Optimistic Parallelism Requires Abstractions [*]

Milind Kulkarni [†],
Keshav Pingali
Department of Computer Science,
University of Texas, Austin.
{milind, pingali}@cs.utexas.edu

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

Abstract

Irregular applications, which manipulate large, pointer-based data structures like graphs, are difficult to parallelize manually. Automatic tools and techniques such as restructuring compilers and runtime speculative execution have failed to uncover much parallelism in these applications, in spite of a lot of effort by the research community. These difficulties have even led some researchers to wonder if there is any coarse-grain parallelism worth exploiting in irregular applications.

In this paper, we describe two real-world irregular applications: a Delaunay mesh refinement application and a graphics application that performs agglomerative clustering. By studying the algorithms and data structures used in these applications, we show that there is substantial coarse-grain, data parallelism in these applications, but that this parallelism is very dependent on the input data and therefore cannot be uncovered by compiler analysis. In principle, optimistic techniques such as thread-level speculation can be used to uncover this parallelism, but we argue that current implementations cannot accomplish this because they do not use the proper abstractions for the data structures in these programs.

These insights have informed our design of the *Galois* system, an object-based optimistic parallelization system for irregular applications. There are three main aspects to Galois: (1) a small number of syntactic constructs for packaging optimistic parallelism as iteration over ordered and unordered sets, (2) assertions about methods in class libraries, and (3) a runtime scheme for detecting and recovering from potentially unsafe accesses to shared memory made by an optimistic computation.

We show that Delaunay mesh generation and agglomerative clustering can be parallelized in a straight-forward way using the Galois approach, and we present experimental measurements to show that this approach is practical. These results suggest that Galois is a practical approach to exploiting data parallelism in irregular programs.

1. Introduction

> A pessimist sees the difficulty in every opportunity;
> an optimist sees the opportunity in every difficulty.
>
> —*Sir Winston Churchill*

The advent of multicore processors has shifted the burden of improving program execution speed from chip manufacturers to software developers. A particularly challenging problem in this context is the parallelization of *irregular* applications that deal with complex, pointer-based data structures such as trees, queues and graphs. In this paper, we describe two such applications: a Delaunay mesh refinement code [8] and a graphics application [41] that performs agglomerative clustering [26].

In principle, it is possible to use a thread library (e.g., pthreads) or a combination of compiler directives and libraries (e.g., OpenMP [25]) to write threaded code for multicore processors, but it is well known that writing threaded code can be very tricky because of the complexities of synchronization, data races, memory consistency, etc. Tim Sweeney, who designed the multi-threaded Unreal 3 game engine, estimates that writing multi-threading code tripled software costs at Epic Games (quoted in [9]).

Another possibility is to use compiler analyses such as points-to and shape analysis [5, 32] to parallelize sequential irregular programs. Unfortunately, static analyses fail to uncover the parallelism in such applications because the parallel schedule is very data-dependent and cannot be computed at compile-time, as we argue in Section 3.

Optimistic parallelization [17] is a promising idea, but current implementations of optimistic parallelization such as thread-level speculation (TLS) [37, 39, 45] cannot exploit the parallelism in these applications, as we discuss in Section 3.

In this paper, we describe the *Galois* approach to parallelizing irregular applications. This approach is informed by the following beliefs.

- Optimistic parallelization is the only plausible approach to parallelizing many, if not most, irregular applications.
- For effective optimistic parallelization, it is crucial to exploit the abstractions provided by object-oriented languages (in particular, the distinction between an abstract data type and its implementation).
- Concurrency should be packaged, when possible, within syntactic constructs that make it easy for the programmer to express what might be done in parallel, and for the compiler and runtime system to determine what should be done in parallel. The syntactic constructs used in Galois are very natural and can be added easily to any object-oriented programming language like Java. They are related to set iterators in SETL [19].
- Concurrent access to mutable shared objects by multiple threads is fundamental, and cannot be added to the system as an afterthought as is done in current approaches to optimistic paral-

[*] This work is supported in part by NSF grants 0615240, 0541193, 0509307, 0509324, 0426787 and 0406380, as well as grants from the IBM and Intel Corportations.

[†] Milind Kulkarni is supported by the DOE HPCS Fellowship.

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

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006…$5.00.

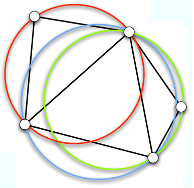


Figure 1. A Delaunay mesh. Note that the circumcircle for each of the triangles does not contain other points in the mesh.

lelization. However, discipline needs to be imposed on concurrent accesses to shared objects to ensure correct execution.

We have implemented the Galois approach in C++ on two shared-memory platforms, and we have used this implementation to write a number of complex applications including Delaunay mesh refinement, agglomerative clustering, an image segmentation code that uses graph cuts [41], and an approximate SAT solver called WalkSAT [34].

This paper describes the Galois approach and its implementation, and presents performance results for some of these applications. It is organized as follows. In Section 2, we present Delaunay mesh refinement and agglomerative clustering, and describe opportunities for exploiting parallelism in these codes. In Section 3, we give an overview of existing parallelization techniques and argue that they cannot exploit the parallelism in these applications. In Section 4, we discuss the Galois programming model and run-time system. In Section 5, we evaluate the performance of our system on the two applications. Finally, in Section 6, we discuss conclusions and ongoing work.

2. Two Irregular Applications

To understand the nature of the parallelism in irregular programs, it is useless to study the execution traces of irregular programs, as most studies in this area do; instead it is necessary to recall Niklaus Wirth's aphorism *program = algorithm + data structure* [43], and examine the relevant algorithms and data structures. In this section, we describe two irregular applications: Delaunay mesh refinement [8, 31], and agglomerative clustering [26] as used within a graphics application [41]. These applications perform refinement and coarsening respectively, which are arguably the two most common operations for bulk-modification of irregular data structures. For each application, we describe the algorithm and key data structures, and describe opportunities for exploiting parallelism.

2.1 Delaunay Mesh Refinement

Mesh generation is an important problem with applications in many areas such as the numerical solution of partial differential equations and graphics. The goal of mesh generation is to represent a surface or a volume as a tessellation composed of simple shapes like triangles, tetrahedra, etc.

Although many types of meshes are used in practice, *Delaunay meshes* are particularly important since they have a number of desirable mathematical properties [8]. The Delaunay triangulation for a set of points in the plane is the triangulation such that no point is inside the circumcircle of any triangle (this property is called the *empty circle property*). An example of such a mesh is shown in Figure 1.

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose quality constraints governing the shape and size of the triangles. For a given Delaunay mesh, this is accomplished by *iterative mesh refinement*, which successively fixes "bad" triangles (triangles that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating. Figure 2 illustrates this process; the shaded triangle in Figure 2(a) is assumed to be bad. To fix this bad triangle, a new point is added at the circumcenter of this triangle. Adding this point may invalidate the empty circle property for some neighboring triangles, so all affected triangles are determined (this region is called the *cavity* of the bad triangle), and the cavity is re-triangulated, as shown in Figure 2(c) (in this figure, all triangles lie in the cavity of the shaded bad triangle). Re-triangulating a cavity may generate new bad triangles but it can be shown that this iterative refinement process will ultimately terminate and produce a guaranteed-quality mesh. Different orders of processing bad elements lead to different meshes, although all such meshes satisfy the quality constraints [8].

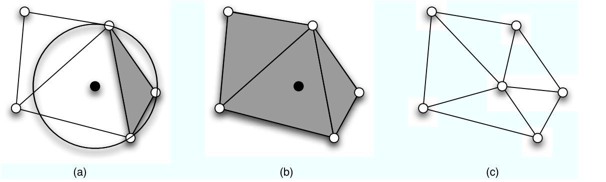


Figure 2. Fixing a bad element.

```
Mesh m = /* read in initial mesh */
WorkList wl;
wl.add(mesh.badTriangles());
while (wl.size() != 0) {
   Element e = wl.get(); //get bad triangle
   if (e no longer in mesh) continue;
   Cavity c = new Cavity(e);
   c.expand();
   c.retriangulate();
   mesh.update(c);
   wl.add(c.badTriangles());
}
```

Figure 3. Pseudocode of the mesh refinement algorithm

Figure 3 shows the pseudocode for mesh refinement. The input to this program is a Delaunay mesh in which some triangles may be bad, and the output is a refined mesh in which all triangles satisfy the quality constraints. There are two key data structures used in this algorithm. One is a worklist containing the bad triangles in the mesh. The other is a graph representing the mesh structure; each triangle in the mesh is represented as one node, and edges in the graph represent triangle adjacencies in the mesh.

Opportunities for Exploiting Parallelism. The natural unit of work for parallel execution is the processing of a bad triangle. Our measurements show that on the average, each unit of work takes about a million instructions of which about 10,000 are floating-point operations. Because a cavity is typically a small neighborhood of a bad triangle, two bad triangles that are far apart on the mesh may have cavities that do not overlap. Furthermore, the entire refinement process (expansion, retriangulation and graph updating) for the two triangles is completely independent; thus, the two triangles can be processed in parallel. This approach obviously extends to more than two triangles. If however the cavities of two triangles overlap, the triangles can be processed in either order but only one of them can be processed at a time. Whether or not two bad triangles have overlapping cavities depends entirely on the structure of the mesh, which changes throughout the execution of the algorithm.

How much parallelism is there in Delaunay mesh generation? The answer obviously depends on the mesh and on the order in which bad triangles are processed, and may be different at different points during the execution of the algorithm. One study by Antonopoulos *et al.* [2] on a mesh of one million triangles found that there were more than 256 cavities that could be expanded in parallel until almost the end of execution.

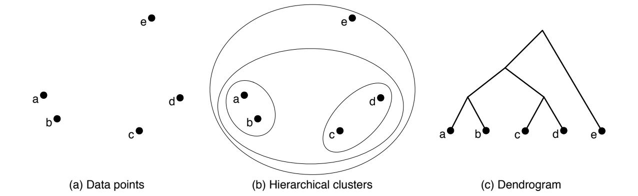


Figure 4. Agglomerative clustering

2.2 Agglomerative Clustering

The second problem is *agglomerative clustering*, a well-known data-mining algorithm [26]. This algorithm is used in graphics applications for handling large numbers of light sources [41].

The input to the clustering algorithm is (1) a data-set, and (2) a measure of the "distance" between items in the data-set. Intuitively, this measure is an estimate of similarity — the larger the distance between two data items, the less similar they are believed to be. The goal of clustering is to construct a binary tree called a dendrogram whose hierarchical structure exposes the similarity between items in the data-set. Figure 4(a) shows a data-set containing points in the plane, for which the measure of distance between data points is the usual Euclidean distance. The dendrogram for this data set is shown in Figures 4(b,c).

Agglomerative clustering can be performed by an iterative algorithm: at each step, the two closest points in the data-set are clustered together and replaced in the data-set by a single new point that represents the new cluster. The location of this new point may be determined heuristically [26]. The algorithm terminates when there is only one point left in the data-set.

Pseudocode for the algorithm is shown in Figure 5. The central data structure is a priority queue whose entries are ordered pairs of points `<x,y>`, such that `y` is the nearest neighbor of `x` (we call this `nearest(x)`). In each iteration of the while loop, the algorithm dequeues the top element of the priority queue to find a pair of points `<p,n>` that are closer to each other than any other pair of points, and clusters them. These two points are then replaced by a new point that represents this cluster. The nearest neighbor of this new point is determined, and the pair is entered into the priority queue. If there is only one point left, its nearest neighbor is the point at infinity.

To find the nearest neighbor of a point, we can scan the entire data-set at each step, but this is too inefficient. A better approach is to sort the points by location, and search within this sorted set to find nearest neighbors. If the points were all in a line, we could use a binary search tree. Since the points are in higher dimensions, a multi-dimensional analog called a *kd-tree* is used [3]. The kd-tree is built at the start of the algorithm, and it is updated by removing the points that are clustered, and then adding the new point representing the cluster, as shown in Figure 5.

Opportunities for Exploiting Parallelism. Since each iteration clusters the two closest points in the current data-set, it may seem that the algorithm is inherently sequential. In particular, an item `<x,nearest(x)>` inserted into the priority queue by iteration `i` at line 17 may be the same item that is dequeued by iteration `(i+1)` in line 5; this will happen if the points in the new pair are closer together than any other pair of points in the current data-set. On the other hand, if we consider the data-set in Figure 4(a), we see that points `a` and `b`, and points `c` and `d` can be clustered concurrently since neither cluster affects the other. Intuitively, if the dendrogram is a long and skinny tree, there may be few independent iterations, whereas if the dendrogram is a bushy tree, there is parallelism that can be exploited since the tree can be constructed bottom-up in parallel. As in the case of Delaunay mesh refinement, the parallelism is very data-dependent. In experiments on graphics scenes with 20,000 lights, we have found that on average about 100 clusters can be constructed concurrently; thus, there is substantial parallelism that can be exploited. For this application, each iteration of the while-loop in Figure 5 performs about 100,000 instructions of which roughly 4000 are floating-point operations.

```
1:  kdTree := new KDTree(points)
2:  pq := new PriorityQueue()
3:  foreach p in points {pq.add(<p,kdTree.nearest(p)>)}
4:  while(pq.size() != 0) do {
5:    Pair <p,n> := pq.get();//return closest pair
6:    if (p.isAlreadyClustered()) continue;
7:    if (n.isAlreadyClustered()) {
8:       pq.add(<p, kdTree.nearest(p)>);
9:       continue;
10:   }
11:   Cluster c := new Cluster(p,n);
12:   dendrogram.add(c);
13:   kdTree.remove(p);
14:   kdTree.remove(n);
15:   kdTree.add(c);
16:   Point m := kdTree.nearest(c);
17:   if (m != ptAtInfinity) pq.add(<c,m>);
18: }
```

Figure 5. Pseudocode for agglomerative clustering

3. Limitations of Current Approaches

Current approaches for parallelizing irregular applications can be divided into static, semi-static, and dynamic approaches.

Static Approaches. One approach to parallelization is to use a compiler to analyze and transform sequential programs into parallel ones, using techniques like *points-to analysis* [5] and *shape analysis* [32]. The weakness of this approach is that the parallel schedule produced by the compiler must be valid for all inputs to the program. As we have seen, parallelism in irregular applications can be very data-dependent, so compile-time parallelization techniques will serialize the entire execution. This conclusion holds even if dependence analysis is replaced with more sophisticated analysis techniques like commutativity analysis [10].

A Semi-static Approach. In the *inspector-executor* approach of Saltz *et al* [27], the computation is split into two phases, an *inspector* phase that determines dependencies between units of work, and an *executor* phase that uses the schedule to perform the computation in parallel. This approach is not useful for our applications since the data-sets, and therefore the dependences, change as the codes execute.

Dynamic Approaches. In dynamic approaches, parallelization is performed at runtime, and is known as *speculative* or *optimistic* parallelization. The program is executed in parallel assuming that dependences are not violated, but the system software or hardware detects dependence violations and takes appropriate corrective action such as killing off the offending portions of the program and re-executing them sequentially. If no dependence violations are detected by the end of the speculative computation, the results of the speculative computation are *committed* and become available to other computations.

Fine-grain speculative parallelization for exploiting instruction-level parallelism was introduced around 1970; for example, Tomasulo's IBM 360/91 fetched instructions speculatively from both sides of a branch before the branch target was resolved [38]. Speculative *execution* of instructions past branches was studied in the abstract by Foster and Riseman in 1972 [7], and was made practical by Josh Fisher when he introduced the idea of using branch probabilities to guide speculation [11]. Branch speculation can expose instruction-level (fine-grain) parallelism in programs but not the data-dependent coarse-grain parallelism in applications like Delaunay mesh refinement.

One of the earliest implementations of *coarse-grain* optimistic parallel execution was in Jefferson's 1985 Time Warp system for distributed discrete-event simulation [17]. In 1999, Rauchwerger and Padua described the LRPD test for supporting speculative execution of FORTRAN DO-loops in which array subscripts were too complex to be disambiguated by dependence analysis [30]. This approach can be extended to while-loops if an upper bound on the number of loop iterations can be determined before the loop begins execution [29]. More recent work has provided hardware support for this kind of coarse-grain loop-level speculation, now known as thread-level speculation (TLS) [37, 39, 45].

However, there are fundamental reasons why current TLS implementations cannot exploit the parallelism in our applications. One problem is that many of these applications, such as Delaunay mesh refinement, have unbounded while-loops, which are not supported by most current TLS implementations since they target FORTRAN-style DO-loops with fixed loop bounds. A more fundamental problem arises from the fact that current TLS implementations track dependences by monitoring the reads and writes made by loop iterations to memory locations. For example, if iteration *i+1* writes to a location before it is read by iteration *i*, a dependence violation is reported, and iteration *i+1* must be rolled back.

For irregular applications that manipulate pointer-based data structures, this is too strict and the program will perform poorly because of frequent roll-backs. To understand this, consider the worklist in Delaunay mesh generation. Regardless of how the worklist is implemented, there must be a memory location (call this location *head*) that points to a cell containing the next bad triangle to be handed out. The first iteration of the while loop removes a bad triangle from the worklist, so it reads and writes to *head*, but the result of this write is not committed until that iteration terminates successfully. A thread that attempts to start the second iteration concurrently with the execution of the first iteration will also attempt to read and write *head*, and since this happens before the updates from the first iteration have been committed, a dependence conflict will be reported (the precise point at which a dependence conflict will be reported depends on the TLS implementation). While this particular problem might be circumvented by inventing some *ad hoc* mechanism, it is unlikely that there is any such work-around for the far more complex priority queue manipulations in agglomerative clustering. The manipulations of the graph and kd-tree in these applications may also create such conflicts.

This is a fundamental problem: *for many irregular applications, tracking dependences by monitoring reads and writes to memory locations is correct but will result in poor performance*.

Finally, Herlihy and Moss have proposed to simplify shared-memory programming by eliminating lock-based synchronization constructs in favor of *transactions* [15]. There is growing interest in supporting transactions efficiently with software and hardware implementations of *transactional memory* [1, 12, 13, 21, 35]. Most of this work is concerned with optimistic *synchronization* and not optimistic *parallelization*; that is, their starting point is a program that has already been parallelized (for example, the SPLASH benchmarks [12] or the Linux kernel [28]), and the goal is find an efficient way to synchronize parallel threads. In contrast, our goal is to find the right abstractions for expressing coarse-grain parallelism in irregular applications, and to support these abstractions efficiently; synchronization is one part of a bigger problem we are addressing in this paper. Furthermore, most implementations of transactional memory track reads and writes to memory locations, so they suffer from the same problems as current TLS implementations. Open nested transactions [22] have been proposed recently as a solution to this problem, and they are discussed in more detail in Section 4.

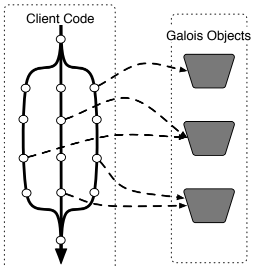


Figure 6. High-level view of Galois execution model

4. The Galois Approach

Perhaps the most important lesson from the past twenty-five years of parallel programming is that the complexity of parallel programming should be hidden from programmers as far as possible. For example, it is likely that more SQL programs are executed in parallel than programs in any other language. However, most SQL programmers do not write explicitly parallel code; instead they obtain parallelism by invoking parallel library implementations of joins and other relational operations. A "layered" approach of this sort is also used in dense linear algebra, another domain that has successfully mastered parallelism.

In this spirit, programs in the Galois approach consist of (i) a set of library classes and (ii) the top-level *client code* that creates and manipulates objects of these classes. For example, in Delaunay mesh refinement, the relevant objects are the mesh and worklist, and the client code implements the Delaunay mesh refinement algorithm discussed in Section 2. This client code is executed concurrently by some number of threads, but as we will see, it is not explicitly parallel and makes no mention of threads. Figure 6 is a pictorial view of this execution model.

There are three main aspects to the Galois approach: (1) two syntactic constructs called *optimistic iterators* for packaging optimistic parallelism as iteration over sets (Section 4.1), (2) assertions about methods in class libraries (Section 4.2), and (3) a runtime scheme for detecting and recovering from potentially unsafe accesses to shared objects made by an optimistic computation (Section 4.3).

4.1 Optimistic iterators

As mentioned above, the client code is not explicitly parallel; instead parallelism is packaged into two constructs that we call *optimistic iterators*. In the compiler literature, it is standard to distinguish between *do-all* loops and *do-across* loops [20]. The iterations of a do-all loop can be executed in any order because the compiler or the programmer asserts that there are no dependences between iterations. In contrast, a do-across loop is one in which there may be dependences between iterations, so proper sequencing of iterations is essential. We introduce two analogous constructs for packaging optimistic parallelism.

- **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 is an iterator over a partially-ordered set (Poset) S. It asserts that in a serial execution of the loop, the iterations must be performed in the order dictated by the ordering of

```
Mesh m = /* read in initial mesh */
Set wl;
wl.add(mesh.badTriangles());
for each e in wl do {
   if (e no longer in mesh) continue;
   Cavity c = new Cavity(e);
   c.expand();
   c.retriangulate();
   m.update(c);
   wl.add(c.badTriangles());
}
```

Figure 7. Delaunay mesh refinement using set iterator

elements in the Poset S. There may be dependences between iterations, and as in the case of the set iterator, elements may be added to S during execution.

The set iterator is a special case of the ordered-set iterator but it can be implemented more efficiently, as we see later in this section.

Figure 7 shows the client code for Delaunay mesh generation. Instead of a work list, this code uses a set and a set iterator. The Galois version is not only simpler but also makes evident the fact that the bad triangles can be processed in any order; this fact is absent from the more conventional code of Figure 3 since it implements a particular processing order. For lack of space, we do not show the Galois version of agglomerative clustering, but it uses the ordered-set iterator in the obvious way.

4.1.1 Concurrent Execution Model

Although the semantics of Galois iterators can be specified without appealing to a parallel execution model, these iterators provide hints from the programmer to the Galois runtime system that it may be profitable to execute the iterations in parallel; of course any parallel execution must be faithful to the sequential semantics.

The Galois concurrent execution model is the following. A master thread begins the execution of the program; it also executes the code outside iterators. 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; for now, we assume that this assignment is done dynamically to ensure load-balancing. All threads are synchronized using barrier synchronization at the end of the iterator.

In our applications, we have not found it necessary to use nested iterators. There is no fundamental problem in supporting nested parallelism, but our current implementation does not support it; if a thread encounters an inner iterator, it executes the entire inner iterator sequentially.

Given this execution model, the main technical problem is to ensure that the parallel execution respects the sequential semantics of the iterators. This is a non-trivial problem because each iteration may read and write to the objects in shared memory, and we must ensure that these reads and writes are properly coordinated. Section 4.2 describes the information that must be specified by the Galois class writer to enable this. Section 4.3 describes how the Galois runtime system uses this information to ensure that the sequential semantics of iterators are respected.

4.2 Writing Galois Classes

To ensure that the sequential semantics of iterators are respected, there are two problems that must be solved, which we explain with reference to Figure 8. This figure shows set objects with methods `add(x)`, `remove(x)`, `get()` and `contains?(x)` that have the usual semantics[1].

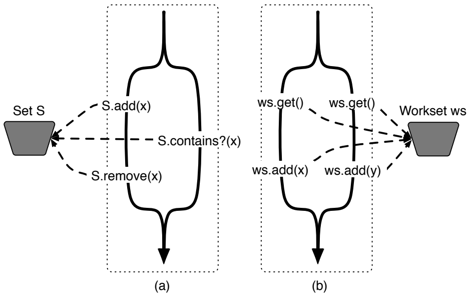


Figure 8. Interleaving method invocations from different iterations

The first problem is the usual one of concurrency control (also known in the database literature as ensuring *consistency*). If a method invocation from one iteration is performed concurrently with an invocation from another iteration, we must ensure that the two invocations do not step on each other. One solution is to use a lock on object S; if this inhibits concurrency, we can use fine-grain locks within object S. These locks are acquired before the method is invoked and released when the method completes.

However, this is not enough to ensure that the sequential semantics of the iterators are respected. Consider Figure 8(a). If S does not contain x before the iterations start, notice that in any sequential execution of the iterations, the method invocation `contains?(x)` will return false. However, for one possible interleaving of operations — `add(x)`, `contains?(x)`, `remove(x)` — the invocation `contains?(x)` will return true, which is incorrect. This is the problem of ensuring *isolation* of the iterations.

One solution for both problems is for an iteration to release its locks only at the end of the iteration: the well-known two-phase locking algorithm used in databases is an optimized version of this simple idea. Transactional memory implementations accomplish the same goal by tracking the read and write sets of each iteration instead of locking them.

While this solves the problem in Figure 8(a), it is not adequate for our applications. The program in Figure 8(b) is motivated by Delaunay mesh generation: each iteration gets a bad triangle at the beginning of the iteration, and may add some bad triangles to the work-set at the end. Regardless of how the set object is implemented, there must be a location (call it `head`) that points to a cell containing the next triangle to be handed out. The first iteration to get work will read and write location `head`, and it will lock it for the duration of the iteration, preventing any other iterations from getting work. Most current implementations of transactional memory will suffer from the same problem since the `head` location will be in the read and write sets of the first iteration for the duration of that iteration.

The crux of the problem is that the *abstract* set operations have useful semantics that are not available to an implementation that works directly on the *representation* of the set and tracks reads and writes to individual memory locations. The problem therefore is to understand the semantics of set operations that must be exploited to permit parallel execution in our irregular applications, and to specify these semantics in some concise way.

4.2.1 Semantic Commutativity

The solution we have adopted exploits the commutativity of method invocations. Intuitively, it is obvious that the method invocations to a given object from two iterations can be interleaved without losing isolation provided that these method invocations commute, since this ensures that the final result is consistent with some serial order of iteration execution. In Figure 8(a), the invocation $contains?(x)$ does not commute with the operations from the other thread, so the invocations from the two iterations must

[1] The method `remove(x)` removes a specific element from the set while `get()` returns an arbitrary element from the set, removing it from the set.

not be interleaved. In Figure 8(b), (1) *get* operations commute with each other, and (2) a *get* operation commutes with an *add* operation provided that the operand of *add* is not the element returned by *get*. This allows multiple threads to pull work from the work-set while ensuring that sequential semantics of iterators are respected.

It is important to note that what is relevant for our purpose is commutativity in the semantic sense. The internal state of the object may actually be different for different orders of method invocations even if these invocations commute in the semantic sense. For example, if the set is implemented using a linked list and two elements are added to this set, the concrete state of the linked list will depend in general on the order in which these elements were added to the list. However, what is relevant for parallelization is that the state of the set abstract data type, which is being implemented by the linked list, is the same for both orders. In other words, we are not concerned with concrete commutativity (that is, commutativity with respect to the implementation type of the class), but with semantic commutativity (that is, commutativity with respect to the abstract data type of the class). We also note that commutativity of method invocations may depend on the arguments of those invocations. For example, an *add* and a *remove* commute only if their arguments are different.

4.2.2 Inverse Methods

Because iterations are executed in parallel, it is possible for commutativity conflicts to prevent an iteration from completing. Once a conflict is detected, some recovery mechanism must be invoked to allow execution of the program to continue despite the conflict. Because our execution model uses the paradigm of optimistic parallelism, our recovery mechanism rolls back the execution of the conflicting iteration. To avoid livelock, the lower priority iteration is rolled back in the case of the ordered-set iterator.

To permit this, every method of a shared object that may modify the state of that object must have an associated *inverse* method that undoes the side-effects of that method invocation. For example, for a set, the inverse of *add(x)* is *remove(x)*, and the inverse of *remove(x)* is *add(x)*. As in the case of commutativity, what is relevant for our purpose is an inverse in the *semantic* sense; invoking a method and its inverse in succession may not restore the concrete data structure to what it was.

Note that when an iteration rolls back, all of the methods which it invokes during roll-back must succeed. Thus, we must never encounter conflicts when invoking inverse methods. When the Galois system checks commutativity, it also checks commutativity with the associated inverse method.

4.2.3 Putting it All Together

Since we are interested in *semantic* commutativity and undo, it is necessary for the class designer to specify this information. Figure 9 illustrates how this information is specified in Galois for a class that implements sets. The interface specifies two versions of each method: the *internal methods* on the object, and the *interface methods*, called from within iterators, that perform the commutativity checks, maintain the undo information and trigger roll backs when commutativity conflicts are detected.

The specification for an interface method consists of three main sections (with pseudo-code representing these in the figure):

- *calls*: This section ties the interface method to the internal method(s) it invokes.
- *commutes*: This section specifies which other interface methods the current method commutes with, and under which conditions. For example, `remove(x)` commutes with `add(y)`, as long as they elements are different.
- *inverse*: This section specifies the inverse of the current method.

```
class Set {
   // interface methods
   void add(Element x);
      [calls] _add(x) : void
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {y != x}
         - get() : y {y != x} //get call that returns y
      [inverse] _remove(x)
   void remove(Element x);
      [calls] _remove(x) : void
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {y != x}
         - get() : y {y != x}
      [inverse] _add(x)
   bool contains(Element x);
      [calls] _contains(x) : bool b
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - get() : y {y != x}
         - contains(*) //any call to contains
   Element get();
      [calls] _get() : Element x
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {y != x}
         - get() : y {y != x}
      [inverse] _add(x)

   //internal methods
   void _add(Element x);
   void _remove(Element x);
   bool _contains(Element x);
   Element _get();
}
```

Figure 9. Example Galois class for a Set

The description of the Galois system in this section implicitly assumed that all calls to parallel objects are made from client code. However, to facilitate composition, we also allow parallel objects to invoke methods on other objects. This is handled through a simple flattening approach. The iteration object is passed to the "child" invocation and hence all operations done in the child invocation are appended to the undo log of the iteration. Similarly, the child invocation functions as an extension of the original method when detecting commutativity conflicts. No changes need to be made to the Galois run-time to support this form of composition.

The class implementor must also ensure that each internal method invocation is atomic to ensure consistency. This can be done using any technique desired, including locks or transactional memory. Recall that whatever locks are acquired during method invocation (or memory locations placed in read/write sets during transactional execution) are released as soon as the method completes, rather than being held throughout the execution of the iteration, since we rely on commutativity information to guarantee isolation. In our current implementation, the internal methods are made atomic through the use of locks.

4.2.4 A small example

Iteration A	**Iteration B**	**Iteration C**

`{`
`   ...`
`   a.accumulate(5)`
`   ...`
`}`
	`{`
	`   ...`
	`   a.accumulate(7)`
	`   ...`
	`}`
		`{`
		`   ...`
		`   a.read()`
		`   ...`
		`}`

Figure 10. Example accumulator code

Consider a program written using a single shared object, an integer accumulator. The object supports two operations: *accumulate* and *read*, with the obvious semantics. It is clear that *accumulate*s commute with other *accumulate*s, and *read*s commute with other *read*s, but that *accumulate* does not commute with *read*. The methods are made atomic with a single lock which is acquired at the beginning of the method and released at the end.

There are three iterations executing concurrently, as seen in Figure 10. The progress of the execution is as follows:

- Iteration A calls *accumulate*, acquiring the lock, updating the accumulator and then releasing the lock and continuing.
- Iteration B calls *accumulate*. Because *accumulate*s commute, B can successfully make the call, acquiring the lock, updating the accumulator and releasing it. Note that A has already released the lock on the accumulator, thus allowing B to make forward progress without blocking on the accumulator's lock.
- When iteration C attempts to execute *read*, it sees that it cannot, as *read* does not commute with the already executed *accumulate*s. Thus, C must roll back and try again. Note that this is not enforced by the lock on the accumulator, but instead by the commutativity conditions on the accumulator.
- When iterations A and B commit, C can then successfully call *read* and continue execution.

In [40], von Praun *et al* discuss the use of ordered transactions in parallelizing FORTRAN-style DO-loops, and they give special treatment to reductions in such loops to avoid spurious conflicts. Reductions do not require any special treatment in the Galois approach since the programmer could just use an object like the accumulator to implement reduction.

4.3 Runtime System

The Galois runtime system has two components: (1) a global structure called the *commit pool* that is responsible for creating, aborting, and committing iterations, and (2) per-object structures called *conflict logs* which detect when commutativity conditions are violated.

At a high level, the runtime systems works as follows. The commit pool maintains an *iteration record*, shown in Figure 11, for each ongoing iteration in the system. The status of an iteration can be RUNNING, RTC (ready-to-commit) or ABORTED. Threads go to the commit pool to obtain an iteration. The commit pool creates a new iteration record, obtains the next element from the iterator, assigns a priority to the iteration record based on the priority of the element (for a set iterator, all elements have the same priority), and sets the status field of the iteration record to RUNNING. When an iteration invokes a method of a shared object, (i) the conflict log of that object and the local_log of the iteration record are updated, as described in more detail below, and (ii) a callback to the associated inverse method is pushed onto the undo log of the iteration record. If a commutativity conflict is detected, the commit pool arbitrates between the conflicting iterations, and aborts iterations to permit the highest priority iteration to continue execution. Callbacks in the undo logs of aborted iterations are executed to undo their effects on shared objects. Once a thread has completed an iteration, the status field of that iteration is changed to RTC, and the thread is allowed to begin a new iteration. When the completed iteration has the highest priority in the system, it is allowed to commit. It can be seen that the role of the commit pool is similar to that of a reorder buffer in out-of-order processors [14].

4.3.1 Conflict Logs

The *conflict log* is the mechanism for detecting commutativity conflicts. There is one conflict log associated with each shared object. A simple implementation for the conflict log of an object is a list containing the method signatures (including the values of the input and output parameters) of all invocations on that object made by currently executing iterations (called "outstanding invocations"). When iteration i attempts to call a method m_1 on an object, the method signature is compared against all the outstanding invocations in the conflict log. If one of the entries in the log does not commute with m_1, then a commutativity conflict is detected, and an arbitration process is begun to determine which iterations should be aborted, as described below. If m_1 commutes with all the entries in the log, the signature of m_1 is appended to the log. When i either aborts or commits, all the entries in the conflict log inserted by i are removed from the conflict log.

```
IterationRecord {
   Status status;
   Priority p;
   UndoLog ul;
   list<LocalConflictLog> local_log;
   Lock l;
}
```

Figure 11. Iteration record maintained by runtime system

This model for conflict logs, while simple, is not efficient since it requires a full scan of the conflict log whenever an iteration calls a method on the associated object. In our actual implementation, conflict logs consist of separate *conflict sets* for each method in the class. Now when i calls m_1, only the conflict sets for methods which m_1 may conflict with are checked; the rest are ignored.

There are two optimizations that we have implemented for conflict logs.

First, each iteration caches its own portion of the conflict logs in a private log called its local_log. This local log stores a record of all the methods the iteration has successfully invoked on the object. When an iteration makes a call, it first checks its local log. If this local log indicates that the invocation will succeed (either because that same method has been called before or other methods, whose commutativity implies that the current method also commutes, have been called before[2]), the iteration does not need to check the object's conflict log.

A second optimization is that not all objects have conflict logs associated with them. For example, the triangles contained in the mesh do not; their information is managed by the conflict log in the mesh. If this optimization is used, care must be taken that modifications to the triangle are only made through the mesh interface. In general, program analysis is required to ensure that this optimization is safe.

4.3.2 Commit Pool

When an iteration attempts to commit, the commit pool checks two things: (i) that the iteration is at the head of the commit queue, and (ii) that the priority of the iteration is higher than all the elements left in the set/poSet being iterated over[3]. If both conditions are met, the iteration can successfully commit. If the conditions are not met, the iteration must wait until it has the highest priority in the system; its status is set to RTC, and the thread is allowed to begin another iteration.

When an iteration successfully commits, the thread that was running that iteration also checks the commit queue to see if more iterations in the RTC state can be committed. If so, it commits those iterations before beginning the execution of a new iteration. When an iteration has to be aborted, the status of its record is changed

[2] For example, if an iteration has already successfully invoked add(x), then contains(x) will clearly commute with method invocations made by other ongoing iterations.

[3] This is to guard against a situation where an earlier committed iteration adds a new element with high priority to the collection which has not yet been consumed by the iterator

to ABORTED, but the commit pool takes no further action. Such iteration objects are lazily removed from the commit queue when they reach the head.

Conflict arbitration The other responsibility of the commit pool is to arbitrate conflicts between iterations. When iterating over an unordered set, the choice of which iteration to roll back in the event of a conflict is irrelevant. For simplicity, we always choose the iteration which detected the conflict. However, when iterating over an ordered set, the lower priority iteration must be rolled back while the higher priority iteration must continue. Without doing so, there exists the possibility of deadlock.

Thus, when iteration i_1 calls a method on a shared object and a conflict is detected with iteration i_2, the commit pool arbitrates based on the priorities of the two iterations. If i_1 has lower priority, it simply performs the standard rollback operations. The thread which was executing i_1 then begins a new iteration.

This situation is complicated when i_2 is the iteration that must be rolled back. Because the Galois run time systems functions purely at the user level, there is no simple way to abort an iteration running on another thread. To address this problem, each iteration record has an *iteration lock* as shown in Figure 11. When invoking methods on shared objects, each iteration must own the iteration lock in its record. Thus, the thread running i_1 does the following:

1. It attempts to obtain i_2's iteration lock. By doing so, it ensures that i_2 is not modifying any shared state.
2. It aborts i_2 by executing i_2's undo log and clearing the various conflict logs of i_2's invocations. Note that the control flow of the *thread* executing i_2 does not change; that thread continues as if no rollback is occurring.
3. It sets the status of i_2 to ABORTED.
4. It then resumes its execution of i_1, which can now proceed as the conflict has been resolved.

On the other side of this arbitration process, the thread executing i_2 will realize that i_2 has been aborted when it attempts to invoke another method on a shared object (or attempts to commit). At this point, the thread will see that i_2's status is ABORTED and will cease execution of i_2 and begin a new iteration.

When an iteration has to be aborted, the callbacks in its undo log are executed in LIFO order. Because the undo log must persist until an iteration commits, we must ensure that all the arguments used by the callbacks remain valid until the iteration commits. If the arguments are pass-by-value, there is no problem; they are copied when the callback is created. A more complex situation is when arguments are pass-by-reference or pointers. The first problem is that the underlying data which the reference or pointer points to may be changed during the course of execution. Thus, the callback may be called with inappropriate arguments. However, as long as all changes to the underlying data also occur through Galois interfaces, the LIFO nature of the undo log ensures that they will be rolled back as necessary before the callback uses them. The second problem occurs when an iteration attempts to free a pointer, as there is no simple way to undo a call to free. The Galois run-time avoids this problem by delaying all calls to free until an iteration commits. This does not affect the semantics of the iteration, and avoids the problem of rolling back memory deallocation.

4.4 Discussion

Set iterators: Although the Galois set iterators introduced in Section 4.1 were motivated in this paper by the two applications discussed in Section 2, they are very general, and we have found them to be useful for writing other irregular applications such as advancing front mesh generators [23], and WalkSAT solvers [34]. Many of these applications use "work-list"-style algorithms, for which Galois iterators are natural, and the Galois approach allows us to exploit data-parallelism in these irregular applications.

SETL was probably the first language to introduce an unordered set iterator [19], but this construct differs from its Galois counterpart in important ways. In SETL, the set being iterated over can be modified during the execution of the iterator, but these modifications do not take effect until the execution of the entire iterator is complete. In our experience, this is too limiting because work-list algorithms usually involve data-structure traversals of some kind in which new work is discovered during the traversal. The *tuple iterator* in SETL is similar to the Galois ordered-set iterator, but the tuple cannot be modified during the execution of the iterator, which limits its usefulness in irregular applications. Finally, SETL was a sequential programming language. DO-loops in FORTRAN are a special case of the Galois ordered-set iterator in which iteration is performed over integers in some interval.

A more complete design than ours would include iterators over multisets and maps, which are easy to add to Galois. MATLAB or FORTRAN-90-style notation like [low:step:high] for specifying ordered and unordered integers within intervals would be useful. We believe it is also advisable to distinguish syntactically between DO-ALL loops and unordered-set iterators over integer ranges, since in the former case, the programmer can assert that run-time dependence checks are unnecessary, enabling more efficient execution. For example, in the standard *i-j-k* loop nest for matrix-multiplication, the *i* and *j* loops are not only Galois-style unordered-set iterators over integer intervals but they are even DO-ALL loops; the *k* loop is an ordered-set interator if the accumulations to elements of the *C* matrix must be done in order.

Semantic commutativity: Without commutativity information, an object can be accessed by at most one iteration at a time, and that iteration shuts out other iterations until it commits. In this case, inverse methods can be implemented automatically by data copying as is done in software transactional memories.

In the applications we have looked at, most shared objects are instances of *collections*, which are variations of sets, so specifying commutativity information and writing inverse methods has been straightforward. For example, the kd-tree is just a set with an additional method for finding the nearest neighbor of an element in the set. Note that the design of Galois makes it easy to replace simple data structures with clever, hand-tuned concurrent data structures [33] if necessary, without changing the rest of the program.

The use of commutativity in parallel program execution was explored by Bernstein as far back as 1966 [4]. In the context of concurrent database systems, Weihl described a theoretical framework for using commutativity conditions for concurrency control [42]. Herlihy and Weihl extended this work by leveraging ordering constraints to increase concurrency but at the cost of more complex rollback schemes [16].

In the context of parallel programming, Steele described a system for exploiting commuting operations on memory locations in optimistic parallel execution [18]. However, commutativity is still tied to concrete memory locations and does not exploit properties of abstract data types like Galois does. Diniz and Rinard performed static analysis to determine *concrete* commutativity of methods for use in compile-time parallelization [10]. Semantic commutativity, as used in Galois, is more general but it must be specified by the class designer. Wu and Padua have proposed to use high level semantics of container classes [44]. They propose making a compiler aware of properties of abstract data types such as stacks and sets to permit more accurate dependence analysis.

Recently, Ni *et al* [24] have proposed to extend conventional transactional memory with a notion of "abstract locking" to introduce the notion of semantic conflicts. Carlstrom *et al* have taken a similar approach to the Java collections classes [6]. Semantic commutativity provides another way of specifying open nesting. More experience is needed before the relative advantages of the two approaches become clear, but we believe that semantic commutativity is an easier notion for programmers to understand.

5. Evaluation

We have implemented the Galois system in C++ on two Linux platforms: (i) a 4 processor, 1.5 GHz Itanium 2, with 16KB of L1, 256KB of L2 and 3MB of L3 cache per processor, and (ii) a dual processor dual-core 3.0 GHz Xeon system, with 32KB of L1 per core and 4MB of L2 cache per processor. The threading library on both platforms was pthreads.

5.1 Delaunay Mesh Refinement

We first wrote a sequential Delaunay mesh refinement program without locks, threads etc. to serve as a *reference* implementation. We then implemented a Galois version (which we call *meshgen*), and a fine-grain locking version (*FGL*) that uses locks on individual triangles. The Galois version uses the set iterator, and the runtime system described in Section 4.3. In all three implementations, the mesh was represented by a graph that was implemented as a set of triangles, where each triangle maintained a set of its neighbors. This is essentially the same as the standard adjacency list representation of graphs. For meshgen, code for commutativity checks was added by hand to this graph class; ultimately, we would like to generate this code automatically from high level commutativity specifications like those in Figure 9. We used an STL queue to implement the workset. We refer to these default implementations of meshgen and FGL as *meshgen(d)* and *FGL(d)*.

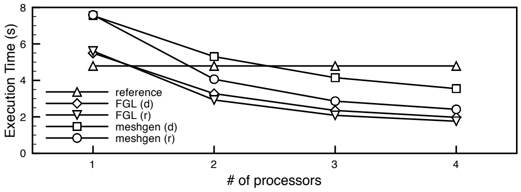
To understand the effect of scheduling policy on performance, we implemented two more versions, *FGL(r)* and *meshgen(r)*, in which the work-set was implemented by a data structure that returned a random element of the current set.

The input data set was generated automatically using Jonathan Shewchuk's Triangle program [36]. It had 10,156 triangles and boundary segments, of which 4,837 triangles were bad.

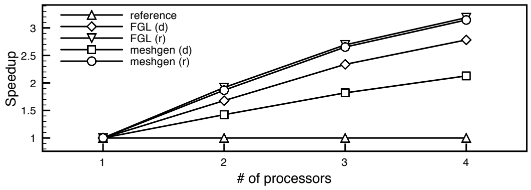
Execution times and speed-ups. Execution times and self-relative speed-ups for the five implementations on the Itanium machine are shown in Figure 12(a,b). The reference version is the fastest on a single processor. On 4 processors, FGL(d) and FGL(r) differ only slightly in performance. Meshgen(r) performed almost as well as FGL, although surprisingly, meshgen(d) was twice as slow as FGL.

Statistics on committed and aborted iterations. To understand these issues better, we determined the total number of committed and aborted iterations for different versions of meshgen, as shown in Figure 12(c). On 1 processor, meshgen executed and committed 21,918 iterations. Because of the inherent non-determinism of the set iterator, the number of iterations executed by meshgen in parallel varies from run to run (the same effect will be seen on one processor if the scheduling policy is varied). Therefore, we ran the codes a large number of times, and determined a distribution for the numbers of committed and aborted iterations. Figure 12(c) shows that on 4 processors, meshgen(d) committed roughly the same number of iterations as it did on 1 processor, but also aborted almost as many iterations due to cavity conflicts. The abort ratio for meshgen(r) is much lower because the scheduling policy reduces the likelihood of conflicts between processors. This accounts for the performance difference between meshgen(d) and meshgen(r). Because the FGL code is carefully tuned by hand, the cost of an aborted iteration is substantially less than the corresponding cost in meshgen, so FGL(r) performs only a little better than FGL(d).

It seems counterintuitive that a randomized scheduling policy could be beneficial, but a deeper investigation into the source of



(a) Execution times



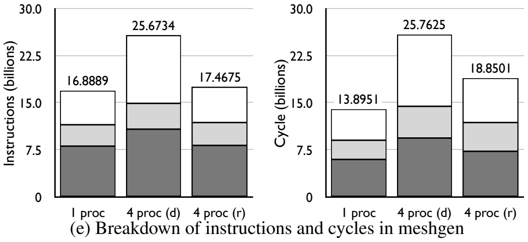
(b) Self-relative Speed-ups

	Committed			Aborted		
# of proc.	Max	Min	*Avg*	Max	Min	*Avg*
1	21918	21918	21918	n/a	n/a	n/a
4 (meshgen(d))	22128	21458	21736	28929	27711	28290
4 (meshgen(r))	22101	21738	21909	265	151	188

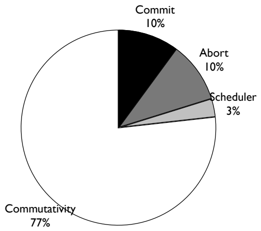
(c) Committed and aborted iterations for meshgen

Instruction Type	reference	meshgen(r)
Branch	38047	70741
FP	9946	10865
LD/ST	90064	165746
Int	304449	532884
Total	442506	780236

(d) Instructions per iteration on a single processor



(e) Breakdown of instructions and cycles in meshgen



(f) Breakdown of Galois overhead

# of procs	Client	Object	Runtime	**Total**
1	1.177	0.6208	0.6884	2.487
4	2.769	3.600	4.282	10.651

(g) L3 misses (in millions) for meshgen(r)

Figure 12. Mesh refinement results: 4-processor Itanium

cavity conflicts showed that the problem could be attributed to our use of an STL queue to implement the workset. When a bad triangle is refined by the algorithm, a cluster of smaller bad triangles may be created within the cavity. In the queue data structure, these new bad triangles are adjacent to each other, so it is likely that they will be scheduled together for refinement on different processors, leading to cavity conflicts.

One conclusion from these experiments is that domain knowledge is invaluable for implementing a good scheduling policy.

Instructions and cycles breakdown. Figure 12(d) shows the breakdown of different types of instructions executed by the reference and meshgen versions of Delaunay mesh refinement when they are run on one processor. The numbers shown are per iteration; in sequential execution, there are no aborts, so these numbers give a profile of a "typical" iteration in the two codes. Each iteration of meshgen performs roughly 10,000 floating-point operations and executes almost a million instructions. These are relatively long-running computations.

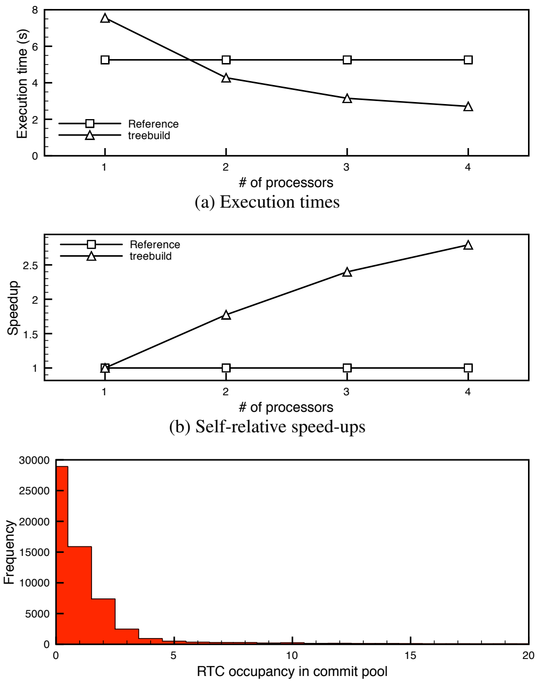
Meshgen executes almost 80% more instructions than the reference version. To understand where these extra cycles were being spent, we instrumented the code using PAPI. Figure 12(e) shows a breakdown of the total number of instructions and cycles between the client code (the code in Figure 7), the shared objects (graph and workset), and the Galois runtime system. The 4 processor numbers are sums across all four processors. The reference version performs almost 9.8 billion instructions, and this is roughly the same as the number of instructions executed in the client code and shared objects in the 1 processor version of meshgen and the 4 processor version of meshgen(r). Because meshgen(d) has a lot of aborts, it spends substantially more time in the client code doing work that gets aborted and in the runtime layer to recover from aborts.

We further broke down the Galois overhead into four categories: commit and abort overheads, which are the time spent committing iterations and aborting them, respectively; scheduler overhead, which includes time spent arbitrating conflicts; and commutativity overhead, which is the time spent performing conflict checks. The results, as seen in Figure 12(f), show that roughly three fourths of the Galois overhead goes in performing commutativity checks. It is clear that reducing this overhead is key to reducing the overall overhead of the Galois run-time.

The 1 processor version of meshgen executes roughly the same number of instructions as the 4 processor version. We do not get perfect self-relative speedup because some of these instructions take longer to execute in the 4 processor version than in the 1 processor version. There are two reasons for this: contention for locks in shared objects and the runtime system, and cache misses due to invalidations. Contention is difficult to measure directly, so we looked at cache misses instead. On the 4 processor Itanium, there is no shared cache, so we measured L3 cache misses. Figure 12(g) shows L3 misses; the 4 processor numbers are sums across all processors for meshgen(r). Most of the increase in cache misses arises from code in the shared object classes and in the Galois runtime. An L3 miss costs roughly 300 cycles on the Itanium, so it can be seen that over half of the extra cycles executed by the 4 processor version, when compared to the 1 processor version, are lost in L3 misses. The rest of the extra cycles are lost in contention.

5.2 Agglomerative clustering

For the agglomerative clustering problem, the two main data structures are the *kd-tree* and the priority queue. The kd-tree interface is essentially the same as Set, but with the addition of the nearest neighbor (`nearest`) method. The priority queue is an instance of a poSet. Since the priority queue is used to sequence iterations, the removal and insertion operations (`get` and `add` respectively) are orchestrated by the commit pool.



(a) Execution times

(b) Self-relative speed-ups

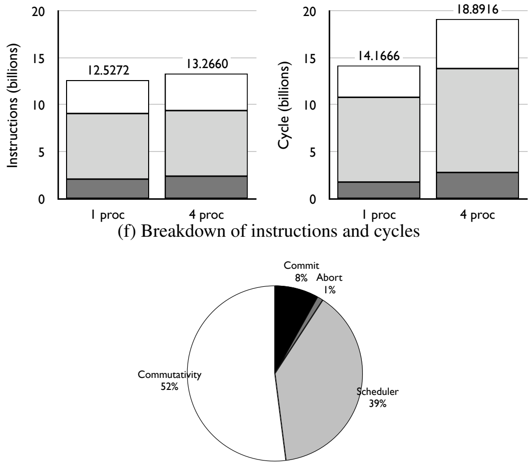
(c) Commit pool occupancy by `RTC` iterations

	Committed			Aborted		
# of proc.	Max	Min	*Avg*	Max	Min	*Avg*
1	57846	57846	57846	n/a	n/a	n/a
4	57870	57849	57861	3128	1887	2528

(d) Committed and aborted iterations in treebuild

Instruction Type	reference	treebuild
Branch	7162	18187
FP	3601	3640
LD/ST	22519	48025
Int	70829	146716
Total	104111	216568

(e) Instructions per iteration on a single processor.



(f) Breakdown of instructions and cycles

(g) Breakdown of Galois overhead

# of procs	User	Object	Runtime	**Total**
1	0.5583	3.102	0.883	4.544
4	2.563	12.8052	5.177	20.545

(h) Number of L3 misses (in millions) on different numbers of processors.

Figure 13. Agglomerative clustering results: 4-processor Itanium

	meshgen(p)		treebuild	
Cores	Time (s)	Speedup	Time (s)	Speedup
1	12.5	1.0	8.19	1.0
2 (non-shared L2)	8.1	1.5	7.77	1.05
2 (shared L2)	6.7	1.9	4.66	1.78

Table 1. Results on dual-core, dual-processor Intel Xeon

To evaluate the agglomerative clustering algorithm, we modified an existing graphics application called lightcuts that provides a scalable approach to illumination [41]. This code builds a light hierarchy based on a distance metric that factors in Euclidean distance, light intensity and light direction. We modified the objects used in the light clustering code to use Galois interfaces and the poSet iterator for tree construction. The overall structure of the resulting code was discussed in Figure 5. We will refer to this Galois version as *treebuild*. We compared the running time of *treebuild* against a *reference* version which performed no threading or locking.

Figure 13 shows the results on the Itanium machine. These results are similar to the Delaunay mesh generation results discussed in Section 5.1, so we describe only the points of note. The self-relative speed-ups in Figure 13(b) show that despite the serial dependence order imposed by the priority queue, the Galois system is able to expose a significant amount of parallelism. The mechanism that allows us to do this is the commit pool, which allows threads to begin execution of iterations even if earlier iterations have yet to commit. To understand the role of the commit pool quantitatively, we recorded the number of iterations in `RTC` state every time the commit pool created, aborted or committed an iteration. This gives an idea of how deeply into the ordered set we are speculating to keep all the processors busy. Figure 13(c) shows a histogram of this information (the x-axis is truncated to reveal detail around the origin). We see that most of the time, we do not need to speculate too deeply. However, on occasion, we must speculate over 100 elements deep into the ordered set to continue making forward progress. Despite this deep speculation, the number of aborted iterations is relatively small because of the high level of parallelism in this application, as discussed in Section 2.2. Note that commit pool occupancy is not the same as parallelism in the problem because we create iteration records in the commit pool only when a thread needs work; the priority queue is distinct from the commit pool. We also see that due to the overhead of managing the commit pool, the scheduler accounts for a significant percentage of the overall Galois overhead, as seen in Figure 13(g).

Figure 13(h) shows that most of the loss in self-relative speedup when executing on 4 processors is due to increased L3 cache misses from cache-line invalidations.

5.3 Results on Xeon

To confirm the role of cache invalidation misses, we investigated the performance of meshgen and treebuild on a dual-core, dual processor Xeon system. In this asymmetric architecture, cores on the same package share the lowest level of cache (in this case, L2). Therefore, a program run using two cores on the same package will incur no L2 cache line invalidations, while the same program running on two cores on separate packages will suffer from additional cache invalidation misses (capacity misses may be reduced because the effective cache size doubles).

Table 1 shows the performance of the two programs when run on a single core and on two cores. We see that when the two cores are on the same package, we achieve near-perfect speedup, but the speedup is much less when the two cores are on separate packages. This confirms that a substantial portion of efficiency loss arises from cache line invalidations due to data sharing, so further improvements in performance require attending to locality.

6. Conclusions and Ongoing Work

The Galois system is the first practical approach we know of for exploiting data-parallelism in work–list based algorithms that deal with complex, pointer-based data structures like graphs and trees. Our approach is based on (1) a small number of syntactic constructs for packaging optimistic parallelization as iteration over mutable ordered and unordered sets, (2) assertions about methods in class libraries, and (3) a runtime scheme for detecting and recovering from potentially unsafe accesses to shared memory made by an optimistic computation. The execution model is an object-based shared-memory model. By exploiting the high level semantics of abstract data types, the Galois system is able to allow concurrent accesses and updates to shared objects. We have some experience in massaging existing object-oriented codes in C++ to use the Galois approach, and the effort has not been daunting at least for codes that use collections of various sorts.

Our experimental results show that (1) our approach is promising, (2) scheduling iterations to reduce aborted computations is important, (3) domain knowledge may be important for good scheduling, and (4) locality enhancement is critical for obtaining better performance than our current approach is able to provide.

Our application studies suggest that the objective of compile-time analysis techniques such as points-to and shape analysis should be to improve the efficiency of optimistic parallelization, rather than to perform static parallelization of irregular programs. These techniques might also help in verification of commutativity conditions against a class specification. Static parallelization works for regular programs because the parallelism in dense-matrix algorithms is independent of the values in dense matrices. Irregular programs are fundamentally different, and no static analysis can uncover the parallelism in many if not most irregular applications.

While exposing and exploiting parallelism is important, one of the central lessons of parallel programming is that exploiting locality is critical for scalability. Most work in locality enhancement has focused on regular problems, so new ideas may be required to make progress on this front. We believe that the approach described in this paper for exposing parallelism in irregular applications is the right foundation for solving the problem of exploiting parallelism in irregular applications in a scalable way.

Acknowledgments

We would like to thank Khubaib Khubaib in our group for his measurements of Galois overhead, and Ramesh Peri and David Levinthal at Intel, Austin Division for their help with VTune. Finally, we would like to thank Tim Harris for being a good shepherd on behalf of the PLDI program committee.

References

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[2] Christos D. Antonopoulos, Xiaoning Ding, Andrey Chernikov, Filip Blagojevic, Dimitrios S. Nikolopoulos, and Nikos Chrisochoides. Multigrain parallel delaunay mesh generation: challenges and opportunities for multithreaded architectures. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, 2005.

[3] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 1966.

[5] Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. Technical Report IBM RC 21055, IBM Yorktown Heights, 1997.

[6] Brian D. Carlstrom, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Transactional collection classes. In *Principles and Practices of Parallel Programming (PPoPP)*, 2007.

[7] C.C.Foster and E.M.Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Transactions on Computers*, 21(12):1411–1415, 1972.

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

[9] Johan de Galas. The quest for more processing power: is the single core CPU doomed? http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377, February 2005.

[10] Pedro C. Diniz and Martin C. Rinard. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997.

[11] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 263–273, New York, NY, USA, 1998. ACM Press.

[12] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *isca*, 00:102, 2004.

[13] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003.

[14] John Hennessy and David Patterson, editors. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[15] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[16] Maurice P. Herlihy and William E. Weihl. Hybrid concurrency control for abstract data types. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 201–210, New York, NY, USA, 1988.

[17] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[18] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th symposium on Principles of Programming Languages*, pages 218–231, 1990.

[19] J.T.Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: An introduction to SETL.* Springer-Verlag Publishers, 1986.

[20] Ken Kennedy and John Allen, editors. *Optimizing compilers for modren architectures:a dependence-based approach*. Morgan Kaufmann, 2001.

[21] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[22] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and preliminary architectural sketches. In *SCOOL '05: Sychronization and Concurrency in Object-Oriented Languages*, 2005.

[23] J.B.C Neto, P.A. Wawrzynek, M.T.M. Carvalho, L.F. Martha, and A.R. Ingraffea. An algorithm for three-dimensional mesh generation for arbitrary regions with cracks. *Engineering with Computers*, 17:75–91, 2001.

[24] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Rick Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Principles and Practices of Parallel Programming (PPoPP)*, 2007.

[25] Openmp: A proposed industry standard api for shared memory programming. See `www.openmp.org`, October 28, 1997.

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

[27] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993.

[28] Hany E. Ramadan, Donald E. Porter Christopher J. Rossbach, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Transactional memory designs for an operating system. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[29] Lawrence Rauchwerger and David A. Padua. Parallelizing while loops for multiprocessor systems. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 347–356, Washington, DC, USA, 1995. IEEE Computer Society.

[30] 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.

[31] Jim Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 83–92, 1993.

[32] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[33] William Scherer and Michael Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, 1996.

[34] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[35] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[36] 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. May 1996.

[37] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[38] Robert Tomasulo. An algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11(1):25–33, 1967.

[39] T. N. Vijaykumar, Sridhar Gopal, James E. Smith, and Gurindar Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.

[40] Christoph von Praun, Luis Ceze, and Calin Cascaval. Implicit parallelism with ordered transactions. In *Principles and Practices of Parallel Programming (PPoPP)*, 2007.

[41] 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.

[42] W.E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12), 1988.

[43] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

[44] Peng Wu and David A. Padua. Beyond arrays - a container-centric approach for parallelization of real-world symbolic applications. In *LCPC '98: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 197–212, London, UK, 1999.

[45] L. Rauchwerger Y. Zhan and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 162, 1998.