Optimistic Parallelism Benefits from Data Partitioning [*]

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

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

Abstract

Recent studies of irregular applications such as finite-element mesh generators and data-clustering codes have shown that these applications have a generalized data parallelism arising from the use of iterative algorithms that perform computations on elements of worklists. In some irregular applications, the computations on different elements are independent. In other applications, there may be complex patterns of dependences between these computations.

The Galois system was designed to exploit this kind of irregular data parallelism on multicore processors. Its main features are (i) two kinds of set iterators for expressing worklist-based data parallelism, and (ii) a runtime system that performs optimistic parallelization of these iterators, detecting conflicts and rolling back computations as needed. Detection of conflicts and rolling back iterations requires information from class implementors.

In this paper, we introduce mechanisms to improve the execution efficiency of Galois programs: data partitioning, data-centric work assignment, lock coarsening, and over-decomposition. These mechanisms can be used to exploit locality of reference, reduce mis-speculation, and lower synchronization overhead. We also argue that the design of the Galois system permits these mechanisms to be used with relatively little modification to the user code. Finally, we present experimental results that demonstrate the utility of these mechanisms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Languages

Keywords Optimistic Parallelism, Irregular Programs, Data Partitioning, Locality, Lock Coarsening, Over-decomposition

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

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

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

1. Introduction

A pessimist sees the difficulty in every opportunity;
an optimist sees the opportunity in every difficulty.

—*Sir Winston Churchill*

Most multicore applications are irregular programs that manipulate pointer-based data structures like trees and graphs, but little is known about the nature of concurrency in these programs, let alone how to exploit this concurrency effectively on multicore processors.

Recent case studies have shown that irregular programs have a generalized data parallelism that manifests itself as iterative computations over worklists of various kinds [17]. Consider 2-D Delaunay mesh refinement, an important irregular code used in graphics and finite-element solvers. The input to the algorithm is an initial triangulation of a region in the plane, as shown in Figure 1. Some of the triangles in this mesh may be badly shaped (these are shown in black in Figure 1(a)); if so, an iterative refinement procedure, shown in Figure 2, is used to eliminate them from the mesh. In each step, the refinement procedure (i) picks a bad triangle from the worklist, (ii) collects a bunch of triangles in the neighborhood of that bad triangle (called its *cavity*, shown in dark grey in Figure 1(a)), and (iii) re-triangulates that cavity (shown in light grey in Figure 1(b)). If this re-triangulation creates new (smaller) badly-shaped triangles in the cavity, they are added to the worklist. The shape of the final mesh depends on the order in which bad triangles are processed, but it can be shown that every processing order will produce a final mesh without badly shaped elements. From this description, it is clear that bad triangles whose cavities do not overlap can be processed in parallel; moreover, since each bad triangle is processed identically, this is a form of data parallelism. Abstractly, the worklist implements a *set*, and the data parallelism arises from computations performed on each element of that set.

Exploiting this kind of data parallelism in irregular programs can be more complex than exploiting data parallelism in array programs. Data parallelism in array programs usually manifests itself as FOR-ALL loops (that is, FORTRAN-style DO loops over integer intervals in which the iterations can be proven statically to be independent). Data parallelism in irregular programs manifests itself as iteration over sets, but the iterations are not necessarily independent. Although static analysis techniques such as points-to and shape analysis [8, 10, 18, 24] can be used in some cases to prove independence, there may be complex dependences between computations with different set elements, as in Delaunay mesh generation. Static analysis fails to discover the potential data parallelism in these cases.

One solution is to use a BSP-style bulk-synchronous model of computation [30], and execute maximally independent subsets of iterations in each super-step. Recently, Gary Miller *et al.* performed a theoretical study of such an execution scheme for Delaunay mesh refinement [11]. A different solution is to exploit *speculative* or *optimistic* parallelism [17]. In the Galois system, described

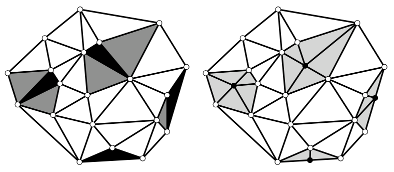

Figure 1. Mesh refinement.

```
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 2. Pseudocode of the mesh refinement algorithm

briefly in Section 2, data parallelism is expressed using set iterators, and these iterators are executed concurrently by some number of threads that pull elements one at a time from the underlying worklist and perform the appropriate computations on that element. Shared-memory is implemented as a collection of concurrent objects. To ensure that the results of the concurrent execution are consistent with the sequential semantics of the program, it is necessary to have a runtime system that detects conflicting method accesses made by different concurrently executing iterations. If a conflict is detected, it is necessary to roll back one of the conflicting iterations and undo any effects it may have had on shared objects. This is accomplished by using commutativity assertions and undo methods specified by the class implementor, as described briefly in Section 2 [17].

We have used the Galois system to successfully parallelize a number of irregular applications, including complex applications that perform refinement and coarsening. However, one of the main lessons from the past twenty years of parallel programming is that exploiting parallelism in a scalable way requires attending to locality. Unfortunately, the current Galois system does not attempt to exploit locality of reference. Sequential implementations of Delaunay mesh refinement often implement the worklist as a stack because doing so improves locality. When a cavity is retriangulated, a number of small bad triangles may be created in the cavity, and working on these bad triangles right away is obviously good for locality. However, using a LIFO worklist in the parallel implementation causes the abort ratio to increase dramatically because it is likely that bad triangles created within some cavity will be scheduled for execution contemporaneously. Therefore, in the current Galois system, a core is given randomly chosen elements from the worklist, which can be good for load balancing and for avoiding speculation conflicts [17] but bad for locality since the core may end up working on bad triangles from all over the mesh. Another problem with the current Galois system is that it does fine-grain synchronization for conflict detection, which can be inefficient.

In this paper, we introduce four interlocking mechanisms for addressing these problems: *data partitioning, data-centric work assignment, lock-coarsening, and over-decomposition*. Partitioning assigns elements of data structures to cores. For example, in Delaunay mesh refinement, the mesh is partitioned by assigning triangles to cores. When a core goes to the worklist to get work, the data-centric work assignment policy ensures that the core is always given a triangle in its partition. If mesh partitions are contiguous regions of the mesh, this work assignment strategy promotes locality.

In the context of optimistic parallelization, this data-centric parallel execution strategy has another significant advantage: the probability of conflicts between concurrent, speculatively executing iterations can be dramatically reduced. In Delaunay mesh refinement, different cores work on different regions of the mesh, and conflicts can happen only when cavities cross partitions, which is rare if partitions are contiguous regions of the mesh.

To reduce overheads further, we also replace fine-grain synchronization on data structure elements with coarser-grain synchronization on data structure partitions. A core can work on its own elements without synchronization with other cores, but when it needs a "foreign" element, it must acquire the lock on the appropriate partition. Therefore, in Delaunay refinement, synchronization is needed only if a cavity crosses partition boundaries.

Finally, to ensure that a core has work to do even if some of its data is locked by other cores, data structures are over-decomposed, that is, we create more data partitions than there are cores so that each core has multiple partitions mapped to it. Thus, even if one or more partitions assigned to a core are locked by other cores, that core may still have work to do.

The rest of this paper is organized as follows. In Section 2, we give a high-level description of the Galois system, which is the basis for the work described in this paper. In Section 3, we describe how the key mechanisms of data partitioning, over-decomposition, and data-centric assignment of work are implemented within the Galois system. In Section 4, we discuss the implementation of lock coarsening. In Section 5, we present experimental results that show the performance improvements from using these mechanisms for four applications: Delaunay mesh refinement, the Boykov-Kolmogorov algorithm (used in image segmentation) [1], a graph-cuts code that uses the preflow-push algorithm [5], and agglomerative clustering [29]. For each application, we describe the algorithm and key data structures as well as opportunities for exploiting parallelism and data partitioning. We conclude in Section 6 with a discussion of related work and future research directions.

2. The Galois system

In this section, we describe the Galois system at a high level to provide background for the rest of the paper. The Galois programming model [17] is a concurrent, object-based shared-memory model that can be implemented on top of an object-oriented language like Java. The design is based on the belief that most programmers should write code with well-understood sequential semantics (we call this *client code*), while the complexity of parallel programming is hidden within library code and the runtime system.

There are three main aspects to the Galois system: (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. Each of these aspects is summarized below. A detailed description can be found in [17].

2.1 Constructs for optimistic parallelism

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

- **Set iterator: for each e in Set S do B(e)**
  The loop body B(e) is executed for each element e of set S. Since set elements are not ordered, this construct asserts that in a serial execution of the loop, the iterations can be executed in any order. There may be dependences between the iterations, as in the case of Delaunay mesh refinement, but any serial order of

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

Figure 3. Delaunay mesh refinement using set iterator

executing iterations is permitted. When an iteration executes, it may add elements to S.

- **Ordered-set iterator: for each e in Poset S do B(e)**
  This construct iterates over a partially-ordered set (Poset) S. It is similar to the Set iterator above, except that any execution order must respect the partial order imposed by the Poset S.

Figure 3 shows client code for Delaunay mesh refinement. The sequential semantics of the set iterators make it easier to write, understand, and debug the client code.

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 and 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.

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 invoke methods of objects in shared memory, so we must ensure that these method invocations are properly coordinated. Section 2.2 describes the information that must be specified by the Galois class writer to enable this coordination. Section 2.3 describes how the Galois runtime system uses this information to ensure that the sequential semantics of iterators are respected.

2.2 Commutativity assertions and undo methods

A simple way to ensure serializability of iteration execution is to implement *strict two-phase locking* [2]. Every object in shared-memory has a lock associated with it, and this lock must be acquired before any of the methods of that object can be invoked. Locks are held until the iteration completes. This ensures serializability, provided deadlocks are handled by some other mechanism.

Unfortunately, two-phase locking is too restrictive for our applications because it can limit parallelism. The iterator that implements the worklist in Delaunay mesh refinement must have some variable that points to the next bad triangle to be handed out. Since this variable is read and written by a thread when it acquires work, it must be guarded by a lock. Two-phase locking requires the thread to hold this lock until the processing of the bad triangle is completed, which prevents other threads from accessing the worklist concurrently.

One solution is to introduce *explicit* concurrency into the programming model and to use notions like open nested transactions to address this problem [21]. As mentioned before, one of the design philosophies of the Galois system is that the client should have a simple *sequential* semantics, so this solution is not appropriate. In Galois, we solve the problem by exploiting the commutativity of method invocations. Intuitively, it is obvious that method invocations to a given object from two iterations can be interleaved without losing serializability provided that these method invocations commute. This ensures that the final result is consistent with some serial order of iteration execution. In the Delaunay example, each iteration removes one element from the current set of bad triangles in the beginning and may add some number of bad triangles at the end. Since set insertions and deletions of distinct elements commute, exploiting commutativity of set operations allows multiple iterations of the set iterator to be executed in parallel without loss of serializability.

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 a 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).

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 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 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 its original state.

Since we are interested in semantic commutativity and undo, it is necessary for the Galois class designer to specify this information.

2.3 Runtime system

The runtime system is responsible for (i) assigning iterations to threads, (ii) detecting conflicts between concurrently executing iterations, (iii) rolling back iterations when conflicts occur and (iv) ensuring that iterations commit in an order that respects the ordering constraints of the iterator [17].

When a thread goes to the runtime system for work, there may be many elements in the worklist of that iterator, so the runtime system needs to implement some policy for determining which element to hand out. The Galois system uses a random assignment policy as the default.

3. Data partitioning and data-centric work assignment

In this section, we describe how data and computations are partitioned between cores to promote inter-core locality. As an extra benefit, this partitioning can also reduce the probability of speculative conflicts. There are a number of requirements that any such scheme should satisfy.

- Some applications may use a mixture of partitioned and non-partitioned data structures, so any scheme for adding partitioned data structures to the Galois system must work smoothly with non-partitioned data structures.
- The programmer must be able to choose whether to partition a data structure or not, and if so, how it should be partitioned. The system should provide default partitioners for important data structure classes but the programmer must be able to override these.
- The client code should change as little as possible when a non-partitioned data structure is replaced with a partitioned data structure (compare this with distributed-memory programming).

Figure 4 illustrates how partitioning works in our implementation. In this figure, the data structure is a regular grid, which is the key data structure used in image segmentation applications such as the Boykov-Kolmogorov code described in Section 5. In our approach, partitioning this grid is done in two stages: the nodes of the grid are mapped to *abstract processors* in an *abstract domain*, and then the abstract domain is mapped to the actual cores. As we discuss in Section 3.1, this two-level partitioning approach has several advantages over the more obvious approach of mapping data structure elements directly to cores. We note that a similar two-level mapping approach is used in HPF [14]. Section 3.2 describes the mapping of data structures to abstract domains. Finally, Section 3.3 describes how the runtime system performs data-centric assignment of work to cores.

3.1 Abstract Domains

The use of abstract domains simplifies the implementation of *over-decomposition*. The basic idea of over-decomposition is to partition data and computation into more partitions than the number of cores in the machine, so that multiple partitions are mapped to each core. For example, in Figure 4, there are four partitions, each of which is mapped to one abstract processor, and each core has two abstract processors mapped to it.

Over-decomposition is the basis for several important mechanisms such as work-stealing and multi-threading. Work-stealing is an implementation of dynamic load-balancing in which idle cores are allowed to steal work from overloaded cores. To promote locality of reference, it is useful to package work together with its associated data, and move both when the work is stolen. Over-decomposition enables this to be implemented as a remapping of abstract processors to cores, which simplifies the implementation. Another use of over-decomposition is multithreading: if the cores support multi-threading, each abstract processor can be executed as a thread on the core it is mapped to, and core utilization may improve. Finally, over-decomposition enables an important optimization in our system called lock coarsening, described in Section 4.

Formally, an abstract domain is simply a set of abstract processors, which may optionally be related by some topology (*e.g.*, a grid or a tree). Abstract domains are implemented as objects in the Galois system, which expose a `distribute` method that takes as an argument the number of cores that the abstract processors should be mapped to. Invoking this method assigns abstract processors to cores.

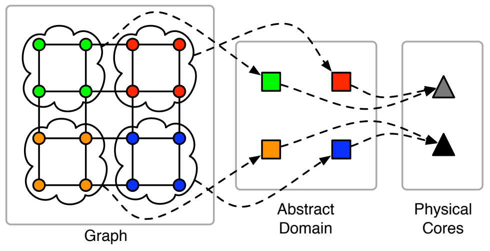

Figure 4. Data partitioning in the Galois system

3.2 Logical and physical partitioning of data structures

In discussing data structure partitioning, it is useful to distinguish between two kinds of data partitioning that we call *logical* partitioning and *physical* partitioning.

In logical partitioning, data structure elements are mapped to abstract processors, but the data structure itself is a single entity that is not partitioned in any way. Logical partitioning can be implemented very simply by using an extra field in each data structure element to record the identity of the abstract processor that owns that element, as is shown graphically in Figure 4.

Logical partitioning is useful for many problems — for example, it can be used to perform data-centric scheduling of iterations in Delaunay mesh refinement. When a core goes to the commit pool to get a bad triangle to work on, the commit pool can examine the worklist of bad triangles and return a bad triangle mapped to that core. If mesh partitions are contiguous regions of the mesh, cores may end up working mostly in their own partitions, improving locality and reducing synchronization. Note that this idea does not require any modification to the client code; only the graph class and the runtime system need to be modified to implement this approach.

Physical partitioning takes this one step further and re-implements each partition as a separate data structure that can be accessed independently of other partitions. The main reason for doing this is to reduce contention for shared data structures. For example, in Delaunay mesh refinement, the worklist of bad triangles is modified by all cores and there can be a lot of contention. If this data structure is partitioned, each core can manipulate its own portion of the global worklist without interference from other cores. Note that while the underlying implementation of the worklist changes, the *interface* to the worklist remains the same. From the perspective of the client code, the worklist is still a single object, and the client code accessing it does not have to change. The "root" of this object is read-only and ends up getting cached at all the cores, reducing contention. Note that physical partitioning in the Galois system is not the same as the data structure partitioning that is performed in distributed memory programming. In the latter case, the data structure is fully partitioned and a processor cannot directly access data assigned to other processors.

The Galois class library provides implementations of common data structures with both logical and physical partitioning. Application programmers can override methods in these classes to modify partitioning algorithms. This is important because it is unlikely that any one partitioning function for an abstract data type is adequate for all applications. Consider, for example, the Graph class. Three of the four applications discussed in Section 5 use graphs, but in the image segmentation applications, the graph is a regular grid, while in Delaunay mesh refinement, the graph is irregular and has no particular structure. Many algorithms have been developed for irregular graph partitioning [13, 15, 28]. One of the simplest approaches for graph bisection is to perform a breadth-first traversal of the graph, starting from some arbitrary node and stopping when half the nodes have been traversed. This process can be applied recursively to partition the mesh further. Kernighan and Lin proposed a local refinement heuristic to reduce the number of cross-partition edges, a useful measure of partition quality in some applications (the set of cross-partition edges is called the *graph separator*) [15]. At the other extreme in complexity are spectral methods that perform eigenvalue computations to determine good graph partitions [28]. However, these partitioning methods are not necessary for regular grids and may even produce poor results compared to a simple block-based partitioning.

At present, the Galois class library provides a simple irregular graph partitioner based on breadth-first graph traversal starting from a boundary node of the graph. It also supports block-block partitioning of two and three-dimensional rectangular grids. These partitioners can be overridden by the application programmer if necessary.

Finally, it may also be useful to cache boundary information for a data structure's partitions. For example, graph nodes that are adjacent to nodes assigned to another core may be labeled as boundary nodes. This exposes some significant optimization opportunities, described in Section 4. This is easily implemented by adding an extra field in each data structure element to record this value, which is set when the data structure is partitioned.

3.3 Computation partitioning

The final step is to ensure that the assignment of work to cores is data-centric. When the Galois system starts up, it spawns a thread for each core. In Java, the virtual machine maps these threads to kernel threads, which the OS is then responsible for mapping to physical cores. Threads spawned by the Galois system rarely sleep, and remain alive until the parallel execution is complete. Hence each thread is effectively "bound" to a specific core. Thus, if data structure elements mapped to a core are only ever touched by the thread mapped to that core, we will achieve significant inter-core locality: very little data will move back and forth between the various cores' caches.

During parallel execution of an iterator, the scheduler in the run-time system assigns work to cores dynamically, but in a partition-sensitive way. If the set being iterated over is not partitioned, the scheduler returns a random element from the current work-list, as in the old Galois system. Otherwise, it returns an element that is mapped to that core. This ensures that worklist elements in a given abstract processor will only be worked on by a single thread. Furthermore, because other data structures in the system may be mapped to the same abstract processor, making the scheduler partition-aware can lead to inter-core locality benefits for other structures as well. For example, in Delaunay mesh generation, this *data-centric scheduling policy* ensures that different cores work on triangles from different partitions of the mesh, reducing data contention and the likelihood of speculation conflicts.

It is not clear that data-centric scheduling is always the best scheduling policy when using partitioned data structures. However, the number of possible scheduling policies is legion, and implementing and evaluating them is beyond the scope of this paper. We leave it to future work.

3.4 Discussion

Some applications (*e.g.*, Delaunay mesh generation) add new elements to data structures during execution, and these elements must be mapped to abstract processors as well. The mutator methods of the data structure (primarily `add` methods) must be modified slightly to handle this. Deciding how this mapping is done is a policy issue, rather than one of correctness. The Galois system's default policy is to map newly added elements to the abstract processor executing the iteration that invoked the mutator method. In Delaunay mesh refinement, this policy means that new triangles created in the cavity of a bad triangle get assigned to the same abstract processor as that bad triangle, which is the right policy. Of course, the application programmer can override the `add` method of the Graph class to change this policy.

3.5 Implementation in the Galois system

Abstract domains are implemented as objects in the Galois system, which expose a `distribute` method, which takes as an argument the number of cores that the abstract processors should be mapped to. Invoking this method performs the distribution of abstract processors to cores.

The implementation of partitioning in the Galois system is straightforward. Data structures that can be logically partitioned implement the `Partitionable` interface, which exposes a method called `partition`. This method accepts as an argument an abstract domain and applies a partitioning function to the data structure, assigning elements of the structure to abstract processors in the specified domain. To change the partitioning function, a programmer simply overrides the `partition` method.

The objects of the data structure that are assigned to abstract processors (such as nodes and edges in a graph) implement the `PartitionObject` interface, which provides simple methods to set and query the abstract processor that the object is assigned to. If boundary information is tracked, objects also implement the `BoundaryObject` interface, which allows the maintenance of this information.

Physically partitioned data structures implement the same interfaces as logically partitioned structures, but also subclass the data structure to provide a partitioned implementation. As mentioned previously, this does not require changing any client code that interacts with the data structure.

Computation partitioning is accomplished purely by a change to the Galois run-time system. When iterating over a partitioned set, the run-time uses partition-aware scheduling rather than random scheduling. This requires no intervention from the programmer (aside from using a partitioned set), and does not change the client code.

4. Lock coarsening: an optimization

A significant source of overhead in the Galois system is the time spent in performing commutativity checks. There are two issues: (i) the code for commutativity checks is complex and (relatively) expensive; and (ii) even if the data structure is partitioned, the conflict logs are not partitioned and thus can become a bottleneck when multiple concurrent iterations access the structure. Data and computation partitioning enable a new optimization that we call *lock coarsening*, which addresses this problem.

4.1 Locks on abstract processors

When a data structure is partitioned, we can often take advantage of the partitioning to replace Galois commutativity checks with two-phase locking based on locking entire partitions. A lock is associated with each abstract processor in the abstract domain. Methods acquire locks on relevant partitions before accessing any elements mapped to these partitions. If any of those locks are already held by other iterations, a conflict is detected and the runtime system rolls back one of the iterations, as before. All locks are held until the iteration completes or aborts.

We implement two optimizations to improve the performance of this basic locking scheme. First, locks on abstract processors are cached by the iteration that holds them. If an iteration accesses multiple elements of a data structure and all of them are mapped to the same abstract processor, the lock on that abstract processor is acquired only once. Furthermore, elements of other data structures that are also mapped to that abstract processor can be accessed without synchronization. We call this optimization *lock caching*.

Second, if boundary information is provided by a data structure, we can elide several of the lock acquires entirely. If an element x accessed by a method is *not* marked as a boundary, the only way it could have been reached is if the iteration had already accessed the abstract processor that element is mapped to. Hence, the iteration does not need not attempt to acquire the lock on that abstract processor. In other words, we need only attempt to acquire locks when accessing boundary objects.

Lock coarsening thus replaces expensive commutativity checks with simple lock acquires and releases, which can dramatically reduce overhead. Furthermore, by using locks to detect conflicts, the burden of conflict checking is no longer centralized in a single conflict log, eliminating a significant concurrency bottleneck. The upshot of lock coarsening, when combined with the two optimizations (lock caching and synchronization on boundaries) is that while an iteration is working on elements mapped to a single abstract processor, no synchronization is required beyond the initial lock acquire. Synchronization instead only occurs when an iteration must cross partition boundaries. In many problems, boundary size grows sublinearly with data structure size (*e.g.*, in a planar graph, boundary size grows as the square root of graph size), and hence synchronization overheads decrease as problem size increases.

4.2 Need for over-decomposition

While lock coarsening can lead to a significant improvement in runtime overheads, it comes at the cost of concurrency. Conceptually, when a thread accesses a partition of a data structure, it "owns" all the elements in that partition, preventing any other thread from accessing them. If a thread crosses partition boundaries and hence must access two partitions, it will own an even greater portion of the data structure.

This problem can be addressed by over-decomposition. Mapping multiple abstract processors to a core makes it more likely that a thread can continue to do useful work even if one or more of its abstract processors are locked by threads executing other iterations.

We do not yet have a good understanding of how much over-decomposition is appropriate. Beyond some level of over-decomposition, conflicts become sufficiently rare that further over-decomposition will not improve performance. In fact, excessive over-decomposition may reduce performance. A simple *reductio ad absurdum* shows this to be the case: if we overdecompose until there is a only single element mapped to each abstract processor, we will essentially be performing fine-grained locking. While this will minimize conflicts, it will result in many more synchronization operations because each new object accessed will require that a new lock be acquired, leading to higher overhead. We leave the subject of determining the right level of over-decomposition for future work.

4.3 Implementation in the Galois system

Over-decomposition is trivially implemented by using abstract domains with more abstract processors than physical cores in the system.

Commutativity checks in the Galois system are implemented by wrapping shared objects in *Galois wrappers*. The wrapper contains the conflict log for the wrapped object and performs commutativity checks when a method is invoked. If the check is successful, the appropriate method of the wrapped object is called. Because lock coarsening is a replacement for commutativity checks, it is implemented by providing a second Galois wrapper for a data structure. Rather than performing commutativity checks, the new wrapper uses the lock coarsening approach for conflict detection. Because both the old and new Galois wrappers provide the same interface, the client code is agnostic to which form of conflict detection is being used and does not need to change.

5. Experimental results

We evaluated our approach on four applications from the graphics domain. Although some regular graphics applications are streaming applications that can be executed efficiently on GPUs, the applications we consider in this section are very irregular, and we believe they are better suited for execution on multicore processors than on GPUs.

The machine we used in our studies is a dual-processor, dual-core 3.0 GHz Xeon system with 16KB of L1 cache per core and 4MB of L2 cache per processor. In our initial experiments, we found performance anomalies arising from automatic power management within the processor. At the suggestion of researchers at Intel, we down-clocked the processor to 2.0 GHz, which eliminated the performance anomalies.

We implemented the Galois system, with the enhancements discussed in this paper, in Java 1.6. Given the relatively small number of cores, we found there was no need for multi-threading or work stealing in our applications, so we did not evaluate these mechanisms. They are likely to be more important on larger numbers of cores. To take into account variations in parallel execution as well as the overhead of JIT compilation, each experiment was run 5 times under a single JVM instance, and the fastest execution time was recorded. Garbage collection can also have a significant impact on performance; to reduce its effects, a full GC was performed before each execution. We used a 2GB heap.

5.1 Delaunay mesh refinement

This application is the running example used in this paper, and it is described briefly in Section 1. Pseudocode is shown in Figure 2.

Opportunities for exploiting parallelism. The natural unit of work for parallel execution is the processing of a bad triangle. 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 and therefore can be processed concurrently.

Partitioning strategy Meshes are usually represented as graphs in which nodes represent mesh triangles and edges represent adjacency of triangles in the mesh. Partitioning the nodes of this graph creates a partition of mesh triangles. The Galois Graph class uses an adjacency list representation of graphs. A partitioner based on a breadth-first walk of the graph is provided in this class, as described in Section 3.

Experiments We implemented and evaluated 5 different versions of the Delaunay benchmark:

- $meshgen_{seq}$ – this is a sequential implementation of Delaunay mesh refinement. It contains no threading or synchronization.
- $meshgen_{gal}$ – a Galois version of the benchmark that employs the original Galois model. It uses the unordered set Galois iterator, and commutativity checks to detect conflicts.
- $meshgen_{par}$ – a version that partitions the worklist and the graph. It uses commutativity checks for conflict detection, but uses partition-aware scheduling as discussed in Section 3.
- $meshgen_{lco}$ – a version that implements lock coarsening as well as partitioning.
- $meshgen_{ovd}$ – a version that implements partitioning, lock coarsening and over-decomposition. This version overdecomposes by a factor of 4 (*i.e.*, four partitions per core)

In all these versions, the worklist is implemented as a stack to promote locality (when the worklist is partitioned, each partition is a stack). For $meshgen_{gal}$ and $meshgen_{par}$, the code for commutativity checks was written by hand. The input data was generated using Jonathan Shewchuck's Triangle program [27]. It had 100,364 triangles and boundary segments, of which 47,768 were bad.

Table 1 shows the wallclock time (in seconds) for the 5 benchmarks. Figure 5 shows the speedup of the four parallel benchmarks, relative to the running time of the best sequential version $meshgen_{seq}$. We see that $meshgen_{gal}$, the version that uses the original Galois system achieves a speedup of only 1.2 on 4 cores[1]. $meshgen_{ovd}$, the version that combines partitioning, lock-coarsening and over-decomposition, achieves the best speedup of 3.26 on 4 cores.

To understand the performance of the different versions, it is useful to consider first the running times of these versions on a single core (shown in the first column of Table 1). Table 2 presents the same data and shows the overheads as a percentage of the execution time of $meshgen_{seq}$. The overheads for $meshgen_{gal}$ and $meshgen_{par}$ are high because they perform full commutativity checks to detect conflicts when running in parallel. These are precise but expensive checks. On the other hand, both $meshgen_{lco}$ and $meshgen_{ovd}$ use locks on partitions to perform conflict detection. These are less precise but also significantly less expensive, as the overheads show.

Benchmark	1 core	2 cores	4 cores
$meshgen_{seq}$	11.316	—	—
$meshgen_{gal}$	13.956	9.935	9.433
$meshgen_{par}$	13.865	7.510	5.315
$meshgen_{lco}$	11.924	6.629	3.925
$meshgen_{ovd}$	11.437	6.186	3.474

Table 1. Execution time (in seconds) for Delaunay mesh refinement.

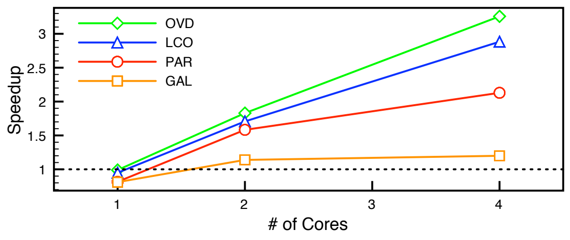

Figure 5. Speedup vs. # of cores for Delaunay mesh refinement

Benchmark	Overhead	Abort Ratio (4 cores)
$meshgen_{gal}$	23.33%	85.22%
$meshgen_{par}$	22.53%	0%
$meshgen_{lco}$	5.37%	56.47%
$meshgen_{ovd}$	1.07%	7.08%

Table 2. Uniprocessor overheads and abort ratios

Another important factor in overall performance is the abort ratio (*i.e.*, the ratio of aborted iterations to completed iterations, expressed as a percentage). A high abort ratio indicates significant contention in the program, which may reduce performance. However, not all aborts are equally expensive since iterations that abort soon after starting do not contribute as much to the overhead as iterations that abort close to the end do. Therefore, a high abort ratio does not necessarily correlate to poor performance.

Table 2 shows the abort ratio for each of the parallel implementations when run on 4 cores. $meshgen_{gal}$ has a very high abort ratio. This is because the worklist is implemented as a stack, which leads to high abort ratios for this application, as mentioned in Section 1. When a cavity is re-triangulated, a number of bad triangles may be created in the interior of the cavity. If the worklist is a stack, all these bad triangles are adjacent to each other in the worklist, and it is likely that they will be refined contemporaneously, leading to conflicts. We experimented with a different scheduling policy for $meshgen_{gal}$, selecting triangles at random from the worklist. This dropped the abort ratio to zero, but the loss of locality attenuated the benefits from concurrency. In spite of having uniprocessor overhead similar to that of $meshgen_{gal}$, $meshgen_{par}$ performs much better because it has a very low abort ratio.

However, the abort ratio does not tell the full story, as $meshgen_{lco}$ outperforms $meshgen_{par}$, achieving a speedup of 2.88 on 4 cores. This version of the benchmark performs better for two reasons: (i) lower overheads due to much simpler conflict checks and (ii) the elimination of Galois conflict logs as a bottleneck, improving concurrency. Thus we see that $meshgen_{lco}$ is not only faster than $meshgen_{par}$ but also scales better. Interestingly, the fairly high abort rate does not hurt this implementation much. This is because the lock-coarsened conflict detection triggers aborts at the very beginning of an iteration, and most of the aborts are due to busy waiting. Furthermore, because the aborted iteration is immediately retried, the abort ratio is misleadingly high. These results suggest that some kind of exponential back-off scheme may be appropriate to reduce the abort ratio, although it is not clear that there will be commensurate improvements in performance.

Finally, the over-decomposed version $meshgen_{ovd}$ combines the benefits of coarse-grain locking with a low abort ratio. Its abort ratio is higher than that of $meshgen_{par}$ because it is performing coarser-grain locking, but its synchronization overhead is lower for the same reason. Since a core has other partitions to work on if one of its partitions is locked by another core, it does not keep trying to reacquire the lock on its partition, and the abort ratio is lower than it is for $meshgen_{lco}$. It achieves a speedup of 3.26 on 4 cores, and thus has the best absolute performance as well as the best scalability.

5.2 Image segmentation using the Boykov-Kolmogorov algorithm

The Boykov-Kolmogorov algorithm is a maxflow algorithm used in image segmentation problems [1] (abbreviated from here on as "B-K algorithm"). Like the standard augmenting paths algorithm [4], it performs a breadth-first walk over the graph to find paths from the source to the sink in the residual graph. However, once an augmenting path has been found and the flow is updated, the current search tree is updated to reflect the new flow, and then used as a starting point for computing the next search tree. In addition, the algorithm computes search trees starting from both the source and the sink. Experiments show that on uniprocessors, the B-K algorithm outperforms other maxflow algorithms for graphs arising from image segmentation problems [1].

The B-K algorithm is naturally a worklist-style algorithm: each node at the frontier of a search tree is on the worklist. When a node is removed from the worklist, its edges are traversed to extend the search, and newly discovered nodes are added to the worklist. If an augmenting path is found, the capacities of all edges along the path are decremented appropriately. Nodes that are disconnected as a result of this augmentation are added back to the worklist. The pseudocode for this algorithm is given in Figure 6. For lack of space, only the code for extending the search tree rooted at the source is shown; the code for extending the search tree rooted at the sink is similar.

Opportunities for exploiting parallelism As in the other applications, the order in which elements are processed from the worklist is irrelevant to proper execution, although different orders will

[1] In PLDI 2007, we presented results for a Galois version of Delaunay mesh refinement, but the worklist used in those experiments implemented random choice, not LIFO order [17].

```
1: worklist.add(SOURCE);
2: worklist.add(SINK);
3: for each Node n in worklist {
     //n in SourceTree or SourceTree
4:   if (n.inSourceTree()) {
5:     for each Node a in n.neighbors() {
6:       if (a.inSourceTree())
7:         continue; //already found
8:       else if (a.inSinkTree()) {
           //decrement capacity along path
9:         int cap = augment(n, a);
           //update total flow
10:        flow.inc(cap);
           //put disconnected nodes onto worklist
11:        processOrphans();
12:      } else {
13:        worklist.add(a);
14:        a.setParent(n); //put a into SourceTree
15:      }
16:    }
17:  } else { //n must be in the SinkTree
18:    ... //similar to code for when n in Source Tree
19:  }
20:}
```

Figure 6. Pseudocode for Boykov-Kolmogorov algorithm

produce different search trees. Therefore, we can process nodes in the worklist concurrently, provided there are no conflicts. There are two sources of potential conflicts: (i) concurrent traversals that grab the same node for inclusion in the tree (so two threads try to set the parent field of the same node concurrently (line 14)), and (ii) augmenting paths that have one or more edges in common (line 9). Whether or not these potential conflicts manifest themselves as actual conflicts at runtime depends on the structure of the graph and the evolution of the computation, so optimistic parallelization seems appropriate.

Partitioning strategy The Boykov and Kolmogorov algorithm works for arbitrary graphs, but it is intended to be used for maxflow problems that arise in image segmentation. Graphs arising in this application have a regular grid structure, which can be partitioned into rectangular blocks trivially. Moreover, the structure of the graph does not change during execution (only the capacities of edges are modified). Therefore, the partitioning can be done once at the beginning, and no effort is needed to maintain appropriate boundary information in the graph. Note that the `flow` variable cannot be partitioned.

Experiments We ported a C implementation of Boykov and Kolmogorov's augmenting paths algorithm to Java and used it to create 5 different versions of the benchmark: $paths_{seq}$, $paths_{gal}$, $paths_{par}$, $paths_{lco}$ and $paths_{ovd}$. In all versions, the worklist is implemented as a queue, matching the C implementation. The input data is a 1024x1024 grid representing a checkerboard pattern. Table 3 shows the wallclock time of the 5 benchmarks. Figure 7 shows speed-ups relative to the sequential version. Table 4 shows the uniprocessor overheads and abort ratios of the four parallel versions on 4 cores.

Benchmark	1 core	2 cores	4 cores
$paths_{seq}$	384	—	—
$paths_{gal}$	1200	1822	1779
$paths_{par}$	1203	738	463
$paths_{lco}$	458	423	279
$paths_{ovd}$	459	253	155

Table 3. Execution time (in milliseconds) for B-K maxflow.

We note that $paths_{gal}$ actually slows down when run on multiple cores. This is due to the nature of the algorithm: much of the work in an iteration is simply adding and removing elements from the worklist. However, when dealing with non-partitioned data structures, these operations must be synchronized. Even though the

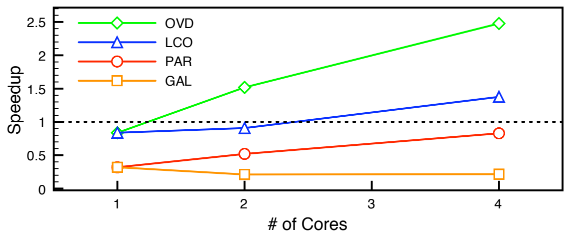

Figure 7. Speedup vs. # of cores for B-K maxflow

Benchmark	Overhead	Abort ratio
$paths_{gal}$	212.5%	16.68%
$paths_{par}$	213.3%	0%
$paths_{lco}$	19.27%	55.88%
$paths_{ovd}$	18.53%	0.04%

Table 4. Uniprocessor overheads and abort ratios

data structure used is the highly efficient ConcurrentLinkedQueue from Java 1.6, this is sufficient to slow down $paths_{gal}$. Furthermore, the queue implementation of the worklist leads to poor locality. Multiple cores are often manipulating the same region of the graph, leading to contention for data. This also manifests itself in a fairly high abort rate despite the fine-grained contention management afforded by Galois, leading to further performance degradation.

Once we begin partitioning the data structures, these bottlenecks disappear. There is no longer contention for the worklist, and cores are largely confined to disjoint regions of the graph, as can be seen from the negligible abort ratio. We thus begin to see performance improvements as the number of cores increases. However, in $paths_{par}$, the Galois overhead overwhelms this speedup and the benchmark on 4 cores is still slower than the sequential code. We see the effects of eliminating this overhead when moving to $paths_{lco}$, which, on four cores, beats the sequential code, running 38% faster. However, the high abort rates, as seen in Table 4, keep this implementation from scaling (as in Section 5.1, the abort rate reflects busy-waiting). With the addition of over-decomposition in $paths_{ovd}$ (this time by a factor of 16), the abort ratio once again becomes negligible. Thus, $paths_{ovd}$ has low overhead and scales, executing 2.48 times faster on four cores than the sequential code.

5.3 Image segmentation using preflow-push

Although experiments on uniprocessors have shown that the Boykov-Kolmogorov algorithm outperforms other maxflow algorithms for graphs arising from image segmentation problems [1], it is not known whether this holds for parallel implementations. Therefore, we also implemented the Goldberg-Tarjan preflow-push algorithm [5], which is known to perform well on general graphs both in an asymptotic sense and in practice. The word "preflow" refers to the fact that nodes are allowed to have excess flow at intermediary stages of the algorithm, unlike the augmenting paths algorithm, which maintains a valid flow at all times.

The basic idea is to maintain a height value at each node that represents a lower bound on the distance to the sink node. The algorithm begins with $h(t) = 0$ and $h(s) = |V|$, the number of vertices in the graph, where s is the source and t is the sink. First, every edge exiting the source is saturated with flow, which deposits excess at all of the source's neighbors. Any node with excess flow is called an *active* node. Then, the algorithm performs two operations, *push* and *relabel*, on the active nodes. The push operation takes excess flow at a node and attempts to move as much as possible to a neighboring node, provided the edge between them still has capacity and the height difference is 1. The relabel operation raises a node's height so that it is at least high enough to push flow to one of its neighbors. Forcing flow to move in height steps of 1 makes it impossible for a node at height $|V|$ to ever reach the sink. Therefore, this phase of the computation terminates when the height of all active nodes is $|V|$, signifying that all possible flow has reached the sink. Finally, the remaining excess is drained back to the source. This is typically very fast and can be done in a variety of ways (we do it by running preflow-push a second time).

Opportunities for parallelism Preflow-push is also a worklist algorithm since all active nodes can be placed on a worklist and processed in any order. Since the operations on a node are purely local in nature, nodes can be operated on in parallel provided they are not adjacent to each other.

Partitioning For image processing applications, input graphs typically have a grid-like structure. Therefore, as in the B-K algorithm, we can trivially partition the grid into rectangular blocks.

Experiments We wrote a Java implementation of preflow push and used that as a base to generate five versions of the benchmark, along the same lines as the other benchmarks: prf_{seq}, prf_{gal}, prf_{par}, prf_{lco} and prf_{ovd}. We evaluated these five implementations on a 128x128 graphcuts instance. Table 5 gives wallclock execution times for the five benchmark versions (in seconds), while Figure 8 shows speedups over the sequential code. Table 6 gives the overheads for the four parallel versions running on a single core, and the abort ratios on four cores.

Benchmark	1 core	2 cores	4 cores
prf_{seq}	4.93	—	—
prf_{gal}	5.68	3.06	6.09
prf_{par}	5.68	2.96	2.26
prf_{lco}	5.44	2.83	2.24
prf_{ovd}	5.29	2.77	1.97

Table 5. Execution time (in seconds) for preflow push.

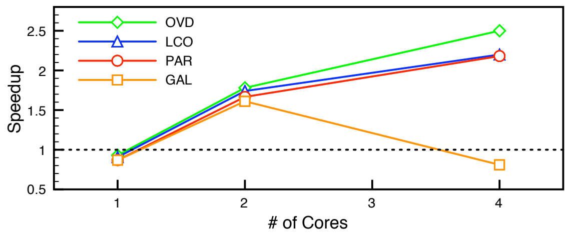

Figure 8. Speedup vs. # of cores for preflow push

Benchmark	Overhead	Abort ratio (4 cores)
prf_{gal}	15.2%	83.99%
prf_{par}	15.2%	0.02%
prf_{lco}	10.3%	43.46%
prf_{ovd}	7.30%	10.31%

Table 6. Uniprocessor overheads and abort ratios

We see that the overheads are reasonable for all four versions of the benchmark, but that the lock-coarsened versions are slightly better than the standard Galois versions. This suggests that commutativity checks are a small portion of the overhead in this application. In fact, most of the overhead in this benchmark comes from accesses to the worklist.

Table 6 shows the abort ratios for the four parallel versions of preflow push. As expected, prf_{gal} has a high abort ratio, as the scheduling is not partition aware. Similarly, we note very high abort ratios for prf_{lco}, as the iterations of preflow push often cross partition boundaries and thus lead to many aborts without over-decomposition.

These abort ratios and overheads are reflected in the actual performance, shown in Figure 8. We see that prf_{gal} slows down when run on four cores. This is due largely to contention for the worklist. This bottleneck is removed in prf_{par}, which achieves a speedup of 2.26 over sequential on four cores. Lock coarsening, as expected, does not provide a benefit, due to the very high abort ratios, and prf_{lco} performs no better than prf_{par}. Over-decomposition is able to reduce contention significantly, while still providing overhead benefits. Thus, prf_{ovd} performs the best of all the parallel versions, achieving a speedup of 2.50 over sequential execution on four cores.

5.4 Agglomerative clustering

The final application is *agglomerative clustering*, a well-known data-mining algorithm [29]. This algorithm is used in graphics applications for handling large numbers of light sources [31]. 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.

Agglomerative clustering can be performed by an iterative algorithm: at each step, two points are examined. If each point agrees that the other is its nearest neighbor, the two points are clustered together and replaced by a single new point that represents the new cluster. The location of this new point may be determined heuristically [29]. The algorithm terminates when there is only one point left in the data set[2]. We accelerate the computation of a point's nearest neighbor by utilizing a hierarchical spatial decomposition structure called a *kd-tree*, which is similar to an oct-tree.

Opportunities for exploiting parallelism In this application, iterations are largely independent, as long as they access different points. A second source of conflict arises in the calculation of the nearest neighbor: one iteration may insert a new point that would change the result of the nearest neighbor computation of another iteration, breaking sequential semantics. Dealing with this scenario requires a complex commutativity condition. In the absence of these conflicts, the iterations can easily be executed in parallel.

Partitioning We would like to partition the points in the input set spatially. This can be easily accomplished as the kd-tree already captures a spatial partitioning of points. Furthermore, the natural partitioning of the kd-tree allows it to be easily physically partitioned.

Experiments We modified the Java implementation of agglomerative clustering used in [31] to use Galois iterators and commutativity checks. We generated three versions of the benchmark along the same lines as the other applications: $cluster_{seq}$, $cluster_{gal}$ and $cluster_{par}$. Due to the complex nature of the commutativity checks in this application, we could not perform the lock coarsening optimization. We evaluated these three implementations on an input set containing 20,000 points. Table 7 gives wallclock execution times for the three benchmark versions (in seconds), while Figure 9 shows speedups over the sequential code. Table 8 gives the overheads for the two parallel versions running on a single core and the abort ratios on four cores.

Here we again see the efficacy of partitioning: $cluster_{par}$ outperforms $cluster_{gal}$, achieving a speedup of nearly 2 on four

[2] Note that this is a variant of the clustering algorithm presented in [17]. Due to the nature of the "distance" metric used in clustering, the unordered algorithm presented here produces the same result as the ordered variant given in [17]

Benchmark	1 core	2 cores	4 cores
$cluster_{seq}$	5.62	—	—
$cluster_{gal}$	6.19	3.83	3.51
$cluster_{par}$	6.21	3.54	2.94

Table 7. Execution time (in seconds) for agglomerative clustering.

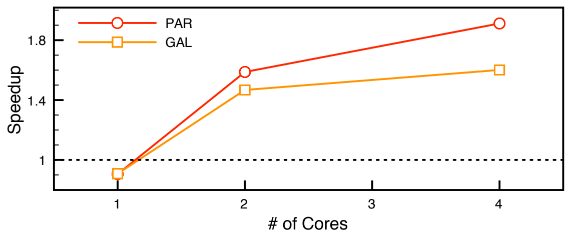

Figure 9. Speedup vs. # of cores for agglomerative clustering

Benchmark	Overhead	Abort ratio (4 cores)
$cluster_{gal}$	10.1%	1.47%
$cluster_{par}$	10.5%	0.13%

Table 8. Uniprocessor overheads and abort ratios

processors over $cluster_{seq}$. The improvement of $cluster_{par}$ over $cluster_{gal}$ is partially attributable to a lower abort ratio, but as the abort ratios for both versions are low, we believe most of the improvement is due to better locality, especially in the kd-tree, which is traversed multiple times in each iteration.

The overhead of both parallel versions is low, suggesting that lock coarsening is not necessary to lower overheads. However, we see the deleterious effects of the centralized conflict log used for the commutativity checks; $cluster_{par}$ does not significantly outperform the sequential version. The low abort ratio indicates that the problem is not due to mis-speculation. Rather, the fact that most of the speedup is achieved by the time $cluster_{par}$ is run on two processors points to contention for the conflict log as the bottleneck. This pattern is exhibited by the *par* versions of the other benchmarks, as well, pointing to a significant optimization opportunity: improving the concurrency of commutativity checks. We leave this to future work.

6. Related work and conclusions

Data and computation partitioning were explored by HPF and related efforts [14, 23]. However, the focus there was on data-parallel array programs on distributed-memory computers, so problems like data-centric dynamic computation partitioning, lock coarsening, interactions with speculation conflicts, etc. did not arise. Müller and Rühl proposed an extension to High Performance Fortran that allowed for its alignment and distribution idioms to be applied to irregular structures but their focus was on sparse arrays [20]. Recent work by Gordon *et al.* has focused on exploiting data and task parallelism in *stream programs* [6]. This approach seems well-suited for signal processing programs, but not for the kinds of applications considered in this paper.

When considering optimistic execution of a parallel program, a commonly proposed mechanism is *Transactional Memory* (TM), with many implementations both in hardware [9, 19] and software [7, 25]. We distinguish between *optimistic synchronization*, where an existing parallel program uses optimistic techniques for synchronization, and *optimistic parallelization*, which is a model for parallelizing sequential programs. While we feel that TM is well-suited to the former, the role of TM in optimistic parallelization is more limited. In particular, TM is not concerned with locality issues, which is one of the main concerns in this paper.

One common approach that does use optimistic parallelization is *Thread Level Speculation* (TLS) [16, 22]. Like most TLS systems, Galois looks to loops for parallelization opportunities (although parallel iterators allow the Galois system to handle more general loops). The substantive difference between TLS systems and Galois is that we take advantage of data structure semantics when determining whether speculative parallel execution is incorrect. In contrast, current TLS systems examine read/write sets of speculative computations, which is far more restrictive. The extensions we propose in this paper rely on data structure partitioning, and they require a runtime system that has the freedom to change the schedule of parallel execution as well as use data-structure specific information for conflict detection. TLS systems are bound to one specific execution schedule (loop order), and one conflict detection scheme (read/write sets) and hence cannot leverage data-structure partitioning effectively.

Recently, Michael Scott *et al.* have studied the use of partitioning in optimistic parallel execution of Delaunay mesh *generation* [26]. Their partitioned code is written manually, and the approach is customized for Delaunay mesh generation. In contrast, our approach uses general-purpose mechanisms implemented within the Galois system, so it is not customized to a particular problem. In addition, they use transactional memory for synchronization between cores, so they do not use locks or lock coarsening.

In the context of task-parallelism, Chen *et al.* [3] schedule threads on CMPs to promote cache-sharing: threads that access similar portions of data should use the same cache. They apply a scheduling heuristic to promote this behavior. Our scheduling is informed by the data partitioning, rather than based on a heuristic, and not only promotes locality in a single core but reduces contention across cores.

Intel has recently released its Thread Building Blocks (TBB) [12], which provide a programming model and toolkit for parallelizing programs. The toolkit supports the partitioning of work but only for structured loops such as `for` loops where the iteration space is defined before the loop begins. They provide a parallel `while` construct similar to Galois' unordered Set iterator. However, they provide no support for work partitioning or scheduling to promote locality.

6.1 Conclusions

The goal of the Galois system is to make it easier to exploit data parallelism in irregular programs. The data parallelism arises from the use of iterative algorithms organized around worklists of various kinds. In our earlier work, we introduced set iterators to express this parallelism and showed how this parallelism can be exploited on multicore machines.

In this paper, we described how data partitioning can be exploited by Galois programs. The key is to perform a logical partitioning of data structures and to assign work to cores in a data-centric way so as to promote locality. In addition, fine-grain synchronization on data structure elements is replaced with coarse-grain synchronization on data partitions, thus reducing the cost of conflict detection. Finally, over-decomposition is used to improve core utilization. We found, across several important benchmarks, that this approach is practical and is successful in exploiting both parallelism and inter-core locality of reference, while keeping parallel overheads low.

The extended Galois system presented in this paper is the first optimistic parallelization system that uses (i) partitioning of irregular data structures and (ii) data-centric work assignment to promote locality, reduce mis-speculation and lower parallel overheads, enabling the exploitation of parallelism in a scalable way.

Acknowledgments

We would like to thank Dimitrios Prountzos for his work on porting the Galois run-time from C++ to Java. We would also like to thank our anonymous reviewers for their helpful comments.

References

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

[2] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. *A history and evaluation of system R*, pages 54–68. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

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

[4] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

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

[6] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM Press.

[7] 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, New York, NY, USA, 2003.

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

[9] 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*, 1993.

[10] S. Horowitz, P. Pfieffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[11] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse parallel delaunay mesh refinement. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 339–347, New York, NY, USA, 2007. ACM Press.

[12] Intel Corporation. Intel thread building blocks 2.0. `http://osstbb.intel.com`.

[13] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

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

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

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

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

[18] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31, New York, NY, USA, 1988. ACM Press.

[19] 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.

[20] Andreas Müller and Roland Rühl. Extending high performance fortran for the support of unstructured computations. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 127–136, New York, NY, USA, 1995. ACM Press.

[21] 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.

[22] 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.

[23] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM Symposium on Programming Language Design and Implementation*, pages 69–80, 1989.

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

[25] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[26] Michael Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, Boston, MA, September 2007.

[27] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1996.

[28] A. Sohn and H. D. Simon. S-HARP: A parallel dynamic spectral partitioner. *Lecture Notes in Computer Science*, 1457:376–385, 1998.

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

[30] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[31] 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.