DOI:10.1145/1562164.1562188

Optimistic Parallelism Requires Abstractions

By Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew

Abstract

The problem of writing software for multicore processors is greatly simplified if we could automatically parallelize sequential programs. Although auto-parallelization has been studied for many decades, it has succeeded only in a few application areas such as dense matrix computations. In particular, auto-parallelization of irregular programs, which are organized around large, pointer-based data structures like graphs, has seemed intractable.

The Galois project is taking a fresh look at auto-parallelization. Rather than attempt to parallelize all programs no matter how obscurely they are written, we are designing programming abstractions that permit programmers to highlight opportunities for exploiting parallelism in sequential programs, and building a runtime system that uses these hints to execute the program in parallel. In this paper, we describe the design and implementation of a system based on these ideas. Experimental results for two real-world irregular applications, a Delaunay mesh refinement application and a graphics application that performs agglomerative clustering, demonstrate that this approach is promising.

1. INTRODUCTION

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

Irregular applications are organized around pointer-based data structures such as graphs and trees, and are ubiquitous in important application areas such as finite-elements, SAT solvers, maxflow computations, and compilers. In principle, it is possible to use a thread library (e.g., pthreads) or a combination of compiler directives and libraries (e.g., OpenMP) to write parallel code for irregular applications, but it is well known that writing explicitly parallel code can be very tricky because of the complexities of memory consistency models, synchronization, data races, etc. Tim Sweeney, who designed the multithreaded Unreal 3 game engine, estimates that writing multithreading code tripled software costs at Epic Games (quoted in de Galas[3]).

From the earliest days of parallel computing, it has been recognized that one way to circumvent the problems of writing explicitly parallel code is *auto-parallelization*.[10] In this approach, application programmers write sequential programs, leaving it to the compiler or runtime system to extract and exploit the latent parallelism in programs. There is an enormous literature on algorithms and mechanisms for auto-parallelization, but like the characters in Pirandello's play *Six Characters in Search of an Author*, most of them are in search of programs that they can parallelize. They can be divided into two categories: compile-time techniques and runtime techniques. Compile-time techniques use static analyses to find independent computations in programs, and have succeeded in parallelizing limited classes of irregular programs such as n-body methods.[1, 5, 20] Runtime techniques use optimistic parallelization: computations are parallelized speculatively, and the runtime system detects conflicts between concurrent computations and rolls them back as needed to preserve the sequential semantics of the program. Optimistic parallelism is the basis of the popular Timewarp algorithm for parallel event-driven simulation,[9] but efforts to build general-purpose systems based on optimistic parallelization, such as thread-level speculation (TLS),[19, 22, 24] have had limited success. Because of these problems, interest in auto-parallelization has waned in recent years.

We are taking a fresh look at auto-parallelization, but from a different perspective than prior work in this area. Instead of trying to parallelize all application programs no matter how obscurely written, the Galois project is focusing on the following questions.

- Can we design sequential programming abstractions that capture the most commonly occurring parallelism patterns in programs?
- If so, what systems support is needed to auto-parallelize programs that use these abstractions?

A useful analogy is relational database programming. The SQL programmer views data as if they were organized as a flat table (relations), and operates on the data using high-level operations like joins and projections. Inside the database system, relations are implemented in very complex ways using B-trees, index structures, etc., and the high-level operations are performed in parallel using locks and transactions, but the relational abstractions enable these complications to be hidden from the SQL programmer.

Can we carry out a similar program for irregular applications? Although we are far from having a complete solution, the outlines of a solution for important patterns of parallelism are emerging from the fog. In this paper, we focus on understanding and exploiting parallelism in *iterative* irregular applications. In Section 2, we describe parallelism patterns in two such applications: a Delaunay mesh refinement

The original version of this paper appeared in the *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

code[2] and a graphics application[23] that performs agglomerative clustering.[17] In Section 3, we discuss the Galois programming model and runtime system for exploiting this parallelism. In Section 4, we evaluate the performance of our system on the two applications. Finally, in Section 5, we discuss conclusions and ongoing work.

2. TWO IRREGULAR APPLICATIONS

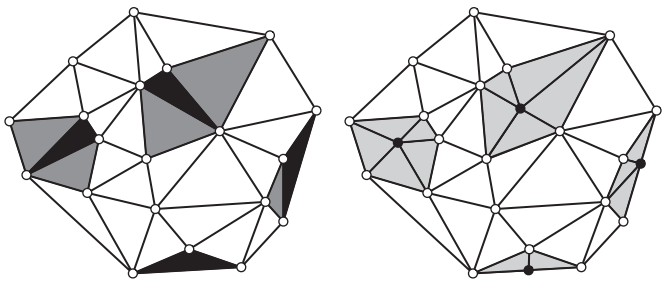
In this section, we describe opportunities for exploiting parallelism in two irregular applications: Delaunay mesh refinement,[2] and agglomerative clustering[17] as used within a graphics application.[23] These applications perform refinement and coarsening, respectively, which are arguably the two most common operations for bulk modification of irregular data structures.

2.1. Delaunay mesh refinement

The input to the 2D Delaunay mesh refinement algorithm is a triangulation of some region in the plane, in which all triangles satisfy a certain geometric property called the Delaunay condition.[2] Some of these triangles may be badly shaped according to certain geometric criteria; for example, excessively large triangles may cause unacceptable discretization errors in finite-element solutions. The goal of mesh refinement is to eliminate these badly shaped triangles from the mesh by replacing them with smaller triangles. However, performing this operation on a bad triangle may violate the Delauanay condition for neighboring triangles, so it is necessary to find all affected triangles (this is called the *cavity* of that bad triangle), and retriangulate the entire cavity. Figure 1 shows the initial mesh on the left (badly shaped triangles are colored black, and cavities are colored gray), and the refined mesh on the right. Refinement may create new badly shaped triangles, but there is a mathematical guarantee, at least in 2D, that if this process is repeated, a mesh without bad triangles will be produced in the end. The structure of the final mesh may depend on the order in which bad triangles are eliminated, but any mesh produced by this process is acceptable.

Figure 2 shows the pseudocode for mesh refinement. It is natural to organize the program around a work-list containing the bad triangles, as seen in lines 3 and 4. This work-list is one of the two key data structures in mesh refinement. 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.

Figure 1. Fixing bad elements.



(a) Unrefined Mesh (b) Refined Mesh

Figure 2. Pseudocode of the mesh refinement algorithm.

```
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:  }
```

Opportunities for Exploiting Parallelism: Delaunay mesh refinement is a relatively complicated code since the central data structure is a graph that is modified repeatedly during the execution of the algorithm. Nevertheless, there may be a lot of parallelism in the algorithm since cavities that do not overlap can be processed in parallel, as in the mesh of Figure 1. If two cavities overlap, they must be processed sequentially in some order. How much parallelism is there in mesh refinement? Our studies[11] have shown that for a mesh of 100,000 triangles in which roughly half the initial triangles are badly shaped, there are more than 256 triangles that can be processed in parallel until almost the end of execution.

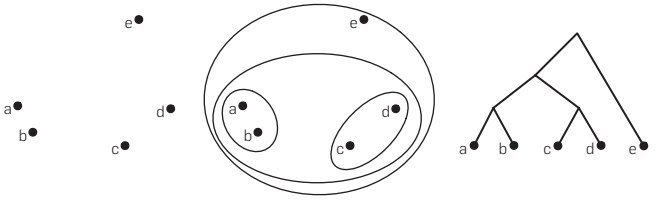
2.2. Agglomerative clustering

The second problem is *agglomerative clustering*, a well-known data-mining algorithm.[17] This algorithm is used in graphics applications for handling large numbers of light sources.[23]

The input to the clustering algorithm is (1) a data-set and (2) a measure of the similarity between items in the data-set. 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 3(a) shows a data-set containing points in the plane, for which the measure of similarity between data points is Euclidean distance. The dendrogram for this data-set is shown in Figure 3(b) and (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.[17] The algorithm terminates when there is only one point left in the data-set. Pseudocode for the algorithm is shown in Figure 4.

Figure 3. Agglomerative clustering.



(a) Data points (b) Hierarchical clusters (c) Dendrogram

Figure 4. Pseudocode for agglomerative clustering.

```
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: }
```

The algorithm iterates over 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 pair of points at the head of the priority queue—the closest pair—are clustered. These two points are replaced by a new, representative point. The nearest neighbor of this point is determined, and the pair is entered into the priority queue.

To find the nearest neighbor of a point, we can scan the entire data-set at each step, but this is too inefficient. Instead, we use a spatial acceleration structure called a *kd-tree* to find nearest neighbors. The kd-tree is built at the start of the algorithm and is kept up to date as points are removed and added from the space, as seen in Figure 4.

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. However, if we consider the data-set in Figure 3(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[11]; thus, there is substantial parallelism that can be exploited.

2.3. Discussion

Existing compile-time parallelization techniques for irregular programs are based on *shape analysis*,[20] which determines structural invariants in the data structures. The graph in mesh refinement has no particular structure, and it is also modified in each iteration of the loop in Figure 2, so compile-time parallelization will not work for this application. Semi-static approaches using the *inspector–executor* model[18] split computation into two phases: an inspector phase that determines dependences between units of work and constructs a computation schedule and an executor phase that executes the resulting schedule in parallel. This approach does not work for mesh refinement since the dependence structure changes when the underlying graph is modified by the algorithm.

These considerations suggest that a fully dynamic approach in which dependences are detected at runtime is needed to parallelize codes like mesh refinement and agglomerative clustering. One such approach to parallelizing mesh refinement has been proposed by Hudson et al.,[8] and it has the following steps: (1) compute the cavities of all bad triangles without making any modifications to the graph, (2) build an interference graph in which nodes represent cavities and edges represent overlapping cavities, (3) find a maximal independent set of nodes in this graph, and (4) retriangulate the cavities corresponding to these nodes in parallel, without any synchronization. These steps are then repeated for the new mesh until convergence. This approach can be viewed as an extended inspector–executor approach in which the execution of the inspector and executor are interleaved. However, this approach is very specific to Delaunay mesh refinement. For example, it is not clear that it can be used for applications like agglomerative clustering in which iterations are performed over priority queues.

3. THE GALOIS APPROACH

The analysis of Section 2 suggests that optimistic parallelization, in which computations are speculatively executed in parallel and rolled back selectively when dependence conflicts are detected by the runtime system, is the only general-purpose approach to exploiting parallelism in irregular applications. In this section, we argue that optimistic parallelization needs to be coupled with appropriate programming abstractions, and we describe an implementation of these ideas in the Galois system.

The need for programming abstractions becomes evident if we consider the mesh refinement code in Figure 2. The work-list of bad triangles determines a particular order in which bad triangles are processed by the sequential program, and any auto-parallelized version of this code will be forced to process bad triangles in the same order. The fact that bad triangles can actually be processed in any order is important for parallelization, but it is missing from this code. Abstractly, we can view the processing of each bad triangle as an operator that is applied to the graph to modify a small region of it; the fact that bad triangles can be processed in any order is equivalent to asserting that the applications of this operator to the graph "commute" with each other. Since the structure of the final graph may actually be different for different operator orderings, we call this *application-specific commutativity* for obvious reasons.

Opportunities to exploit commutativity may also arise in abstract data type (ADT) implementations. Consider a set ADT that is implemented using a linked list. With respect to the semantics of sets, insert operations commute with each other since all insertion orders produce the same set even though the linked list representation internal to the ADT may be different for different insertion orders. In this case,

commutativity arises from the fact that there may be several concrete (memory) states that represent a single abstract state. Exploiting this kind of *ADT commutativity* obviously requires an object-oriented language.

We will refer to application-specific and ADT commutativity as *semantic commutativity*. In contrast, traditional compile-time parallelization techniques such as dependence analysis[10] and Diniz and Rinard's commutativity analysis[4] focus on *concrete commutativity* in which all orders of performing operations result in the same concrete state. Semantic commutativity is more general, and it permits more interleavings of operations.

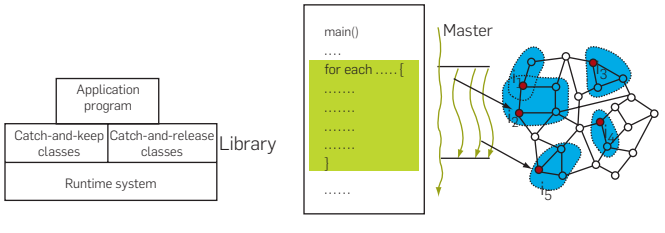
The programming abstractions introduced in this section permit programmers to highlight opportunities for exploiting semantic commutativity. They can be added to any sequential object-oriented language (the results in this paper are from an implementation in C++). Figure 5(a) is a conceptual picture of the Galois system. Application programs use two constructs called *Galois set iterators*, described in Section 3.1, for highlighting opportunities for exploiting parallelism. Section 3.2 describes the data structure library. Data structures in which there are opportunities to exploit ADT commutativity are implemented by *catch-and-release* classes, while others are implemented in *catch-and-keep* classes. The runtime system implements optimistic parallelization, and detects and recovers from potentially unsafe accesses to shared objects, as explained in Section 3.3.

3.1. Galois set iterators

The Galois programming model is sequential and object-oriented; programs are written in an object-oriented language like C++ or Java extended with two constructs called Galois set iterators.

- **Unordered-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 refinement, 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 5. The Galois system.



(a) Layered architecture (b) Execution model

Note that new elements may be added to a set while iterating over it, which is not allowed in conventional set iterators in languages like SETL or Java. Figure 6 shows the client code for Delaunay mesh generation. Instead of a work-list of bad triangles, this code uses a set of bad triangles and a set iterator. Since set elements are not ordered, the iterator is permitted to iterate over the set in any order. Therefore, the Galois version makes evident the fact that the bad triangles can be processed in *any* order, a fact that is absent from the more conventional code of Figure 2. 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.

The parallel execution model is shown in Figure 5(b). A master thread executes all code outside the Galois set iterators. When it encounters a Galois set iterator, it enlists worker threads to help execute iterations concurrently. The assignment of iterations to threads is done dynamically, but this policy can be changed by an expert programmer.[12] Threads synchronize by barrier synchronization at the end of the iterator.

Given this execution model, the main technical problem is to ensure that the parallel execution respects the sequential semantics of the iterators. For an unordered-set iterator, this can be accomplished by ensuring that the execution of each iteration has *transactional semantics*. These semantics guarantee *serializability*; the parallel execution will behave as if the iterations were executed serially in some order. For the ordered-set iterator, we must also ensure that the iterations appear to execute in the order prescribed by the ordering on set elements. Guaranteeing sequential semantics is a nontrivial problem because each iteration may read and write objects in shared memory, so we must ensure that these reads and writes are properly coordinated. Next, we describe how this is accomplished.

3.2. Galois library classes

The library has two kinds of classes: catch-and-keep and catch-and-release. As in Java, a lock is automatically associated with each object, but the locking policy is different for the two kinds of classes.

Catch-and-Keep Classes: Catch-and-keep classes are the default, and they are implemented in Galois using a variation of the well-known two-phase locking policy. To invoke a method on a catch-and-keep object, an iteration must

Figure 6. Delaunay mesh refinement using set iterator.

```
1:  Mesh m = /* read in initial mesh */
2:  Set wl;
3:  wl.add (mesh.badTriangles ( ) );
4:  for each e in wl do {
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: }
```

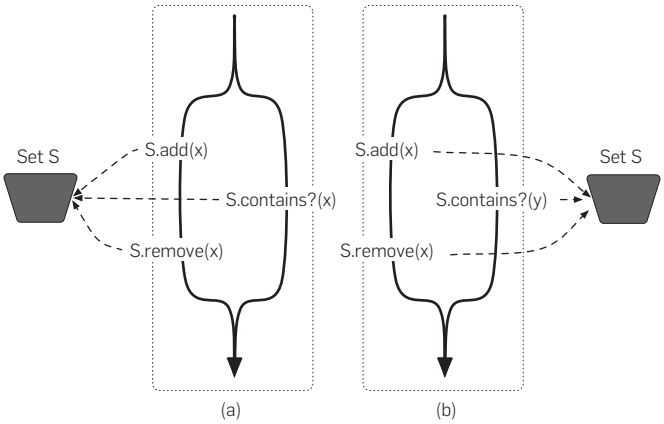
first acquire the lock associated with it. This lock is held until the iteration terminates, at which point the iteration releases all of its locks. If an iteration is unable to acquire a lock on an object, this means that a second iteration is currently accessing the object, and one of the two iterations must be rolled back. Rollbacks are accomplished by copying an object before it is modified, and restoring from that copy upon rollback. Thus, in a system in which all objects use the catch-and-keep policy, serialization of iterations is easy to ensure. Acquiring and releasing locks, making backup copies of objects, etc., is performed automatically by our runtime system, as explained in Section 3.3. It is also possible to use hardware transactional memory (TM).[7]

While catch-and-keep classes are simple to implement, they may not provide enough concurrency. Work-sets are themselves data structures, and they are implemented using classes in the Galois library. Since each iteration gets an element from the work-set at the beginning and may add elements to it at the end, a catch-and-keep implementation of the work-set class would permit only one iteration to execute at a time.

Catch-and-Release Classes: To solve this problem, the Galois system supports the catch-and-release policy for concurrently accessed objects such as the graph and work-set in mesh refinement, or the kd-tree in agglomerative clustering. To access a catch-and-release object, an iteration must acquire the lock on the object. However, unlike in catch-and-keep, the lock is released as soon as the method completes. Releasing the lock allows interleaving of method invocations from different iterations, which increases concurrency.

The key problem in catch-and-release is to ensure that the method interleavings do not violate serializability of iterations. This is nontrivial, as demonstrated by the programs in Figure 7, which show iterations manipulating a set that supports `add`, `remove`, and `contains`, with the standard semantics. In Figure 7(a), we see that in any sequential execution, the call `contains(x)` will return false. However, for the interleaving shown in the figure, the call will return true. On the other hand, for the program in Figure 7(b), all possible interleavings of the methods match a serial execution. For a catch-and-release object, how can we determine which interleavings are legal, and which should be disallowed?

Figure 7. Interleaving method invocations from different iterations.



The key is semantic commutativity, described at the beginning of this section. Method invocations to a given object from two iterations can be interleaved safely provided that the final abstract state is consistent with some serial order of iteration execution. In Figure 7(a), the invocation `contains(x)` does not commute with the operations from the other thread, so the invocations from the two iterations must not be interleaved. In Figure 7(b), `contains(y)` commutes with the other operations, so the iterations can execute in parallel. Note that commutativity may depend on the arguments or return values of methods.

Because iterations are executed in parallel, it is possible for commutativity conflicts to prevent an iteration from completing, requiring that iterations be rolled back. Because semantic commutativity does not track the concrete state of an object, simply creating copies of the concrete state (as in catch-and-keep classes) does not suffice. Instead, every method of a catch-and-release 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 that does not contain x, the inverse of a method invocation that adds an element x to a set is a method invocation that removes it from that set. 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.

Putting It All Together: ADT commutativity and undo must be specified by the class designer. Figure 8 illustrates how

Figure 8. Example Galois class for a set.

```
class Set {
   //interface methods
   void add (Element x);
      [commutes]
         – add (y) {y != x}
         – remove (y) {y != x}
         – contains (y) {y != x}
      [inverse] remove (x)
   void remove (Element x);
      [commutes]
         – add (y) {y != x}
         – remove (y) {y != x}
         – contains (y) {y != x}
      [inverse] add (x)
   boolean contains (Element x);
      [commutes]
         – add (y) {y != x}
         – remove (y) {y != x}
         – contains (*)//any call to contains
}
```

this information is specified in Galois for a class that implements sets. For each method, the implementor specifies the following:

- *Commutes*: This section specifies which other methods the current method commutes with, and under which conditions. For example, `remove(x)` commutes with `add(y)`, as long as the elements are different.
- *Inverse*: This section specifies the inverse of the current method.

Note that `add(x)` does not commute with `add(x)` according to this specification. This is because rolling back `add(x)` requires invoking `remove(x)`, which would conflict with other invocations of `add(x)`. This choice simplifies the implementation.

3.3. 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) structures called *conflict logs* which detect when commutativity conditions are violated for catch-and-release objects.

The commit pool maintains an *iteration record*, shown in Figure 9, 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, (1) the conflict log of that object is updated, as described in more detail below and (2) 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.

Figure 9. Iteration record maintained by runtime system.

```
IterationRecord  {
     Status status;
     Priority p;
     UndoLog ul;
     Lock 1;
}
```

Conflict Logs: The *conflict log* is the mechanism for detecting commutativity conflicts. There is one conflict log associated with each catch-and-release 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.

Consider the effects of calling `contains(x)` on a set implementing the interface shown in Figure 8. The conflict log contains all the outstanding invocations of methods on the set. Because `contains` can only conflict with `add` or `remove`, the runtime system will scan the log to ensure that no other iteration called `add(x)` or `remove(x)`.

Note that for efficiency, a runtime system may use an optimized implementation of conflict logs which does not require a full scan of all outstanding method invocations to detect conflicts. The full version of this paper describes a number of these optimizations in more detail.[14]

Commit Pool: When an iteration attempts to commit, the commit pool checks two things: (1) that the iteration is at the head of the commit queue, and (2) that the priority of the iteration is higher than all the elements left in the set/Poset being iterated over. 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 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 runtime functions at the user-level, there is no way to roll back an iteration running on another thread. Instead, i_1 undoes the effects of i_2 without explicitly rolling back execution. Next, i_1 sets a flag on i_2 telling it to roll back. When the thread running i_2 invokes a shared method or attempts to commit, it checks this flag and completes the rollback.

When an iteration has to be aborted, the callbacks in its undo log are executed in LIFO order. Note that the arguments used by the callback must have the values present when the callback was created. This is ensured due to the LIFO ordering of the undo log, as any later changes to the arguments will be undone first.

3.4. Discussion

There is no analog of unordered-set iterators or catch-and-release objects in current TLS systems[22, 24] (in fact, most of these systems auto-parallelize programs in FORTRAN and C, which have no notion of data abstraction). It is possible that this might account for the limited performance of these systems.

The TM paper of Herlihy and Moss[7] has inspired a vast literature on transactions and TM (see Larus and Rajwar[15] for a survey of the more important results). The starting point for the transactional approach is an explicitly parallel program, and the focus is on reducing the complexities and overhead of synchronization through the use of the transactional model. In contrast, our starting point is a sequential program, and the focus is on auto-parallelization. The Galois runtime system exploits optimistic parallelism just as TM exploits optimistic synchronization. Hardware TM can be used to implement catch-and-keep classes with low overhead, but catch-and-release classes must be supported in software. Herlihy and Koskinen have recently introduced catch-and-release objects into a software TM to *boost* its performance.[6]

Ni et al.[16] have proposed to extend the conventional transactional model with open nested transactions and abstract locking to allow more abstract conflict checking. Open nesting is a *mechanism*, and it does not specify how the abstract locks should be used. Semantic commutativity provides the appropriate definition of semantic conflict for data structures, and open nesting is one possible means of implementing semantic commutativity.

4. EVALUATION

Our initial implementation of the Galois system was in C++. Our evaluation platform was a 4 processor, 1.5 GHz Itanium 2, with 16KB of L1, 256KB of L2 and 3MB of L3 cache per processor. The threading library was pthreads.

4.1. 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*), as well as an explicitly parallel, fine-grain locking program (*FGL*) that uses locks on individual triangles. The Galois version uses the set iterator, and the runtime system described in Section 3.3. 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. 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 8. We used an STL queue to implement the work-set. We refer to these default implementations of meshgen and FGL as *meshgen(d)* and *FGL(d)*.

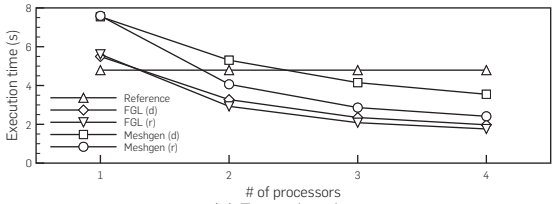
To understand the effect of scheduling policy on performance, we implemented two more versions, *FGL(r)* and *meshgen(r)*, in which the work-set was implemented by a data structure that returned a random element of the work-set.

The input data-set was generated automatically using Jonathan Shewchuk's Triangle program.[21] It had 10,156 triangles and boundary segments, of which 4,837 triangles were bad.

Execution Times and Speedups: Execution times for the five implementations on the Itanium machine are shown in Figure 10(a). The reference version is the fastest on a single processor. On four 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 10(b). On one processor, meshgen executed and committed 21,918 iterations. Because of the inherent nondeterminism of the set

Figure 10. Mesh refinement results.



(a) Execution times

	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

(b) Committed and aborted iterations for meshgen

Source of overhead	% of overhead
Abort	10
Commit	10
Scheduler	3
Commutativity	77

(c) Breakdown of Galois overhead for meshgen(r)

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 10(b) shows that on four processors, meshgen(d) committed roughly the same number of iterations as it did on one 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. The lower abort ratio 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 cavity conflicts showed that the problem could be attributed to our use of an STL queue to implement the work-set. 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.

Overhead Breakdown: The Galois system introduces some overhead over the reference code, even when running on one processor; meshgen(r) takes 58% longer to execute the reference code on the same input. To understand the overheads of the Galois implementations, we instrumented the code using PAPI. We broke down the Galois overhead into four categories: (1) commit overhead, (2) abort overhead, (3) scheduler overhead, which includes time spent arbitrating conflicts, and (4) commutativity check overhead. The results, as seen in Figure 10(c), 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 runtime.

4.2. 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.

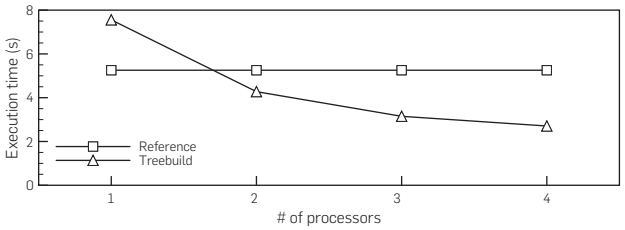
To evaluate the agglomerative clustering algorithm, we modified an existing graphics application called lightcuts that provides a scalable approach to illumination.[23] The code builds a light hierarchy based on a distance metric that factors in Euclidean distance, light intensity, and light direction. 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 4. We will refer to this Galois version as *treebuild*. We compared the running time of *treebuild* against a sequential *reference* version.

Figure 11 gives the performance results. These results are similar to the Delaunay mesh generation results discussed in Section 4.1, so we describe only the points of note. The execution times in Figure 11(a) 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. The overhead introduced by the Galois system is 44% on a single processor. We 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 11(c).

4.3. Ongoing work

In recent work, we introduced the notion of logical *data partitioning* into the Galois system.[13] For mesh refinement, each partition of the graph is mapped to a core, and each core processes bad triangles in its own partition. This mapping reduces the likelihood of conflicts since different cores work in different regions of the graph; unlike the randomized schedule discussed in Section 4, this approach also promotes locality of reference. Furthermore, commutativity checks, which are expensive, can be replaced with locking on partitions. Over-decomposition of the graph increases the likelihood that a core will have work to do even if some of its partitions are locked by other cores

Figure 11. Agglomerative clustering results.



(a) Execution times

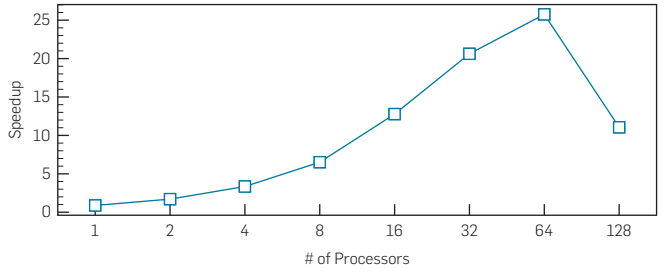
	Committed			Aborted		
# of proc.	Max	Min	Avg	Max	Min	Avg
1	57846	57846	57846	n/a	n/a	n/a
4	57870	57849	57861	3128	1887	2528

(b) Committed and aborted iterations in treebuild

Source of overhead	% of overhead
Abort	1
Commit	8
Scheduler	39
Commutativity	52

(c) Breakdown of Galois overhead

Figure 12. Speedup vs. # of processors for mesh refinement.



working on cavities that span multiple partitions. However, load-balancing is more of a problem than in the baseline approach. We also developed a scheduling framework that gives programmers control over the scheduling policy used by the Galois runtime.[12]

We produced a new implementation of the Galois system in Java, which incorporates these changes. We then evaluated an implementation of mesh refinement, using an input mesh of 100,000 triangles. Figure 12 shows the performance of this implementation on a 128-core Sunfire system, normalized to a sequential implementation in plain Java. We see that the Galois system is able to achieve significant speedup up to 64 cores, beyond which load imbalance and communication latency begin to dominate performance.

5. CONCLUSION

In this paper, we described the Galois system, which is a fresh approach to automatic parallelization of irregular applications. Rather than attempt to parallelize all programs no matter how obscurely they are written, our system provides programming abstractions that programmers use to highlight opportunities for exploiting parallelism. The runtime system uses optimistic parallelization to exploit these opportunities for parallel execution highlighted by the programmer. It detects conflicts between concurrent computations, and rolls back computations appropriately to preserve the sequential semantics of the program. Experimental results for two real-world irregular applications, a Delaunay mesh refinement application and a graphics application that performs agglomerative clustering, demonstrate that this approach is promising.

The Galois approach should be viewed as a baseline parallel implementation for irregular applications. Handwritten parallel versions of many irregular applications exploit particular kinds of structure in these applications to reduce parallel overheads. How do we identify such opportunities for exploiting structure in irregular programs? Can the relevant optimizations be performed automatically by the compiler? How do we reduce runtime overheads? These are some of the exciting research opportunities that lie ahead.

This work is supported in part by NSF grants 0833162, 0719966, 0702353, 0724966, 0739601, and 0615240, as well as grants from IBM and Intel Corporation.

References

1. Burke, M., Carini, P., Choi, J.-D. *Interprocedural Pointer Alias Analysis*. Technical Report IBM RC 21055, IBM Yorktown Heights, 1997.
2. Chew, L.P. Guaranteed-quality mesh generation for curved surfaces. In *SCG'93: Proceedings of the 9th Annual Symposium on Computational Geometry* (1993), 274–280.
3. de Galas, J. The quest for more processing power: is the single core CPU doomed? http://www.anandtech.com/cpuchipsets/showdoc.aspx?I=2377, February 2005.
4. Diniz, P.C., Rinard, M.C. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Prog. Lang. Syst. 19*, 6 (1997), 942–991.
5. Ghiya, R., Hendren, L. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in c. In *POPL*, 1996.
6. Herlihy, M., Koskinen, E. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Principles and Practices of Parallel Programming (PPoPP)*, 2008.
7. Herlihy, M., Moss, J.E.B. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993).
8. Hudson, B., Miller, G.L., Phillips, T. Sparse parallel Delaunay mesh refinement. In *SPAA* (2007).
9. Jefferson, D.R. Virtual time. *ACM Trans. Prog. Lang. Syst. 7*, 3 (1985), 404–425.
10. Kennedy, K., Allen, J., editors. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2001.
11. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Cascaval, C. How much parallelism is there in irregular applications? In *Principles and Practices of Parallel Programming (PPoPP)*, 2009.
12. Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P. Scheduling strategies for optimistic parallel execution of irregular programs. In *Symposium on Parallel Architectures and Algorithms (SPAA)* (2008).
13. Kulkarni M., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News 36*, 1 (2008), 233–243.
14. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P. Optimistic parallelism requires abstractions. *SIGPLAN Not (Proceedings of PLDI 2007) 42*, 6 (2007), 211–222.
15. Larus, J., Rajwar, R. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.
16. Ni, Y., Menon, V., Adl-Tabatabai, A.-R., Hosking, A.L., Hudson, R., Moss, J.E.B., Saha, B., Shpeisman, T. Open nesting in software transactional memory. In *Principles and Practices of Parallel Programming (PPoPP)*, 2007.
17. Pang-Ning Tan, M.S., Kumar, V., editors. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.
18. Ponnusamy, R., Saltz, J., Choudhary, A. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (1993).
19. Rauchwerger, L., Padua, D.A. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst. 10*, 2 (1999), 160–180.
20. Sagiv, M., Reps, T., Wilhelm, R. 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).
21. Shewchuk, J.R. 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*. May 1996, 203–222.
22. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C. A scalable approach to thread-level speculation. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000).
23. Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., Greenberg, D. Lightcuts: a scalable approach to illumination. *ACM Trans. Graphics (SIGGRAPH) 24*, 3 (July 2005), 1098–1107.
24. Zhan, L.R.Y., Torrellas, J. 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* (1998).

Milind Kulkarni and Keshav Pingali ({milind,pingali}@cs.utexas.edu), University of Texas, Austin.

Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew (bjw@graphics.cornell.edu, {graman,kb,chew}@cs.cornell.edu), Cornell University, Ithaca, NY.