Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs*

Milind Kulkarni,† Patrick Carribault, Keshav Pingali
University of Texas, Austin
{milind,pingali}@cs.utexas.edu, patrick@ices.utexas.edu

Ganesh Ramanarayanan, Bruce Walter
Kavita Bala,‡ L. Paul Chew
Cornell University, Ithaca, New York
graman@cs.cornell.edu, bjw@graphics.cornell.edu, {kb,chew}@cs.cornell.edu

ABSTRACT

Recent application studies have shown that many irregular applications have a generalized data parallelism that manifests itself as iterative computations over worklists of different kinds. In general, there are complex dependencies between iterations. These dependencies cannot be elucidated statically because they depend on the inputs to the program; thus, optimistic parallel execution is the only tractable approach to parallelizing these applications.

We have built a system called Galois that supports this style of parallel execution. Its main features are (i) set iterators for expressing worklist-based data parallelism, and (ii) a runtime system that performs optimistic parallelization of these iterators, detecting conflicts and rolling back computations as needed.

Our work builds on the Galois system, and it addresses the problem of scheduling iterations of set iterators on multiple cores. The policy used by the base Galois system is to assign an iteration to a core whenever it needs work to do, but we show in this paper that this policy is not optimal for many applications. We also argue that OpenMP-style DO-ALL loop scheduling directives such as chunked and guided self-scheduling are too simplistic for irregular programs. These difficulties led us to develop a general scheduling framework for irregular problems; OpenMP-style scheduling strategies are special cases of this general approach. We also provide hooks into our framework, allowing the programmer to leverage application knowledge to further tune a schedule for a particular application.

To evaluate this framework, we implemented it as an extension of the Galois system. We then tested the system using five real-world, irregular, data-parallel applications. Our results show that (i) the optimal scheduling policy can be different for different applications and often leverages application-specific knowledge and (ii) implementing these schedules in the Galois system is relatively straightforward.

*This work is supported in part by NSF grants 0719966, 0702353, 0615240, 0541193, 0509307, 0509324, 0426787 and 0406380, as well as grants from IBM and Intel Corportation.

†Milind is supported by a DOE HPCS Fellowship.

‡Kavita Bala is supported in part by NSF Career Grant 0644175.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA'08, June 14–16, 2008, Munich, Germany.

Copyright 2008 ACM 978-1-59593-973-9/08/06 ...$5.00.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms

Languages

Keywords

Optimistic Parallelism, Irregular Programs, Scheduling

1. INTRODUCTION

The majority of applications that will run on multicore processors are *irregular* applications that manipulate pointer-based data structures like trees and graphs, rather than regular applications that deal with arrays and dense matrices. Little is known about the nature of concurrency in irregular applications, let alone how to exploit this concurrency effectively on multicore processors.

Recent case studies of irregular programs have shown that many have a generalized data parallelism that manifests itself as iterative computations over worklists of various kinds [11]. Consider 2-D Delaunay mesh refinement [3], an important irregular code used in graphics and finite-element solvers. The input to the algorithm is an initial triangulation of a region in the plane, as shown in Figure 1. Some of the triangles in this mesh may be badly shaped (these are shown in red in Figure 1); if so, an iterative refinement procedure, shown in Figure 2, is used to eliminate them from the mesh. At each step, the refinement procedure (i) picks a bad triangle from the worklist, (ii) collects a bunch of triangles in the neighborhood of that bad triangle (called *cavity*, shown in blue in Figure 1), and (iii) re-triangulates that cavity. If this re-triangulation creates new badly-shaped triangles in the cavity, they are added to the worklist. The shape of the final mesh depends on the order in which bad triangles are processed, but it can be shown that every processing order halts, producing a final mesh without badly shaped elements. From this description, it is clear that bad triangles whose cavities do not overlap can be processed in parallel. Moreover, since each bad triangle is processed identically, this is a form of data parallelism. Abstractly, the worklist implements a *set*, and the data parallelism arises from computations performed on each element of that set.

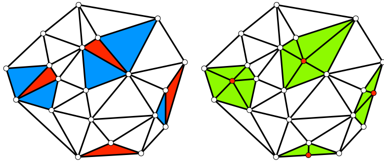


Figure 1: Mesh refinement.

```
1: Mesh m = /* read in initial mesh */
2: WorkList wl;
3: wl.add(mesh.badTriangles());
4: while (wl.size() != 0) {
5:   Element e = wl.get(); //get bad triangle
6:   if (e no longer in mesh) continue;
7:   Cavity c = new Cavity(e);
8:   c.expand();
9:   c.retriangulate();
10:  mesh.update(c);
11:  wl.add(c.badTriangles());
12:}
```

Figure 2: Pseudocode of the mesh refinement algorithm

Exploiting data parallelism in irregular programs can be more complex than exploiting data parallelism in array programs. Data parallelism in array programs usually manifests itself in DO-ALL loops (*i.e.*, FORTRAN-style DO loops over integer intervals in which the iterations can be proven statically to be independent). Data parallelism in irregular programs often manifests itself in iteration over sets, but the iterations are not necessarily independent. Although static analysis techniques such as points-to and shape analysis [6, 18] can be used in some cases to prove independence, there may be complex dependences between iterations, as in the case of Delaunay mesh refinement. Static analysis fails to discover the potential data parallelism in these cases.

One promising approach is to use *speculative* or *optimistic* parallel execution. While there is a large body of work in this area [11, 16, 22], the work described in this paper is based on the Galois system we have previously developed [11]. In this system, data parallelism is expressed using *set iterators*, and these iterators can be executed concurrently by some number of threads that pull elements one at a time from the underlying worklist. To ensure that the results of concurrent execution are consistent with the sequential semantics of the program, there is a runtime system that detects conflicting method accesses made by different concurrently executing iterations. If a conflict is detected, one of the conflicting iterations is rolled back and any side-effects it may have had on shared objects are undone. This mechanism is explained in further detail in Section 2.2.

Figure 3 shows the Galois version of Delaunay mesh refinement. There is no *a priori* ordering on set elements, so a sequential execution of the set iterator is permitted to execute the iterations in any order. Although the program can be executed sequentially, the iterator provides a hint to the Galois runtime system that it may be profitable to execute the iterations in parallel. In the Galois system's execution model, a master thread begins the execution of the program. When this master thread encounters an iterator, it enlists the assistance of some number of worker threads to execute iterations concurrently with itself. The assignment of iterations to threads is under the control of a scheduling policy implemented by the runtime system; in the current Galois implementation, this assignment is done dynamically to ensure load balancing. All threads are synchronized using barrier synchronization at the end of the iterator. As mentioned above, the runtime system takes care of detecting conflicting accesses made to global objects, and recovering from unsafe accesses. The Galois system has been used successfully to parallelize a number of irregular applications [10].

```
1: Mesh m = /* read in initial mesh */
2: Worklist wl;
3: wl.add(mesh.badTriangles());
4: for each e in wl do {//optimistic set iterator
5:   if (e no longer in mesh) continue;
6:   Cavity c = new Cavity(e);
7:   c.expand();
8:   c.retriangulate();
9:   m.update(c);
10:  wl.add(c.badTriangles());
11:}
```

Figure 3: Delaunay mesh refinement using set iterator

1.1 Scheduling iterations of set iterators

In this paper, we address the problem of scheduling the iterations of Galois set iterators for parallel execution. In principle, these iterations can be assigned arbitrarily to different cores and each core has the freedom to execute the iterations mapped to it in any order. In practice, we have found that even for sequential execution, the performance of the program is affected dramatically by the scheduling policy. Consider a sequential execution of Delaunay mesh refinement. If newly created bad triangles (line 10 in Figure 3) are processed immediately, we get the benefits of exploiting temporal and spatial locality. For this reason, hand-written implementations of Delaunay mesh refinement use a stack to implement the worklist. A scheduling policy that picks a bad triangle at random from the current worklist will not exploit locality and may therefore perform poorly. Just how much performance is lost depends on the size and shape of the mesh, cache parameters, etc. but the experiments reported in Section 4 show that the slow-down over the LIFO schedule can be more than 33% for even moderately sized meshes. Paradoxically, other applications such as Delaunay *triangulation* [5] suffer enormous slow-downs if the schedule tries to exploit locality. In Section 4, we show that using a locality-aware schedule for this problem can triple the execution time compared to using a random schedule!

Even for the same application, a good sequential scheduling strategy may be bad for parallel execution. For *parallel* Delaunay mesh refinement, using a stack (with atomic push and pop operations) to implement the worklist can double execution time compared to using the randomized scheduling policy, as we show in Section 4. This has nothing to do with the overhead of accessing the global worklist since both scheduling strategies involve the same number of accesses; instead, it turns out that there is significant misspeculation if the worklist is implemented as a stack.

This discussion shows that the problem of scheduling iterations of Galois set iterators is considerably more complex than the more familiar problem of scheduling iterations of DO-ALL loops in regular programs. In particular, we have found that the relatively simple scheduling policies in OpenMP for supporting scheduling of DO-ALL loops [13] are not adequate for scheduling iterations of Galois set iterators. To ease the implementation of application-specific schedules, we have designed a general scheduling framework and have used it to implement a number of specific scheduling policies in the Galois system.

1.2 Organization of paper

The rest of this paper is organized as follows. In Section 2, we give a high-level description of the Galois system, the basis for our work. In Section 3, we describe our scheduling framework and discuss how it is integrated with the Galois system. In Section 4, we describe experimental results for several real-world irregular applications: Delaunay mesh refinement [3], Delaunay triangulation [5],

the Boykov-Kolmogorov maxflow algorithm (used in image segmentation) [1], the preflow-push algorithm [4] for maxflow, and agglomerative clustering [23]. We use our scheduling framework to evaluate a number of different schedules for each application, illustrating the effects of scheduling on performance and the efficacy of our framework. We conclude in Section 5 with a discussion of lessons learnt as well as future research directions.

2. THE GALOIS SYSTEM

In this section, we describe the Galois system at a high level to provide background for the rest of the paper. The Galois programming model [11] is a concurrent, object-based shared-memory model that can be implemented on top of an object-oriented language like Java. The design is based on the belief that programmers should write code with well-understood sequential semantics (we call this the *client code*), while the complexity of parallel programming is hidden within library code and the runtime system.

2.1 Client code

The client code is not explicitly parallel; instead data parallelism is implicit, and is packaged into two constructs called optimistic set iterators.

- **Set iterator: for each e in Set S do B(e)**
  The loop body B(e) is executed for each element e of set S. Since set elements are not ordered, this construct asserts that in a serial execution of the loop, the iterations can be executed in any order. There may be dependences between the iterations, as in the case of Delaunay mesh generation, but any serial order of executing iterations is permitted. When an iteration executes, it may add elements to S.
- **Ordered-set iterator: for each e in Poset S do B(e)**
  This construct iterates over a partially-ordered set (Poset) S. It is similar to the Set iterator above, except that any execution order must respect the partial order imposed by the Poset S.

Figure 3 shows client code for Delaunay mesh refinement. The well-defined sequential semantics makes it easier to write, understand, and debug the client code. The runtime system exploits the data-parallelism by speculatively executing iterations of set iterators in parallel, as described in Section 1. The client program may contain nested iterators, so the current Galois execution model "flattens" inner iterators, always executing them sequentially. For ordered set iterators, the runtime system ensures that the iterations commit in the set order.

2.2 The Galois run-time and class libraries

In this execution model, the key hurdle is ensuring that the parallel execution respects the sequential semantics of the iterators. This is difficult because each iteration may invoke methods on shared objects, requiring coordination of concurrent invocations.

The Galois approach lifts the burden of parallelism from the programmer, placing it instead on the class libraries and the runtime system. The Galois run-time system detects conflicts through *commutativity checks*. Intuitively, concurrent accesses by two iterations to a given object do not conflict if the methods they invoke on that object commute. This is a property of an object's interface, and is expressed through an annotation of its class definition. Commutativity checks and annotations are discussed in further detail in [11].

We proposed an alternate means of conflict detection in [10], which leverages partitioned data structures (for example, the mesh in Delaunay mesh can be geometrically partitioned). If two iterations touch the same partition of a partitioned data structure, a conflict is triggered and one is rolled back. We call this scheme *partition locking*. For example, in Delaunay refinement if two iterations each access triangles from the same *partition* of the mesh, there is a conflict even if they touch distinct triangles. Partition locking is less precise than commutativity checking, but we have found that it can have significantly lower overhead, and several of the applications studied in Section 4 use this means of conflict detection.

When a conflict is detected, one or more iterations must be rolled back. To undo the effects of an iteration, the run-time executes a series of *undo methods*. Each method of a class in the Galois system has an associated undo written by the class implementor(for example, the undo of *add(x)* in a set is *remove(x)*). These undo methods are recorded during execution and used to perform the rollback when a conflict is detected.

3. SCHEDULING FRAMEWORK

In principle, the iterations of an unordered set iterator can be executed in any order, and the runtime system has complete freedom in how it assigns iterations to processors for execution. However, the performance of the program may depend critically on the scheduling policy used to execute the loop for the following reasons.

1. *Algorithmic effects*: In some irregular applications, the scheduling policy can affect the efficiency of an algorithm or data structure used by the application. For example, a commonly used algorithm for Delaunay triangulation, described in Section 4.2, uses a data structure called the history DAG whose operations have good expected-case complexity but bad worst-case complexity (*cf* the behavior of a binary search tree). Rewriting the application to use a different algorithm or data structure is one option, but this may not always be possible.

2. *Locality*: To promote temporal and spatial locality, it is desirable that iterations that touch the same portion of a global data structure be assigned to the same core and executed contemporaneously. For example, locality is improved in Delaunay mesh refinement if bad triangles close to each other in the mesh are assigned to the same core and are processed at roughly the same time. Unfortunately, there are also algorithms, such as Delaunay *triangulation*, in which exploiting locality may trigger worst-case behavior of the underlying data structures(*cf* inserting sorted elements into a binary search tree).

3. *Conflicts*: Iterations that are likely to conflict should not be scheduled for concurrent execution on different cores. For example, in Delaunay mesh refinement, bad triangles that are close to each other in the mesh should not be processed simultaneously on different cores since their cavities are likely to overlap.

4. *Load-balancing*: The assignment of work to cores should attempt to balance the computation load across cores. This can be difficult in irregular programs because work is often dynamically created, and because load-balancing may conflict with locality exploitation. For example, in Delaunay mesh refinement, load-balancing can be accomplished by assigning each core a randomly chosen bad triangle whenever the core needs work [11]. However, this policy limits locality.

5. *Contention and access overhead for global data structures*: Finally, a good scheduling policy may be able to reduce contention and access overhead for global data structures such as worklists.

3.1 Comparison with scheduling of DO-ALL loops

These issues make the problem of scheduling set iterators in irregular programs much more complex than the well-studied problem of scheduling DO-ALL loops in regular programs. DO-ALL loops are usually used to manipulate dense arrays and are often written so that executing iterations in standard order exploits spatial locality. There are few if any algorithmic effects to worry about, and there are no conflicts between different iterations, so the main concerns are load-balancing, and reducing contention and access overhead for global data structures. Therefore, simple policies suffice.

For example, OpenMP supports three scheduling policies for DO-ALL loops: *static*, *dynamic*, and *guided*. Static schedules assign iterations to cores in a cyclic (round-robin) fashion before loop execution begins; to exploit locality, the programmer can specify that the assignment be done in a block-cyclic fashion in *chunks* of c contiguous iterations at a time. Static schedules can lead to load imbalance if the execution times of iterations vary widely. In dynamic scheduling, the system assigns iterations to cores whenever the core needs work; this is good for load-balancing, but if each iteration does only a small amount of work, the overhead of assigning iterations dynamically can be substantial. To ameliorate this problem and to permit locality exploitation, the programmer can ask the system to hand out chunks of c contiguous iterations at a time. Guided self-scheduling is a more sophisticated form of dynamic scheduling in which the chunk size is decreased gradually towards the end of loop execution.

These policies are not adequate for irregular codes. Most irregular codes such as Delaunay mesh refinement create work dynamically, so static scheduling is not useful. There is no *a priori* ordering on the iterations of an unordered set iterator, so chunking is not well-defined. One interpretation of chunking is the following: when a core asks for work, the scheduler gives it some number of elements from the worklist, rather than just a single element. However, worklist elements are not ordered in any way, so there is no reason to believe that this kind of chunking promotes locality.

3.2 Our approach

In our framework, a fully defined schedule for a set iterator requires the specification of three *scheduling functions* (see Figure 4).

1. *Clustering:* A cluster is a group of iterations all of which are executed by a single core. The clustering function maps each iteration to a cluster.
2. *Labeling:* The labeling function assigns each cluster of iterations to a core. A single core may execute iterations from several clusters, as shown in Figure 4.
3. *Ordering:* The ordering function maps the iterations in the different clusters assigned to a given core to a linear order that defines the execution order of these iterations.

To understand these scheduling functions, it is useful to consider how the static and dynamic scheduling schemes supported by OpenMP map to this framework. For a static schedule with chunk size c, the clustering function partitions the iterations of the DO-ALL loop into clusters of c contiguous iterations. The labeling function assigns these clusters to cores in a round-robin fashion, so each core may end up with several clusters. The ordering function can be described as *cluster-major* order since a core executes clusters in lexicographic order, and it executes all iterations in a cluster before it executes iterations from the next cluster. Notice that for static schedules of DO-ALL loops, the iteration space, clusters and the three scheduling functions are known before the loop begins execution. For dynamic schedules on the other hand, some of these are defined incrementally as the loop executes. Consider a dynamic schedule with chunk size c. As in the case of static schedules, the clustering function partitions iterations into clusters of c contiguous iterations, and this function is defined completely before the loop begins execution. However, the labeling function is defined incrementally during loop execution since the assignment of clusters to cores is done on demand. The ordering function is cluster-major order, as in the static case. In general therefore, Figure 4 should be viewed as a post-execution report of scheduling decisions, some of which may be made before loop execution, while the rest are made during loop execution.

Scheduling in irregular programs can be viewed as the most general case of Figure 4 in which even the iteration space and clusters are defined dynamically. In applications like Delaunay mesh refinement, elements can be added to the worklist as the loop executes, and this corresponds abstractly to the addition of new points to the iteration space of the loop during execution. It is convenient to distinguish between the *initial iterations* of the iteration space, which exist before loop execution begins, and *dynamically created iterations*, which are added to the iteration space as the loop executes. The initial iterations may be clustered before loop execution begins, but the runtime system may decide to create new clusters for dynamically created iterations, so both the iteration space and clusters may be defined dynamically.

A clustering/labeling/ordering *policy* is a systematic procedure for generating some category of clustering/labeling/ordering functions. For example, a random clustering policy assigns iterations to clusters randomly, and may produce different assignments of iterations to clusters (*i.e.*, different clustering functions) in different runs. We now describe a number of policies for clustering, labeling and ordering that we have found to be useful in our application studies.

Clustering.

We have implemented the following policies for assigning initial iterations to clusters.

- *Chunking:* This policy is defined only for *ordered-set* iterators, and it is a generalization of OpenMP-style chunking of DO-ALL loops. The programmer specifies a chunk size c, and the policy clusters c contiguous iterations at a time.
- *Data-centric:* In some applications, there is an underlying global data structure that is accessed by all iterations. Partitioning this data structure between the cores often leads to a natural clustering of iterations; for example, if the mesh in Delaunay mesh refinement is partitioned between the cores, the responsibility for refining a bad triangle can be given to whichever core owns the partition that contains that bad triangle[1] [10]. The data-centric policy is similar in spirit to what is done in High Performance FORTRAN (HPF) [8, 17]. The number of data partitions is specified by the programmer or is determined heuristically by the system.
- *Random:* In some applications, it may be desirable to assign initial iterations to clusters randomly. The number of initial clusters is specified by the programmer or is chosen heuristically.
- *Unit:* Each iteration is in a cluster by itself. This can be considered to be a degenerate case of random clustering in which each cluster contains exactly one iteration. This is the default policy.

[1] Note that if the cavity of the bad triangle spans multiple partitions, the core refining that triangle will need to access several partitions.

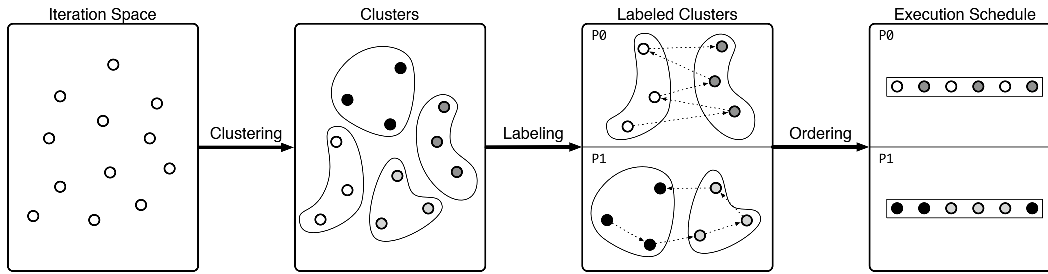


Figure 4: Scheduling framework

For applications that dynamically create new iterations, the policy for a new iteration can be chosen separately from the decision made for the initial iterations. Dynamically created iterations can be clustered using the *Data-centric*, *Random*, and *Unit* policies described above. In addition, we have implemented one policy specifically for dynamically created iterations.

- *Inherited:* If the execution of iteration i_1 creates iteration i_2, i_2 is assigned to the same cluster as i_1. This particular policy is interesting because it lends itself to an efficient implementation using *iteration-local worklists*. Newly created work gets added to the iteration local worklist, which can be accessed without synchronization.

An aborted iteration, by default, is treated as a dynamically created iteration. For example, if a schedule uses the *inherited* clustering policy, an aborted iteration will be assigned to the same cluster it was in previously, but if it uses the *random* policy, an aborted iteration will be assigned to a random cluster.

Labeling.

Labeling policies can be static or dynamic. In static labeling, every cluster is assigned to a core before execution begins. In dynamic labeling, clusters are assigned to cores on demand.

We have implemented the following static labeling policies.

- *Round-robin:* For ordered-set iterators, clusters can be assigned to cores in a round-robin fashion. This is similar to what is done in OpenMP.
- *Data-centric:* If clustering is performed using a data-centric policy, the cluster can be assigned to the same core that own the corresponding data partition. This promotes locality and also reduces the likelihood of conflicts because cores work on disjoint data for the most part.
- *Random:* Clusters are assigned randomly to cores.

We have also implemented the following dynamic policies.

- *Data-centric:* This is implemented using over-decomposition (*i.e.* the underlying data structure is divided into more partitions than there are cores). Clustering is done using the data-centric policy. When a core needs work, it is given a data partition and its associated cluster of iterations. The distribution of code and data can be implemented by a centralized scheduler or it can be implemented in a decentralized way using work-stealing.
- *Random:* Clusters are assigned randomly to cores.
- *LIFO/FIFO:* These policies can be used when clusters are created dynamically. For example, LIFO labeling means when a core needs work it is given the most recently created cluster.

Ordering.

The ordering policy is determined by the following decisions.

1. Given a choice between some number of clusters, which cluster should be chosen for execution? We have implemented random, lexicographic order (for ordered-set iterators), and LIFO/FIFO (for dynamically created clusters).
2. Once a cluster is picked, what order should iterations in that cluster be executed in? We have implemented random, lexicographic order (for ordered-set iterators), and LIFO/FIFO (for dynamically created iterations in the current cluster).
3. When should execution switch between clusters (*i.e.* how are clusters interleaved)?
    - *Cluster-major order:* All iterations in the current cluster are executed before those from a different cluster are executed.
    - *Switch-on-abort:* Iterations are executed from the current cluster till some iteration aborts. At that point, execution switches to a different cluster.
    - *Random:* Execution switches between clusters at random.

3.3 Implementation

Upto this point, we have presented a general conceptual framework for scheduling irregular, data-parallel applications. For the framework and system to be truly useful, it must be easy for a programmer to implement application-specific scheduling policies. Recall that scheduling of iterations from the worklist is handled by the Galois runtime system. By hiding the worklist in the optimistic iterator construct, the run-time can provide different scheduling policies with minimal intervention from the user.

The object that controls scheduling in the Galois run-time system is the *GaloisScheduler*. It provides methods which specify how a thread obtains a new piece of work (essentially performing the actions of the labeling policy and ordering policy), and how a core adds new work to the iterator (performing the actions of the clustering policy). By subclassing the GaloisScheduler and selectively overriding these methods, it is straightforward for a programmer to implement any schedule he or she desires. Our implementations of the policies described above use this technique.

Given a set of GaloisScheduler objects that implement different scheduling policies, the question then becomes how a programmer can specify which policy to use for his or her program. One way of setting this policy is through compiler directives similar to OpenMP pragmas: the data-parallel loop is annotated to indicate the policy the programmer desires. Unfortunately, this is too restrictive for our purposes. By limiting the policies to those provided by compiler directives, programmers lose the ability to specify custom scheduling policies (which are often necessary, as we see in Section 4.5).

Instead, we take a programmatic approach. When instantiating the Galois runtime system, a programmer can pass in a particular

GaloisScheduler object to set a scheduling policy. This requires minimal changes to the user code, comparable to compiler directives, but retains the full flexibility of the scheduling framework. The Galois system provides several common policies which provide acceptable performance for a number of applications (see Section 4.6).

4. EVALUATION

We have evaluated our scheduling approach on five irregular applications. The scheduling framework described in Section 3 can be used to implement a vast number of policies and it is both infeasible and pointless to evaluate all these policies on all benchmarks. Instead, we studied the algorithms and data structures in these applications, and determined a number of interesting scheduling policies for each one. We then implemented these policies in the Galois system and measured the performance obtained for that application using each of these policies.

The machine we used in our experiments is a dual-processor, dual-core 3.0 GHz Xeon system with 16KB of L1 cache per core and 4MB of L2 cache per processor. This particular system exhibits performance anomalies due to automatic power management; to eliminate these, we downclocked the cores to 2 GHz. Our implementation of the Galois system, as well as the scheduling framework described in this paper, is in Java 1.6. To take into account variations in parallel execution, as well as the overhead of JIT compilation, each experiment was run 5 times under a single JVM instance, and the fastest execution time was recorded. In an attempt to minimize the effects of GC on running time, a full GC was performed before each execution. We used Sun's HotSpot JVM, which was run with a 2GB heap.

4.1 Delaunay Mesh Refinement

This application is described in Section 1. The input data was generated using Shewchuk's Triangle program [20]. It had 100,364 triangles and boundary segments, of which 47,768 were bad.

Scheduling issues.

As discussed in Section 3, the scheduling policy can affect performance because of algorithmic effects, locality, conflicts, load balancing, and overhead. In mesh refinement, algorithmic effects are minor: the total number of bad triangles that are created dynamically depends on the scheduling policy but the variation in this number is small. The final mesh depends on the order in which bad triangles are refined, and although different orders perform different amounts of work, the variation in the amount of work is small. Furthermore, the cost of getting work from the worklist is relatively small compared to cost of an iteration, so the effect of scheduling overhead is small. Therefore, the main concerns are locality, conflicts, and load balancing.

A significant feature of this application is that when the cavity of a bad triangle is re-triangulated, a number of new bad triangles may be created in that cavity. These new triangles will be (i) in the same region of the mesh as the original bad triangle, and (ii) near one another in the updated mesh. To exploit temporal and spatial locality, these new triangles should be processed right away. However, if these triangles are refined concurrently, their cavities are likely to overlap and the abort ratio will be high.

Evaluation.

The baseline sequential implementation, called *seq* in this discussion, uses a LIFO scheduling policy, implemented using a stack as the worklist to exploit spatial and temporal locality. In all the parallel implementations discussed in this section, the mesh is partitioned between the cores, and conflict detection is performed using partition locking rather than commutativity checks, as described in Section 2. This ensures that all parallel versions use the same mechanism for conflict checks, so the only difference between them is the scheduling policy[2].

We evaluated four different parallel schedules:

- *default* – This is the default schedule used by the base Galois system: the worklist is centralized, and a core is given one bad triangle, chosen at random, on demand. In terms of the scheduling framework introduced in Section 3, we can describe this schedule as follows: it uses the *unit* clustering policy for both initial and dynamically generated iterations, and the labeling policy is dynamic and random. Obviously, there are no ordering concerns in this policy.
- *stack* – This policy is similar to *default*, except that the worklist is stack-like, so the labeling policy is dynamic and LIFO rather than random. This policy mimics the scheduling policy of *seq*.
- *part* – This schedule uses data-centric clustering for both initial iterations and dynamic iterations, with 4 times as many partitions as processors. The labeling policy is also data-centric. The cluster interleaving used is *switch-on-abort*. Within a cluster, iterations are ordered in a LIFO manner, processing newer work first. The mesh is partitioned using breadth-first search, a simplified version of the Kernighan-Lin method [7].
- *hist* – This schedule uses a *random* clustering policy for initial iterations, with each cluster containing 16 elements. Dynamically created iterations use *inherited* clustering; newly created work is assigned to the cluster that is currently being processed. The labeling policy is the same as in *default*. Because the labeling policy is dynamic, there is no cluster interleaving. Iteration ordering within a cluster is LIFO.

We compared the parallel implementation using these schedules with the sequential implementation. Figure 5(a) gives the wallclock time, in seconds, for *seq* as well as the four parallel versions on different numbers of cores. Figure 5(b) shows the speedup of the five parallel versions relative to sequential execution time. We see that *stack* has the worst performance, achieving a speedup of 1.2 on 4 cores, while *hist* performs the best, achieving a speedup of 3.3.

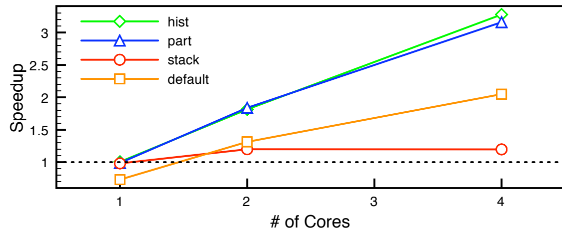
There are a number of interesting points to note in these results. First, we note that *seq*, which is the sequential implementation that uses LIFO scheduling, and *stack*, which is the parallel implementation of LIFO-like scheduling, perform almost identically on one core; since *stack* is a parallel code, it has a small additional overhead even when run on a single core. Both versions exploit locality and therefore outperform the *default* version, which uses randomized scheduling. The fact that *hist* also performs well on one core shows that most of the locality benefits can be obtained by focusing on one bad triangle from the original mesh at a time, and repeatedly eliminating all new bad triangles created in its cavity before moving on to a different bad triangle from the original mesh.

Interestingly, single-core performance does not always translate to parallel performance. While *default* is slower than *stack* on a single core, it is faster on 4 cores. This is likely due to speculation conflicts, as discussed before. To investigate this, we measured the *abort ratio*, the percentage of executed iterations which are rolled back. A high abort ratio is indicative of significant mis-speculation

[2]Note that the scheduling policy does not need to be cognizant of the data partitioning (for example, bad triangles can still be assigned randomly to cores), although we would expect to obtain some locality benefits if the scheduling policy was data-centric.

Schedule	1 core	2 cores	4 cores	
	Exec. Time	Exec. Time	Exec. Time	Abort Ratio
seq	11.495	—	—	—
default	15.724	8.754	5.609	19.64%
stack	11.721	9.584	9.603	96.97%
part	11.634	6.255	3.639	5.79%
hist	11.435	6.338	3.508	7.19%

(a) Execution time (in seconds) and abort ratios



(b) Speedup vs. # of cores

Figure 5: Results for Delaunay mesh refinement

in the program, which may reduce performance. There is no direct correlation between abort ratio and performance: some iterations abort soon after starting (essentially a busy-wait), while others abort towards the end, resulting in more lost work. The rightmost column of Figure 5(a) shows the abort ratio for the parallel schedules on 4 cores. From these numbers, we see that *stack* has a very high abort ratio, as expected. By processing triangles chosen at random, *default* avoids this problem, and the gain in concurrency outweighs the cost in lost locality.

Both *part* and *hist* perform well in terms of locality and speculation behavior. They both execute iterations in a LIFO manner within a cluster, and their clustering policies ensure that newly created iterations are immediately executed, leading to good locality. They also both exhibit a low abort ratio. In the case of *part*, this is because of reduced mis-speculation. On the other hand, *hist* uses the same random scheduling as *default* to avoid excessive aborts. Unsurprisingly, the two schedules perform similarly, despite very different behaviors. We conjecture that *hist* is a better schedule than *part* due to better load-balancing. *part* uses a static labeling, so it cannot correct for load imbalance between processors. *hist* leverages dynamic labeling to achieve load balance.

4.2 Delaunay Triangulation

The second benchmark we studied is Delaunay *triangulation*, the creation of a Delaunay mesh, given a set of input points. In general, this mesh may have bad triangles, which can be eliminated using the refinement code discussed in Section 4.1.

Pseudocode for this algorithm is shown in Figure 6. The main loop iterates over the set of points, inserting a new point into the current mesh at each iteration to create a new mesh that satisfies the Delaunay property. When all the points have been inserted, mesh construction is complete. To insert a point p into the current mesh, the algorithm determines the triangle t that contains point p (line 6), then splits t into three new triangles that share point p as one of their vertices (line 7). These new triangles may not satisfy the Delaunay property, so a procedure called *edge flipping* is used to restore the Delaunay property. Edge flipping examines each edge of the newly created triangles (lines 9-15); if any edge is non-Delaunay, the edge is flipped, removing the two non-Delaunay triangles and replacing them with two new triangles (line 12). The edges of these newly created triangles are examined in turn (line 13). When this loop terminates, the resulting mesh is once again a Delaunay mesh.

```
 1: Mesh m = /* initialize with one surrounding triangle */
 2: Set points = /* read points to insert */
 3: Worklist wl;
 4: wl.add(points);
 5: for each Point p in wl {
 6:   Triangle t = m.surrounding(p);
 7:   Triangle newSplit[3] = m.splitTriangle(t, p);
 8:   Worklist wl2;
 9:   wl2.add(edges(newSplit));
10:   for each Edge e in wl2 {
11:     if (!isDelaunay(e)) {
12:       Triangle newFlipped[2] = m.flipEdge(e);
13:       wl2.add(edges(newFlipped))
14:     }
15:   }
16: }
```

Figure 6: Pseudocode for Delaunay triangulation

To locate the triangle containing a given point, we use a data structure called the *history DAG* [5]. Intuitively, this data structure can be viewed as a ternary search tree. The leaves of the DAG represent the triangles in the current mesh. When a triangle is split (line 7), the three new triangles are added to the data structure as children of the original triangle. The only twist to this intuitive picture is that when an edge is flipped (line 12), the two new triangles are children of both old triangles, so the data structure is a DAG in general, rather than a tree. If the DAG is more or less balanced, point location can be performed in $O(log(N))$ time where N is the number of triangles in the current mesh. However, if the data structure becomes long and skinny, point location can take $O(N)$ time, resulting in poor performance. To avoid this worst-case behavior, Guibas *et al* recommend inserting points in random order rather than in a spatially coherent order.

Opportunities for exploiting parallelism.

This algorithm exhibits nested iterators (over *wl* and *wl2*). We parallelize the outer iterator, as in the default policy of the Galois system, inserting multiple points in parallel. Inserting a new point only affects the triangles in its immediate neighborhood, so inserting multiple points in parallel can be profitable, as long as they are sufficiently far apart in the geometry. Conflicts can occur when two threads attempt to manipulate the same triangles in the mesh (lines 7 and 13). Notice that this algorithm does not add elements to the worklist dynamically.

Scheduling issues.

For this application, algorithmic effects are the most important, since it is critical to avoid worst-cast behavior of the history DAG. The best sequential implementation (*seq*) uses a random worklist [5]. In the parallel implementation, we can exploit temporal and spatial locality if points are inserted in sorted order. Unfortunately, this can lead to worst-case behavior of the history DAG.

Evaluation.

The input data for our experiments is a set of 75,000 random, uniformly-distributed points; the final mesh has roughly 150,000 triangles.

We evaluated three different parallel schedules:

- *default* – The default Galois schedule. Note that this approximates the schedule used by the sequential implementation.
- *part* – The points are partitioned geometrically. The schedule uses data-centric clustering for the initial iterations, with 8 times as many clusters as cores. The labeling policy is also data-centric. The ordering is cluster-major; within each cluster, the ordering is random. Intuitively, this scheduling policy

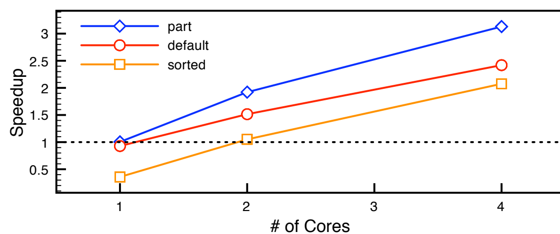
tries to exploit locality when clustering iterations, but it does not try to exploit locality when executing a given cluster.

- *sorted* – This schedule is similar to the *part* schedule except that the ordering within each cluster is the (geometrically) sorted order rather than random. Intuitively, this scheduling policy tries to exploit locality when creating clusters and also when executing iterations in each cluster.

Figure 7(a) gives the overall execution time on different numbers of cores, as well as the abort ratio on four cores. Figure 7(b) shows the speedup relative to *seq* for the various schedules.

Schedule	1 core	2 cores	4 cores	
	Exec. Time	Exec. Time	Exec. Time	Abort Ratio
seq	17.623	—	—	—
default	18.982	11.634	7.284	3.54%
part	17.555	9.174	5.631	0.34%
sorted	49.250	16.770	8,491	0.67%

(a) Execution time (in seconds) and abort ratios



(b) Speedup vs. # of cores

Figure 7: Results for Delaunay triangulation

The *sorted* schedule exhibits the worst performance of all the evaluated schedules. Although sorting the points achieves better locality in the mesh, these results illustrate the tradeoff described earlier: sorting the points is good for locality but it leads to a poorly shaped history-DAG and affects algorithmic performance. However, abandoning locality completely is not the best solution either. The best performance is achieved with the *part* schedule. This achieves a good balance between locality in the mesh (as each core focuses on points from a single partition at a time) and randomness for the DAG (as the interleaving of different cores' iterations is essentially random). With this schedule, we achieve a speedup of 3.13 on 4 cores.

4.3 Boykov-Kolmogorov maxflow

The Boykov-Kolmogorov algorithm [1] is a maxflow algorithm tuned for image segmentation problems and based on augmenting paths (we abbreviate the algorithm as "B-K maxflow"). It performs a breadth-first walk over the graph to find paths from the source to the sink in the residual graph. Once an augmenting path has been found and the flow is updated, the current search tree is updated to reflect the new flow, and then used as a starting point for computing the next search tree. The algorithm computes search trees starting from both the source and the sink.

B-K maxflow is naturally a worklist-style algorithm, as seen in Figure 8: each node at the frontier of a search tree is on the worklist. When a node is removed from the worklist, its edges are traversed to extend the search, and newly discovered nodes are added to the worklist. If an augmenting path is found, the capacities of all edges along the path are decremented appropriately. Nodes that are disconnected as a result of this augmentation are added back to the worklist.

```
1: worklist wl = /*initialize with SOURCE and SINK*/
2: for each Node n in worklist {
     //n in SourceTree or SinkTree
3:   if (n.inSourceTree()) {
4:     for each Node a in n.neighbors() {
5:       if (a.inSourceTree())
6:         continue; //already found
7:       else if (a.inSinkTree()) {
           //decrement capacity along path
8:         int cap = augment(n, a);
           //update total flow
9:         flow.inc(cap);
           //put disconnected nodes onto worklist
10:        processOrphans();
11:      } else {
12:        worklist.add(a);
13:        a.setParent(n); //put a into SourceTree
14:      }
15:    }
16:  } else { //n must be in the SinkTree
17:    ... //similar to code for when n in Source Tree
18:  }
19:}
```

Figure 8: Pseudocode for B-K maxflow

Opportunities for exploiting parallelism.

As in the other applications, the order in which elements are processed from the worklist is irrelevant to proper execution, although different orders will produce different search trees. Therefore, we can process nodes in the worklist concurrently, provided there are no conflicts. Conflicts can occur when concurrent traversals visit the same node (line 13), or when augmenting paths overlap (line 8).

Scheduling issues.

Unlike Delaunay triangulation and refinement, the iterations in B-K maxflow do relatively little work. Most perform a single step of a breadth-first search. Furthermore, augmenting paths in image segmentation problems tend to be short, so even iterations that perform augmentation are short. The cost of obtaining work from the worklist, especially in parallel, is a significant concern in this application, and thus the effect of scheduling on overhead and contention for shared structures is paramount.

Evaluation.

The sequential implementation is a Java port of the original sequential C code, which uses a queue for the worklist. The input data is a 1024x1024 image segmentation problem based on overlapping checkerboards. We evaluated five parallel schedules for this application:

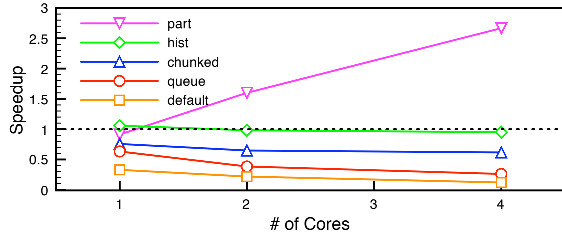
- *default* – the default schedule.
- *queue* – this is the same policy as *default*, except the labeling policy uses FIFO labeling. This schedule approximates the sequential schedule of execution since it will perform an approximate breadth-first traversal.
- *chunked* – this is the same policy as *queue*, except it uses the *random* clustering policy, and aims to create clusters of size 16. Dynamically generated iterations are assigned to new, unlabeled clusters. The ordering within a cluster is random.
- *hist* – this is the same policy as *chunked*, except the clustering function uses the *inherited* rule for dynamically generated iterations. Intra-cluster ordering is LIFO ordering. This is essentially the same schedule as *hist* in mesh refinement.
- *part* – This schedule is the same as *part* in mesh refinement, except the ordering policy uses *cluster-major* ordering and the clustering policy uses the *inherited* rule for new work. This clustering policy is key, because in this appli-

cation, data-centric clustering may not assign newly created work to the current cluster. As the input graph has a grid structure, the partitioning used for data-centric clustering is block-based.

Figure 9(a) gives the execution time on different numbers of cores, as well as the abort ratio on four cores. Figure 9(b) shows the speedup relative to *seq* for the parallel schedules. We see that *default* performs the worst, actually slowing down compared to sequential execution by a factor of 10 on four cores, while *part* performs the best, achieving a speedup of 2.67x over *seq*.

Schedule	1 core	2 cores	4 cores	
	Exec. Time	Exec. Time	Exec. Time	Abort Ratio
seq	384	—	—	—
default	1166	1759	3191	0.367%
queue	608	1000	1470	0.235%
chunked	508	593	623	0.109%
hist	363	391	404	0.071%
part	421	240	144	0.001%

(a) Execution time (in ms) and abort ratios



(b) Speedup vs. # of cores

Figure 9: Results for B-K maxflow

This application illustrates the effects of scheduling overhead on execution performance. The locality effect of newly created work manifests itself in better single-core performance for *queue* over *default*. However, both *default* and *queue* perform poorly on four cores, slowing down compared to the same schedule on one core. This is because accessing the worklist is a significant portion of each iteration, and if a schedule uses dynamic labeling, the worklist is global. These accesses are guarded by locks to ensure correct labeling of clusters, thus resulting in poor performance.

This effect can be mitigated by reducing the overhead of dynamic labeling. One approach is demonstrated by *chunked*: iterations are grouped into clusters, reducing the amount of labeling that must be done. This reduces execution time, but still has significant overhead during clustering: newly created work is assigned to new clusters, thus requiring synchronization to ensure correct cluster formation. The *hist* schedule keeps newly created work in the current cluster, eliminating the need to add and remove newly created work in the global worklist, which produces better results.

All of the previously discussed schedules rely on dynamic labeling to some extent, and this requires synchronization on a global worklist. By using static labeling, we no longer require a global worklist and this bottleneck is removed. We see that the *part* schedule, which uses static labeling and *inherited* cluster assignment for new work, is the best performing schedule.

This application demonstrates the need to carefully consider the overhead implicit in a scheduling decision, as it can have dramatic effects on application performance.

4.4 Preflow-Push

The preflow-push algorithm [4] is another approach to solving maxflow. Unlike augmenting-paths algorithms, which iteratively improve a valid flow until it is maximal, preflow-push algorithms

```
 1: Worklist wl = /* Nodes with excess flow */
 2: for each Node u in wl {
 3:    for each Edge e of Node u {
        /* push flow from u along edge e
           update capacity of e and excess in u
           flow == amount of flow pushed */
 4:       double flow = Push(u, e);
 5:       if(flow > 0)
 6:          worklist.add(e.head);
 7:    }
 8:    Relabel(u); // raise u's height if necessary
       // put u back in worklist if still active
 9:    if(u.excess > 0)
          worklist.add(u);
10: }
```

Figure 10: Pseudocode for preflow-push

work on an 'invalid' flow, called a *preflow*, where nodes can have excess inflow. Any nodes with excess are called *active*; by processing these nodes, the preflow is converted into a maximum flow.

The algorithm begins by pushing flow from the source to all of its neighbors, activating them. The goal is to eliminate the excess at these nodes by either directing it towards the sink or draining it back to the source. The basic idea is to maintain a height value at each node and push flow from higher nodes to lower nodes, all the way to the sink which is at height 0. Height values act as a lower bound on distance to the sink node, and are updated as the residual graph changes during the algorithm.

The basic preflow-push algorithm maintains a single worklist of active nodes that is processed until empty. Two operations, *push* and *relabel*, are permitted on each active node. A *push* operation moves excess flow from a node x at height h to a node y at height $h-1$; this activates node y and adds it to the worklist. A *relabel* operation lifts x up until it is high enough to push its excess to one of its neighbors. The pseudocode for the algorithm is given in Figure 10. Various heuristics have been proposed to improve performance, but a full examination of each variant is beyond the scope of this work, so here we focus on the basic algorithm.

Opportunities for exploiting parallelism.

Like the other algorithms in this paper, the preflow-push worklist can be processed in any order, making it suitable for parallelization. The push and relabel operations are both local, involving only an active node and its immediate neighborhood. Conflicts can occur if two cores attempt to manipulate the same node via either a push (line 4) or a relabel (line 8) operation. This algorithm is unique among our applications in that all successfully completed push and relabel operations remain valid even if followed immediately by a conflict. Thus, it is not necessary to undo any preflow-push work after an abort.

Scheduling issues.

Iterations in preflow-push are the shortest among our example applications; the push and relabel operations are cheap, and the regular nature of the graph means that nodes have few neighbors. The majority of the overhead arises from contention, either in interacting with the global worklist, or in keeping processors isolated from each other. Furthermore, unlike B-K maxflow, this algorithm is almost entirely based on newly generated work. Thus it is important to have a scheduling policy that can suitably cluster and label this new work to assign it to the appropriate processor.

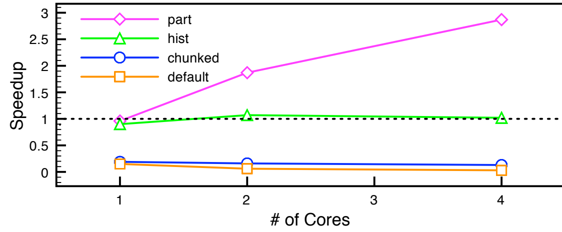
Evaluation.

The best performing sequential implementation uses the following schedule (inspired by our parallel experiments): when a unit of

work is removed from the main worklist, it is transferred to a secondary worklist. The algorithm processes the secondary worklist until exhausted, then returns to the main worklist to get the next unit of work. We compared this sequential schedule, *seq*, to the schedules *default*, *chunked*, *hist*, and *part* from B-K maxflow, using a 128x128 instance of the segmentation problem.

Schedule	1 core	2 cores	4 cores	
	Exec. Time	Exec. Time	Exec. Time	Abort Ratio
seq	4.93	——	——	——
default	32.09	83.62	144.59	12.23%
chunked	25.69	30.64	37.87	22.17%
hist	5.45	4.63	4.83	14.04%
part	5.12	2.64	1.72	<0.01%

(a) Execution time (seconds) and abort ratios



(b) Speedup vs. # of cores

Figure 11: Results for Preflow-Push

Preflow-push, even more than B-K maxflow, shows tremendous sensitivity to scheduling overheads. Computing a maxflow on the input data requires about 30 million iterations of the main loop. Unsurprisingly, *default* performs very poorly, and although *chunked* does show improvement due to larger iteration clusters, newly generated work is still handled too slowly to result in speedup.

By using a local worklist to speed up clustering of new work, we are able to at least match sequential performance, as shown in the *hist* schedule. However, the abort ratios here show the importance of intelligently clustering the iteration space. Thus, the *part* schedule, which statically clusters iteration space to reduce conflicts and lower scheduling overhead, achieves the best performance and results in a 2.86x speedup on 4 cores.

4.5 Agglomerative Clustering

Agglomerative clustering is a commonly used algorithm in data-mining [21] and other fields[3]. We use a version from a graphics application for handling a large numbers of light sources [23]. It builds a hierarchical representation (a binary tree or dendogram) from a set of points based on similarity or distance. The input is a set of points and a metric that assigns a size to any subset of points. The algorithm then progressively clusters pairs of elements (each element is subset of input points) that are the most similar (i.e., their union has the smallest size), until only one element is left (the root node of the binary tree). A kdtree is used to efficiently answer queries for the nearest neighbor of an element. When two elements are clustered, they are removed from the kdtree and a new element representing their union is added to the kdtree.

The algorithm described in [11] is a greedy algorithm using an ordered set, but under some mild conditions on the metric, there is an equivalent algorithm using an unordered set iterator, shown in Figure 12. Intuitively, we can cluster two elements together whenever we can prove that the ordered greedy algorithm would also cluster them eventually. If the metric is non-decreasing with respect to set membership and if two elements agree that they are each other's best match then it is safe to cluster them immediately.

[3] Our scheduling framework uses the notion of iteration clustering as described in Section 3. This is an instance of the general notion of data clustering from data-mining. To disambiguate these related notions, we will refer to *data clusters* and *iteration clusters* in this section.

```
 1: worklist = new Set(input_points);
 2: kdtree = new KDTree(input_points);
 3: for each Element a in worklist do {
 4:   b = kdtree.findNearest(a);
 5:   if (b == null) break; //stop if a is last element
 6:   c = kdtree.findNearest(b);
 7:   if (a == c) {
        //create new cluster e that contains a and b
 8:     Element e = cluster(a,b);
 9:     kdtree.remove(a);
10:     kdtree.remove(b);
11:     kdtree.add(e);
12:     worklist.remove(b);
13:     worklist.add(e);
14:   } else { //can't cluster a yet, try again later
15:     worklist.add(a);  //add back to worklist
16:   }
17: }
```

Figure 12: Psuedocode for agglomerative clustering

Opportunities for Parallelism.

The resulting tree is not affected by the order in which the worklist is processed, allowing elements to be processed in parallel. Conflicts arise when one thread modifies the kdtree (lines 9 to 11), and this changes the result of another thread's ongoing nearest neighbor computations (lines 4 and 6).

Scheduling issues.

Agglomerative clustering builds a binary tree from the bottom up so a node cannot be created before its children. Specifically, an element often cannot be clustered until after some other elements have been processed. A poor schedule can result in repeatedly attempting to cluster elements which cannot be clustered yet (line 15), leading to an explosion in the amount of work done.

Evaluation.

We evaluated three different schedules for this application using 200,000 initial points. The best sequential version uses the same locality optimizations as the *hist* schedule below, but without the conflict checking and synchronization overheads.

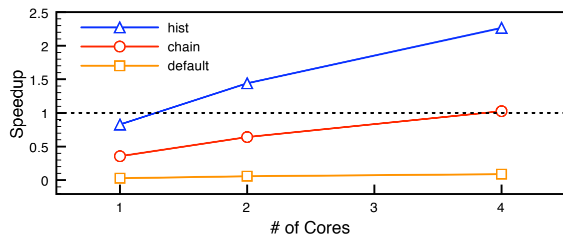
- *default* – The default schedule.
- *chain* – This schedule improves locality using a programmer-specified dynamic labeling policy. If an iteration does not successfully form a data cluster between *a* and *b*, the labeling policy assigns the iteration associated with *b* to the processor next, based on a scheduler hint inserted into the iteration at line 15.
- *hist* – This schedule is the same as *chain*, except when *a* and *b* are successfully combined in a data cluster (line 13). In this case, the schedule assigns the newly created iteration to the current iteration cluster (as in the *inherited* clustering policy), using another scheduler hint. This does not affect iterations generated by line 15, otherwise the loop would never terminate.

While *default* achieves good self-relative speedup using more cores, its performance is poor compared to the best sequential *seq*. Due to the scheduling issue discussed above, *default* executes more than 13 times as many iterations as *seq*.

The user-defined labeling function in *chain* results in a much

Schedule	1 core	2 cores	4 cores	
	Exec. Time	Exec. Time	Exec. Time	Abort Ratio
seq	4.28	—	—	—
default	153.41	73.93	47.93	0.27%
chain	12.02	6.68	4.17	0.22%
hist	5.17	2.97	1.89	0.07%

(a) Execution time (in seconds) and abort ratios



(b) Speedup vs. # of cores

Figure 13: Results for agglomerative clustering

more efficient schedule than *default*, performing about 10 times fewer iterations. It also exhibits better locality than *default*, and hence runs 12 times faster. Finally, *hist* exploits additional locality due to its clustering of newly created work after a successful clustering, leading to a real speedup of 2.3 on four cores.

We see that for this application, algorithmic effects dominate the performance, and the necessary schedule to mitigate these effects is complex and problem-specific. Our scheduling framework allows us to specify this kind of complex scheduling.

4.6 Summary of Results

Our experimental results clearly demonstrate it is beneficial to provide scheduling flexibility across applications — the default Galois scheduling policy tends to perform poorly. Furthermore, across different applications the optimal scheduling policy can differ. Table 1 summarizes the scheduling policies that we found to perform the best for each of our applications. The policies are presented as follows: clustering shows first the policy for initial work, then the policy for dynamically generated work; labeling specifies dynamic or static labeling, then the specific policy; and ordering shows first the cluster interleaving policy, then the intra-cluster ordering policy.

While every application we evaluated (other than the two maxflow problems) required a different set of scheduling policies to produce the best results, there are some common features which can inform a programmer's choice of schedules. When dealing with partitioned data structures, as in all applications other than agglomerative clustering, it is beneficial to perform *data-centric* clustering and labeling (though this is not the best schedule for mesh refinement, it approaches the optimal schedule in performance). When new work is created, *inherited* clustering should be chosen (a slight modification of this policy was necessary for agglomerative clustering to ensure termination). Cluster-major ordering is useful as it promotes locality. As these choices seem like natural starting points for designing a scheduling policy for an application, the Galois system provides this policy for programmers to use "out-of-the-box."

It is possible that even for a single application there is not a particular scheduling policy that performs the best. In irregular programs, behavior can be very input dependent. For lack of space, we have only evaluated each application on a single input set, and have not investigated this input-dependent variability in scheduling. However, for the general *types* of inputs we have considered for each application (*e.g.* image segmentation problems for B-K maxflow and preflow-push), we feel that there is likely only small amounts of variability in the optimal scheduling policy across inputs. We leave a full study, which would also consider other types of inputs which may have significantly different behavior, to future work.

5. RELATED WORK AND CONCLUSIONS

Related Work.

While data parallelism in *regular* applications has been studied extensively, it is only in the last decade or so that parallelism in *irregular* applications has been explored. A common hardware approach is Thread Level Speculation (TLS) [9, 16], which speculatively executes loop iterations in order. There have also been software-based approaches to parallelism. In [22], Vachharajani *et al* proposed a speculative variant of Decoupled Software Pipelining (DSWP) [14], which parallelizes loops by executing different strongly connected components of a loop on different processors – in essence transforming data parallelism into task parallelism.

However, both variants of loop parallelization are fundamentally tied to loop order; TLS requires loop ordering to perform rollback, while DSWP relies on loop ordering to detect strongly connected components, and hence extract parallelism. In essence, by taking a low-level view of data parallelism, they do not allow the abstractions required to perform scheduling as we propose here. Furthermore, all these systems detect speculative conflicts by examining the read and write sets of speculatively executed iterations. In contrast, we use commutativity properties of method invocations of abstract data types, which allows the use of semantic information in detecting conflicts. Our experience is that this reduces the number of reported conflicts dramatically.

In [15], Philbin *et al.* reordered loops in sequential applications to improve locality, using information about data accesses. In the realm of task-parallelism, Chen *et al* proposed scheduling concurrent execution in order to promote cache sharing on CMPs [2]. Both techniques, while different in their approaches, are similar in spirit to our use of data-centric clustering and labeling functions to promote locality.

In the past several years, there has been significant interest in Transactional Memory (TM) as a synchronization technique [12]. However, because TM is a concurrency construct for explicitly parallel programs, not a parallelization technique, the type of scheduling we discuss in this paper is meaningless; when using TM, a thread executes a transaction whenever it encounters one. There is a second aspect to scheduling which TM does consider: contention management (*i.e.* which speculative thread should be rolled back on a conflict). While there have been several policies proposed in the literature [19], in our experience we have not found it necessary to use anything other than the Galois default policy: the thread detecting the conflict rolls back.

Conclusions.

In this paper we presented a general framework for scheduling data-parallel computation, suited for both regular and irregular applications. We described how a schedule can be defined through three policies: clustering, labeling and ordering. We also showed how the object-oriented nature of the Galois system can be leveraged to easily implement our framework. Through an evaluation of the framework on several real-world applications, we demonstrated that different schedules can exhibit widely varying performance on a given application, and that there is no single, best-performing schedule across all applications. By extending the Ga-

Application	Scheduling Policy Clustering	Labeling	Ordering
Mesh refinement	random / inherited	dynamic / random	— / LIFO
Delaunay triangulation	data-centric / —	static / data-centric	cluster-major / random
B-K maxflow	data-centric / inherited	static / data-centric	cluster-major / LIFO
Preflow-push maxflow	data-centric / inherited	static / data-centric	cluster-major / LIFO
Agglomerative clustering	unit / custom	dynamic / custom	— / —

Table 1: Highest-performing scheduling policies for each application

lois system with our scheduling framework, programmers can explore the space of possible schedules, arriving at the particular schedule that best suits their application.

6. REFERENCES

[1] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *International Journal of Computer Vision (IJCV)*, 70(2):109–131, 2006.

[2] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM Press.

[3] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, 1993.

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

[5] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7(1):381–413, December 1992.

[6] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[7] B. W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–308, February 1970.

[8] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Jr. Guy L. Steele, and Mary E. Zosel. *The high performance Fortran handbook*. MIT Press, Cambridge, MA, USA, 1994.

[9] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[10] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News*, 36(1):233–243, 2008.

[11] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.

[12] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[13] OpenMP. `http://www.openmp.org`.

[14] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[15] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[16] Lawrence Rauchwerger and David A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[17] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 69–80, New York, NY, USA, 1989. ACM.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[19] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[20] Jonathan Richard 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.

[21] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, editors. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.

[22] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald Greenberg. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (SIGGRAPH)*, 24(3):1098–1107, July 2005.