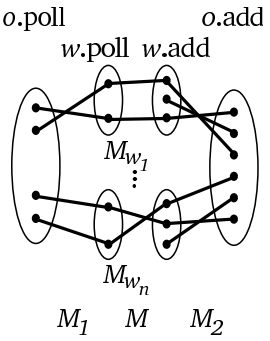

Figure 10. Relationship between M_1, M and M_2 in the proof of Theorem 2

Let M_2 be the bijective function from events of the form f_2 to events of the form e_2.

We now show there exists an injective function M_C such that $M_\text{C}(e_1) = e_2$, $x_1 = x_2$ and $e_1 \to e_2$. By definition, $M_1(e_1) = f_1 = \langle w.\text{poll}()/x_1\rangle$. Since w is a correct workset, there exists an injective function M_w such that $M_w(f_1) = f_2 = \langle w.\text{add}(x_1)\rangle$ and $f_2 \to f_1$. Let M be the expansion of M_w over the range of all $w \in W$. The ranges of $M_{w_1}, \ldots, M_{w_n}$ are disjoint and each M_w is an injective function so M is also an injective function (see Figure 10). By definition, $M_2(f_2) = e_2 = \langle o.\text{add}(x_1)\rangle$. Let M_C be the composed function $M_2 \circ M \circ M_1$. M_C is injective because M is injective and M_1 and M_2 are bijective. $M_\text{C}(e_1) = e_2$ by function composition. From clauses D1 and D2, $e_2 \to f_2$ and $f_1 \to e_1$, and by the correctness of w, $f_2 \to f_1$; so, by transitivity, $e_2 \to e_1$.

Second, H satisfies condition B2. Clients of o do not invoke o.poll-s() concurrently. From clauses D3 and D5, it is clear that if there are no pending invocations immediately before o.poll-s() is invoked, then there will be no pending invocations immediately before w.poll-s() for all $w \in W$.

Finally, H satisfies condition B3. From above, we know that M_C satisfies condition B1 and is an injective function from non-**null** poll and poll-s responses to add invocations on o. We now show that when o produces the **null** response $\langle o.\text{poll-s}()/\mathbf{null}\rangle$, M_C is a bijection as well. From clause D5, if o produces a **null** response, all $w \in W$ have produced a **null** response as well. Thus, M is a bijection, and correspondingly, M_C is one as well. □

The ChunkedFIFO rule is implemented with a single linearizable (global) queue whose elements (chunks) are instances of the inner workset. Each inner workset contains at most k items. Each thread maintains a thread-local chunk to poll from. When the chunk is empty, the thread polls from the global queue for the next chunk. Each thread also maintains a thread-local chunk to add to. When the chunk is full, the thread adds it to the global queue and creates a new empty chunk to add to. The poll-s method walks each thread-local chunk and the global queue calling poll-s on each.

Theorem 2 does not immediately apply because chunks are created and discarded dynamically. One modification would be to keep track of all the chunks ever created. However, one observation is that chunks are accessed by at most one thread at a time, and they are discarded when they are empty, which can be determined by invoking poll-s on the chunk. Empty chunks will never contain any more items. Thus, discarded chunks do not affect the eventual correctness of the workset. Only chunks that may contain items matter, which are precisely those traversed by poll-s.

The ChunkedLIFO rule is implemented similarly to ChunkedFIFO except with a linearizable stack.

The Ordered rule is implemented with a concurrent heap with additional locks to protect the inner worksets. Showing the correct-

ness of its implementation is beyond the scope of this paper because it uses commutativity conditions to reduce the granularity of the locking [20].

Although not a rule *per se*, the composition of a global and local rule, used to implement global and local orders in specifications, is implemented with worksets as well. There is one workset that implements the global rule and thread-local worksets that implement the local rule. Initial work is added to the global workset, while newly created work is added to the thread-local workset. New work is retrieved from the thread-local workset, if possible, and from the global workset otherwise. For proving correctness, this implementation can be viewed as a refinement of a chunked workset where chunks are never discarded but are instead refilled from the global workset.

4.4 Implementation and Optimization

In the Galois system, the specification language is implemented as a library-based domain-specific language in Java. Each rule is represented by a Java class that implements the corresponding workset. Figure 11 gives an example. Figure 12 gives the general form. The sequence of method calls produces an AST that is passed to the workset synthesizer.

Additionally, we take advantage of the semantics of rules to choose optimized workset implementations.

- **Use Serial**: As mentioned in Section 4.3, the inner worksets used by ChunkedFIFO and ChunkedLIFO are thread-local. The worksets generated from the local part of a specification are also thread-local. Thread-local worksets can be implemented with non-concurrent data structures that are typically more efficient than concurrent ones.
- **Ignore Size**: In certain cases, worksets require inner worksets to maintain an estimate of the number of items they contain. The chunked worksets use this to keep track of when a chunk is full. The Ordered workset uses these sizes to implement commutativity conditions. This overhead may be significant in concurrent worksets because keeping track of sizes may require atomic increments. When sizes are not needed, the size metadata and effort maintaining it may be removed.
- **Use Bounded**: When a chunked workset is used, each inner workset can be no larger than the chunk size. The inner worksets can be optimized for a bounded size rather than using dynamically sized data structures.

The synthesizer applies rewrite rules over the AST to detect the above cases and selects, if possible, implementations that are non-concurrent, do not keep track of their size or are bounded.

To implement these optimizations, the synthesizer introduces new rules (classes) to represent serial implementations of all the rules and bounded size implementations of the FIFO, LIFO and Random rules. It also adds an ignore size parameter to each rule, which determines if the implementation keeps track of its size. Then, the synthesizer traverses the AST (1) rewriting any rule after a chunked rule to use its serial implementation, (2) rewriting only rules after an Ordered rule to keep track of their sizes, and (3) rewriting rules after a chunked rule to use bounded implementations if possible, using the chunk size as the bound.

Table 1 summarizes the impact of these optimizations for the PFP application and a synthesized scheduler called BASE (described in more detail in Section 5). Positive numbers indicate how much slower a combination is relative to all optimizations on. We chose to highlight this application and scheduler because all the optimizations described above can be applied and the amount of work done per workset item is small, which increases the relative impact of an efficient workset implementation. From the table, it is clear that these optimizations on worksets have a significant and mostly beneficial impact on single-threaded and multi-threaded performance.

```
Lambda<T, Integer> indexer = new Lambda<T, Integer>() {
  public Integer call(T item) {
    return item.height;
  }
}
Priority.first(OrderedByMetric.class, indexer)
        .then(FIFO.class)
```

Figure 11. Concrete syntax of AS1 for PFP.

```
Priority.first(G1.class, args).then(...)
        .thenLocally(L1.class, args).then(...)
```

Figure 12. Concrete syntax of Global: G_1 ... Local: L_1 ... .

Ignore Size	Use Serial	Use Bounded	$t = 1$	$t = 8$
+	+	+	0.0	0.0
-	+	+	0.8	12.1
+	-	+	2.4	5.5
-	-	+	7.8	7.7
+	+	-	3.6	3.5
-	+	-	11.3	11.5
+	-	-	5.0	16.8
-	-	-	2.9	17.5

Table 1. Relative difference in percent (%) of the runtime of PFP and BASE scheduler on Shanghai machine varying synthesizer optimizations (+: on, -: off) relative to all optimizations on for one and eight threads.

5. Evaluation

We implemented the algorithms presented in Section 2 and parallelized them using the Galois system.[5] Figure 13 shows the datasets we used for each application. The shortest-path application uses atomic compare-and-set operations to update distances. The currently distributed implementation uses abstract locks [19], but compare-and-set operations are much faster for this application.

We modified the Galois system to support the scheduler specifications and synthesis algorithm described in Section 4. To evaluate our approach, we ran each algorithm with the following set of schedulers.

- **BASE**: This is the default scheduler used by the Galois system. It is a synthesized ChunkedFIFO with a chunk-size of 32. Each chunk is a thread-local LIFO.
- **FIFO, LIFO, RAND**: These schedulers are synthesized from the final rules FIFO, LIFO and Random. Application-specific schedulers typically use one of these schedulers as their lowest-level (final) scheduler.
- **WS-L, WS-F**: These schedulers are work-stealing with local LIFOs and FIFOs respectively. These schedulers were ported directly from the Fork-Join implementation in JSR166 and should appear in Java JDK 7. WS-L is widely used in many parallel systems.
- **BS-L, BS-F**: These schedulers use a bulk-synchronous strategy with global LIFOs and FIFOs respectively. A barrier is used to safely swap between queues concurrently.
- **AS1, AS2**: These schedulers are synthesized from the application-specific specifications in Figure 8.

[5] Available at `http://iss.ices.utexas.edu/galois`

We ran our experiments on three architectures:

- **Nehalem**: a Sun Fire X2270 machine running Ubuntu Linux 8.04.4 LTS 64-bit. It contains two 4-core 2.93 GHz Intel Xeon X5570 (Nehalem) processors. The two CPUs share 24 GB of main memory. Each core has a 32 KB L1 cache and a unified 256 KB L2 cache. Each processor has an 8 MB L3 cache that is shared among the cores.
- **Shanghai**: a machine also running Ubuntu Linux 8.04.4 LTS 64-bit. It contains four 4-core 2.7 GHz AMD Opteron 8384 (Shanghai) processors. Each core has a 64 KB L1 cache and a 512 KB L2 cache. Each processor has a 6 MB L3 cache that is shared among the cores.
- **Niagara**: a Sun T5440 machine running SunOS 5.10. It contains four 8-core 1.4 GHz Sun UltraSPARC T2 Plus (Niagara 2) processors. Each processor has a 4 MB L2 cache that is shared among the cores.

We used the Sun JDK 1.6.0_21 to compile and run the programs with a heap size of 20 GB. To control for JIT compilation, we ran each application four times within the same JVM instance and report only the last run.

The Galois system uses speculative parallelization, which introduces overheads from (1) using concurrent implementations of data structures and schedulers rather than their sequential counterparts, (2) acquiring locks to guarantee disjointness of neighborhoods, and (3) recording undo actions to implement rollback. The *serial* version of an application uses sequential data structures only and does not acquire locks or perform undo actions. The difference in performance between the serial version and the parallel, one-threaded version is the overhead of enabling speculative execution but never using it. This overhead can be significant for applications with short activities like PFP and SSSP.

Table 2 shows the runtimes for serial applications. Table 3 shows the parallel speedup over the best performing serial scheduler (shown in bold in Table 2). The runtimes for PTA exclude time to read input, perform offline-cycle detection and write results. This differs from the methodology of Mendez-Lojo *et al.*, which includes the time to perform offline-cycle detection [24]. For all other applications, runtimes exclude time to read input data or write results but may include sections of the application that are not parallelized. This portion of time is usually negligible, but for DT with the AS1 scheduler, this includes time to construct the oct-tree.

Empty entries indicate combinations of schedulers and applications that would either be redundant or perform significantly worse (by an order of magnitude or more) than the best serial version. For DT, the performance without randomizing the initial points is much worse than with randomization. Since the application does not create any new tasks, we evaluated the BASE and WS-L schedulers after including a timed phase that randomizes the input points. We omit the other non-random schedulers because they perform similarly after including this phase. For PTA, the RAND, LIFO and WS-L schedulers timed out. For SSSP, only the AS1 and AS2 schedulers are competitive.

An important result is that the best scheduler for serial execution tends to be the best for parallel execution as well. This supports the use case where users experiment with scheduling specifications within a sequential programming model and rely on a synthesis routine to generate efficient, concurrent schedulers that faithfully maintain the desired scheduling. Also, note the large swings between best and worst schedulers and recall that the missing entries for PFP, PTA and SSSP correspond to combinations that perform significantly worse than the recorded times. Choosing the wrong scheduler can have a drastic impact on performance.

Overall, the AS1 schedulers perform as well or better than any fixed-function scheduler for the same application. Surprisingly, the BASE scheduler, which is implemented by a global queue with fixed-size local stacks, performs relatively well across applications. DMR benefits from LIFO policies, while PFP, PTA and SSSP benefit from FIFO policies.

While direct comparison is not possible, these results are similar to previously reported results of application-specific schedulers on similar inputs for PFP [3], PTA [24] and SSSP [23].

Now, we turn to each application in more detail.

Delaunay Mesh Refinement Table 2 shows that for the serial implementation, the best performance is obtained by the LIFO scheduler. When a bad triangle is fixed, it may create a set of new bad triangles whose cavities overlap with the cavity of the original bad triangle. The LIFO scheduler exploits the potential temporal and spatial locality. However, both of the global LIFO scheduling policies, LIFO and BS-L, perform poorly in a parallel setting because the probability of conflicts increases if triangles close to each other in the mesh are processed speculatively in parallel. Figure 14 shows the relative number of aborted to total iterations on Shanghai. The trends are similar for the other two machines. Not surprisingly, global FIFO schedulers behave the same way. The BASE scheduler, which uses a global FIFO, ameliorates this problem to some extent by distributing chunks of work to each thread.

The best performance is obtained with the AS1 scheduler, which is implemented by a global workset processed in chunked FIFO order and local worksets maintained in LIFO order. This enables exploitation of locality while controlling the abort ratio, as can be seen in Figure 14.

Delaunay Triangulation DT does not create any new work. Its performance is governed by the initial work order, which should be randomized for best algorithmic performance. With randomization, the BASE and WS-L schedulers perform similarly. AS1 is significantly faster than the other two schedulers serially, but the performance difference dissipates as the number of threads increases. Recall that the AS1 scheduler is designed to increase spatial locality between activities. In speculative parallel execution, this scheduling strategy causes the abort ratio to increase, as can be seen in Figure 15.

Preflow-push PFP is an example of an algorithm whose performance is highly schedule-dependent because different schedulers result in dramatically different amounts of work. Figure 16 shows the number of iterations committed relative to the best performing serial version on Shanghai. Most schedulers result in twice as many iterations as the best serial version. LIFO and WS-L perform four times more iterations in some cases. Table 2 shows the impact of the varying work on the serial versions. The runtime of the fastest and slowest schedulers differ by a factor of more than three.

For a hand-parallelized implementation of PFP using the heuristics described in Section 2, Bader and Sachdeva reported a maximum speedup of about 2 with eight processors on an UltraSPARC II architecture with an input similar to that used here, which is refered to as an RMF graph [3].

Inclusion-based Points-to Analysis PTA is another example of a highly schedule-dependent algorithm. PTA performs a fixpoint computation, and for these computations, FIFO policies usually perform well because a variable gets to accumulate several updates before its value is propagated down-stream. LIFO policies perform poorly, and most versions time-out. BS-F, which alternates between two queues, does the best on this input. On other inputs, not shown here, BASE outperforms BS-F.

As noted earlier, these results are based on the implementation of Mendez-Lojo *et al.*, the first parallel implementation of PTA [24].

DMR	Triangle mesh of 550,000 triangles of which 261,100 are initially bad
DT	1,733,360 points generated from edge detection of a photograph
PFP	A flow network of 526,904 vertices arranged in 14 consecutive 194x194 frames with uniformly random capacities
PTA	Analysis of the gimp program
SSSP	Road network of western USA, weights are distances between locations: 6,262,104 vertices, 15,119,284 edges

Figure 13. Datasets used for each application.

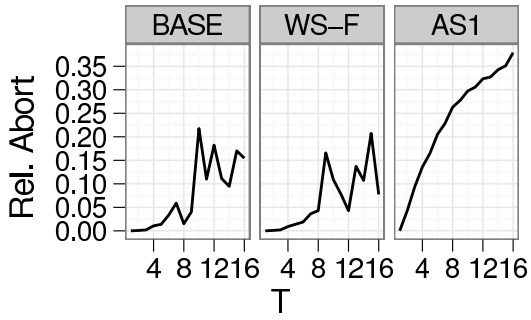

Figure 15. Relative number of aborted to total iterations for DT on Shanghai.

Single-source Shortest-path SSSP is a case where the generally accepted best serial scheduler, which is AS2 and is based on Dijkstra's algorithm, does not perform well in parallel. It has better theoretical algorithmic complexity, but the concurrent priority queue limits performance on most inputs. The difficulty in implementing such queues is one motivation for the delta-stepping order (AS1), which seeks to balance good order with efficient concurrent implementation. From experiments not shown here, it can be shown that the delta-stepping order performs only 1.2 times more work than Dijkstra's algorithm for this input, an overhead that is modest enough to overcome through parallelism.

Madduri *et al.* produced an implementation of delta-stepping for the Cray MTA-2 architecture [23]. On the same input used here, a road network of the western USA, they reported a maximum speedup of about 2 on sixteen processors.

6. Related Work

Although we synthesize schedulers from specifications, this is not the standard concurrent program synthesis problem considered in the literature [33, 34]. Scheduling specifications for unordered algorithms are inherently "fuzzy" (even FIFO scheduling of the workset can result in different executions depending on the speed of processors). Therefore, scheduling specifications are advice to the runtime system about how to bias scheduling decisions for efficiency, whereas in conventional program synthesis, implementations must satisfy specifications exactly.

Recent work has shown that more efficient workset implementations may be possible if the application is written so that it can tolerate duplicate work items [22, 26]. Exploiting this idempotence property would require changes to the definitions in Sections 3 and 4, but the implementation would be straightforward for final worksets. However, some non-final worksets, such as the chunked worksets, depend on the correspondence between adds and polls, which makes taking advantage of idempotence a challenge in these cases.

Another avenue of further research is to examine more algorithms to see if our current set of rules is adequate. Our FIFO and LIFO workset implementations would benefit from some of the memory-level optimizations.

Previously, Kulkarni *et al.* [18] explored different handwritten scheduling policies in the Galois system. However, the policies were a small number of fixed forms. In contrast, the schedulers described here are synthesized (not handwritten), they include policies studied in the literature such as delta-stepping, and they can be composed arbitrarily, which is important in practice.

The inspector-executor method [36] is a popular approach for parallelizing irregular applications in which dependences can be computed at runtime once the input has been supplied to the program. It is orthogonal to our work since it does not consider scheduling strategies for controlling the amount of work performed by the implementation, for enhancing locality, etc., which are the main considerations of this paper.

7. Conclusion

Scheduling order matters even in the sequential implementation of irregular algorithms. In the context of parallel implementations, prior work on scheduling assumed that activities are independent or that dependences are subsumed by fork-join control dependences, so current general-purpose runtime schedulers embody only a few simple scheduling policies. In contrast, handwritten schedulers for parallel implementations of irregular algorithms often use carefully crafted policies that trade-off excess work for increased parallelism, but they increase the burden of parallel programming and may be difficult to reuse for other algorithms.

In this paper, we proposed one solution to this problem by (1) developing a specification language for schedulers and (2) showing how efficient concurrent implementations can be synthesized from these specifications. This allows application programmers to focus on what they do well — coming up with scheduling policies — while delegating the tedious task of implementing the policy to the synthesizer and runtime system.

Acknowledgements: We thank David August (Princeton) for being a "good shepherd". We are also grateful to Milind Kulkarni (Purdue) and Roman Manevich (UT Austin) for their suggestions for improving the paper.

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *Proc. Symp. on Computational Geometry (SCG)*, pages 211–219, 2003.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] D. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. 18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS)*, September 2005.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[5] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[6] B. V. Cherkassy and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. In *Proc. Intl. Conf. on Integer Programming and Combinatorial Optimization (IPCO)*, pages 157–171, London, UK, 1995.

	BASE	RAND	LIFO	FIFO	WS-L	WS-F	BS-L	BS-F	AS2	AS1
					Nehalem					
DMR	12.88	14.80	**11.45**	13.09	11.51	13.27	12.76	13.17	15.56	11.62
DT	25.04					25.42				**14.78**
PFP	110.93	109.77	169.86	115.40	173.47	116.44	110.18	118.59		**45.94**
PTA	13.87	-	-	**12.58**	-	12.74	20.26	12.84		
SSSP	-	-	-	-	-	-	-	-	7.66	**4.96**
					Shanghai					
DMR	16.29	19.52	**13.55**	16.76	13.74	16.76	16.25	16.71	19.59	13.64
DT	43.40					43.55				**27.86**
PFP	237.04	210.57	320.24	237.17	314.53	234.13	216.50	217.67		**74.26**
PTA	19.99	-	-	18.80	-	**18.79**	26.44	18.82		
SSSP	-	-	-	-	-	-	-	-	11.08	**9.53**
					Niagara					
DMR	61.76	68.10	54.79	63.51	**53.84**	63.31	62.86	64.17	77.81	60.33
DT	178.21					179.00				**149.42**
PFP	787.05	734.27	1264.61	741.01	1297.71	775.04	720.20	827.07		**342.41**
PTA	59.17	-	-	57.73	-	57.30	76.16	**56.99**		
SSSP	-	-	-	-	-	-	-	-	33.84	**23.35**

Table 2. Runtimes of serial versions in seconds. In bold are the best serial times, the basis for the speedup numbers in Table 3. Entries with - timed out. Blank entries indicate invalid or redundant combinations.

	BASE	RAND	LIFO	FIFO	WS-L	WS-F	BS-L	BS-F	AS2	AS1
					Nehalem ($t \leq 8$)					
DMR	5.70	4.82	0.95	3.81	4.35	5.13	2.64	3.53	2.01	**6.15**
DT	2.21					2.09				**2.35**
PFP	1.30	0.71	0.20	1.15	0.72	2.30	0.37	0.89		**3.35**
PTA	2.83	-	-	3.53	-	2.05	2.37	**3.77**		
SSSP	-	-	-	-	-	-	-	-	0.61	**3.16**
					Shanghai ($t \leq 16$)					
DMR	7.85	3.43	0.95	3.74	6.94	7.53	1.91	3.83	2.32	**10.45**
DT	2.64					**2.65**				2.53
PFP	1.28	0.62	0.20	1.00	0.65	2.19	0.37	0.74		**2.56**
PTA	3.69	-	-	3.63	-	3.08	3.25	**5.03**		
SSSP	-	-	-	-	-	-	-	-	0.80	**3.04**
					Niagara ($t \leq 32$)					
DMR	18.77	5.95	0.89	6.81	11.47	18.53	3.60	5.89	3.59	**21.53**
DT	5.43					**5.48**				3.29
PFP	2.30	1.25	0.32	2.84	2.18	4.46	0.80	2.13		**5.92**
PTA	4.20	-	-	4.49	-	5.42	4.62	**6.16**		
SSSP	-	-	-	-	-	-	-	-	0.50	**2.33**

Table 3. Speedup over serial versions. In bold are the best speedups for each (application, machine) pair. Entries with - timed out. Blank entries indicate invalid or redundant combinations.

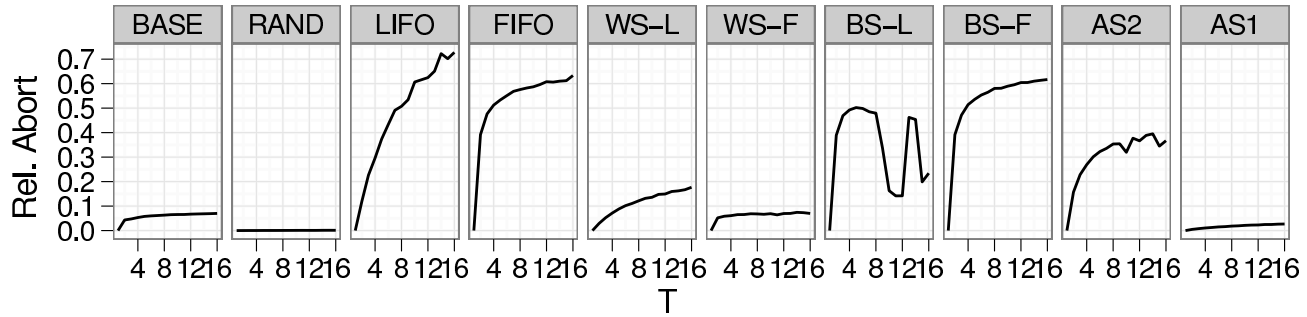

Figure 14. Relative number of aborted to total iterations for DMR on Shanghai.

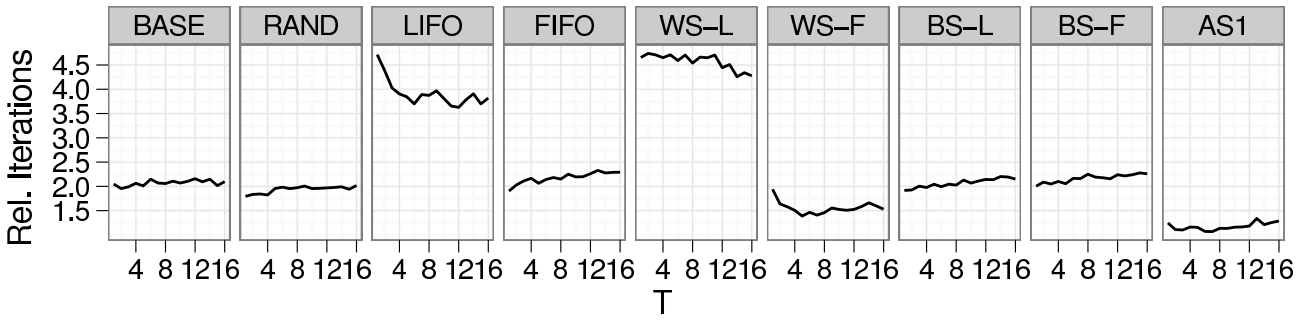

Figure 16. Relative number of committed iterations to that of the best performing serial version for PFP on Shanghai.

[7] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. Symp. on Computational Geometry (SCG)*, 1993.

[8] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, ii. *Discrete Comput. Geom.*, 4(5):287–421, 1989.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[10] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[11] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proc. Conf. on OpenMP in a new era of parallelism (IWOMP)*, pages 100–110, 2008.

[12] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[13] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[14] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 1993.

[16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[17] Intel Corporation. Intel threading building blocks 3.0. `http://threadingbuildingblocks.org`.

[18] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proc. Symp. on Parallelism in algorithms and architectures (SPAA)*, pages 217–228, New York, NY, USA, 2008.

[19] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI)*, 42(6):211–222, 2007.

[20] M. Kulkarni, D. Prountzos, D. Nguyen, and K. Pingali. Defining and implementing commutativity conditions for parallel execution. regular rech report TR-ECE-09-11, School of Electrial and Computer Engineering, Purdue University, August 2009.

[21] D. Lea. A Java fork/join framework. In *Proc. Conf. on Java Grande*, pages 36–43, 2000.

[22] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proc. Conf. on object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 227–242, 2009.

[23] K. Madduri, D. Bader, J. Berry, and J. Crobak. Parallel shortest path algorithms for solving large-scale instances. Technical Report GT-CSE-06-19, Georgia Institute of Technology, September 2006.

[24] M. Mendez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, October 2010.

[25] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *Proc. European Symp. on Algorithms (ESA)*, pages 393–404, 1998.

[26] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proc. Symp. on Principles and practice of parallel programming (PPoPP)*, pages 45–54, 2009.

[27] G. L. Miller. A time efficient Delaunay refinement algorithm. In *Proc. Symposium on Discrete Algorithms (SODA)*, pages 400–409, New Orleans, 2004.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[29] D. J. Pearce, P. H. J. Kelly, and C. L. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Intl. IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 3–12, 2003.

[30] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[31] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[32] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1996.

[33] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 136–148, New York, NY, USA, 2008.

[34] M. T. Vechev, E. Yahav, D. F. Bacon, and N. Rinetzky. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 456–467, New York, NY, USA, 2007.

[35] D. F. Watson. Computing the n-dimensional tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[36] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44, 1995.