THE GALOIS SYSTEM: OPTIMISTIC PARALLELIZATION OF IRREGULAR PROGRAMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Milind Vidyadhar Kulkarni

August 2008

THE GALOIS SYSTEM: OPTIMISTIC PARALLELIZATION OF IRREGULAR PROGRAMS

Milind Vidyadhar Kulkarni, Ph.D.

Cornell University 2008

The last several years have seen multicore architectures become ascendant in the computing world. As a result, it is no longer sufficient to rely on increasing single-threaded performance to improve application performance; instead, programmers must turn to parallelization to realize the performance gains of multicore architectures. While much research over the past three decades have focused on parallelizing *regular* programs which operate over arrays and matrices, much less effort has been focused on *irregular* programs which operate over pointer-based data structures such as trees and graphs. In fact, it is not even clear that a significant amount of parallelism even exists in these applications.

We identify a common type of parallelism that arises in irregular programs that operate over worklists of various kinds, which we call *amorphous data-parallelism*. Due to the data-dependent nature of these applications, static compiler analysis does not suffice to uncover any parallelism. Instead, successful parallelization requires speculative, or *optimistic*, parallelization. However, existing speculation techniques, such as thread-level speculation, are too low-level to recognize and extract useful parallelism from these applications.

We present the *Galois system* for optimistic parallelization which uses high-level abstractions to express amorphous data-parallelism in irregular programs, and uses semantic properties of data structures to automatically parallelize such programs. These abstractions allow programs with amorphous data-parallelism

to be written in a sequential manner, relying on run-time support to extract parallelism.

We then develop abstractions which allow programmers to succinctly capture locality properties of irregular data structures. We show how these abstractions can be used to improve locality, improve speculation performance and reduce speculation overhead. We also present a novel parallel scheduling framework which allows programmers to leverage algorithm semantics to intelligently schedule concurrent computation, improving performance.

We demonstrate the utility of the Galois approach, as well as the extensions that we propose, across a wide range of irregular applications demonstrating amorphous data-parallelism. We find that the Galois approach can be used to extract significant parallelism with low programmer overhead.

BIOGRAPHICAL SKETCH

Milind Kulkarni was born in Chapel Hill, North Carolina on August 19th, 1982 but soon moved to Durham, North Carolina. Living in Durham proved beneficial, as it meant he could attend the North Carolina School of Science and Mathematics while still being able to come home every weekend for home cooking. He graduated from NCSSM in 1998.

As a North Carolinian born-and-bred, Milind had to pick sides among the great college basketball powerhouses of Tobacco Road. Rather than taking the easy way out, he found himself at North Carolina State University, where he made lifelong friends and discovered the joys and sorrows of rooting for a perennial underdog (Wolf! Pack!). He graduated from NC State in 2002 with degrees in both Computer Science and Computer Engineering, and a plan to pursue a Ph.D. in "either Artificial Intelligence or Compilers." Soon after joining the Department of Computer Science at Cornell University, he chose the latter.

He spent four years in upstate New York, learning to love snow and long winters, and then jumped at the opportunity to move to Austin, Texas, where there is no snow and "winter" means occasionally wearing a long-sleeve shirt. He has lived in "the ATX" for the past two years.

To my parents

ACKNOWLEDGEMENTS

Pursuing a Ph.D. is a long and often arduous process. The list of people who helped me along the way is equally long, and attempting to acknowledge each of them and thank them appropriately is similarly arduous. Because I have no hope of adequately expressing my gratitude to all those who got me to this point, I must instead list a few and hope that others not recognized understand that the omission is one of forgetfulness and oversight, not of intent.

I must first, of course, thank my parents, Radhika and Vidyadhar Kulkarni. They have always been supportive and encouraging, not only by word and deed but by example. Both of them have Ph.D.s, which made it clear to me that in order to have the kind of life that I wanted, it was necessary that I, too, obtain a Ph.D. Their memories of pursuing a Ph.D. served me in good stead as I encountered the ups and downs common to any grad school experience. I must also thank my brothers, Ashwin and Arvind, as they made coming home to decompress from work a worthwhile experience.

When I first arrived at Cornell, I was assigned an office randomly. As luck would have it, a few of my first year officemates became my closest friends through grad school. For shared experiences, from adjusting to life as a graduate student, to movie nights, to heading off to Montreal in the summer on a whim, I must thank Kamal and Ganesh sincerely. I also must acknowledge friends, in Ithaca and Austin, and old friends from home, who kept me sane and happy: Vinney, Chethan, Siggi, Nick, Mike, Ben and many others.

My group-mates over the years provided invaluable advice to me, from how to manage time, to how to balance the demands of graduate school and one's social life, to how to best catch my advisor for meetings. Most importantly, they provided a productive and enjoyable work environment. The cast of characters

is continually changing, but I thank them all: Greg, Dan, Kamen and Rohit at Cornell, and Patrick, Martin and Dimitrios at UT.

I have the utmost gratitude to my Special Committee members, Kavita Bala, José Martínez and Radu Rugina, who taught me, read revisions of this document, collaborated on papers and provided advice. Without their help, this work would be less advanced and presented far more crudely. And, of course, I must thank my advisor, Keshav Pingali, who has molded my academic career, influenced my perspectives on research, provided ideas and in general been a mentor to me.

Finally, I would like to thank Monique Shah. Her support and companionship have been the foundation that I needed to finish my research, begin writing this thesis and finally complete my Ph.D.

This research has been supported by the US Department of Energy High-Performance Computer Science Fellowship program, administered by the Krell Institute.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The need for parallel programs

Since the advent of the microprocessor, computer performance has been dominated by *Moore's Law* [85]: the number of transistors on a single chip grows at an exponential rate over time. This "law" has held for the past half-century, and may well continue to hold in the future. However, a classic misstatement of Moore's Law holds that the *performance* of processors will grow exponentially, and, indeed, double every 18 months. In other words, a given program would run twice as fast on a current processor than it would on a year-and-a-half-old processor. It is this misstated variant law that has held currency in the popular consciousness, and striving to meet its dictates has driven much of the architecture research of the past several decades. Unlike the original Moore's Law, the future relevance of this misstated law is still very much in question.

Over the past several decades, architects have successfully translated the increased transistor density predicted by Moore's Law into the higher performance expected by the misstated law. Until recently, processor speeds (as a proxy for processor performance) have adhered to this exponential growth curve [99]. Unfortunately, this improved performance came at the cost of increased power demands, as power consumption is proportional to frequency [111]. As a result of this "power wall," the frequency of processors has leveled off in recent years; hence, uniprocessor performance no longer grows at its historical rate.

Because increasing transistor density can no longer translate into increased uniprocessor performance, architects have turned to *multicore* processors [95].

Rather than a single processor, CPUs now contain multiple processors on a single chip. Furthermore, the number of *cores* on a chip is expected to increase exponentially, to continue tracking with Moore's Law. While multicore architectures successfully utilize the extra transistors at hand, the consequence of this paradigm shift is that what was previously a *hardware* problem has instead become a *software* problem.

In years past, programmers could count on increasing uniprocessor speeds to improve the performance of their programs. In other words, the problem of improving performance was solved by hardware. However, now that uniprocessor speed increases have stalled, programmers can no longer rely on hardware to deliver the expected performance improvements. Instead, *software* must improve to utilize the parallel processing capabilities afforded by multicore architectures.

To take advantage of multicore processors, software must be multithreaded. Unfortunately, writing parallel programs is significantly more difficult than writing sequential code. In the past, the high barrier to entry of parallel programming was not an issue. The only demand for parallel programs was in the high performance computing space, and only a small subset of programmers needed to be able to write correct, efficient parallel code. However, the coming ubiquity of multicore processors means that we must cast the net of parallelism wider, opening up more classes of programs to parallelization and making parallel programming accessible to more programmers.

It is thus imperative to investigate what kinds of parallelism exists in a wide variety of programs and to ameliorate the difficulties of writing parallel programs. In this thesis, we focus on writing and efficiently executing parallel programs that exhibit *amorphous data-parallelism*, which we explain in the following

section.

1.2 Amorphous data-parallelism

There are two key issues that programmers must consider when writing multithreaded code: (i) dividing a program up among multiple processors, and (ii) ensuring that multiple threads correctly coordinate access to shared data. Essentially, these are the problems of *finding parallelism* and *exploiting parallelism*.

A common source of parallelism is *data-parallelism* [62]. In data-parallelism, a loop iterates over some iteration space, and each iteration performs the same operations on a different point in the iteration space. If the iterations are independent of one another, these iterations can be executed in parallel. By focusing on data-parallelism, we can reduce the problem of parallelization to (i) identifying potentially data-parallel loops and (ii) correctly parallelizing such loops.

Data-parallelism in regular programs Data-parallelism often appears in *regular* programs, which manipulate dense matrices and arrays. Common linear algebra operations such as matrix-vector multiply and matrix-matrix multiply are naturally data-parallel. Many language extensions have been proposed to allow programmers to express data-parallelism in their applications, from `DO-ALL` loops in High-performance Fortran [75] to the `parallel-for` construct of OpenMP [96]. Special purpose languages, such as ZPL, have also been proposed for writing data-parallel, regular programs [22].

In order to successfully use these systems, a programmer must be able to say precisely which loops are data-parallel, and hence have iterations that are independent. For regular programs, several sophisticated dependence analyses have been proposed to determine whether iterations of a loop are independent

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

Figure 1.1: Pseudocode of the mesh refinement algorithm

[35, 103]. These analyses can be integrated into a compiler, allowing programmers to simply write sequential programs while leaving the tasks of finding and exploiting parallelism to compilers and run-time systems. In practice, though, it is up to the programmer to determine which loops are data-parallel.

Data-parallelism in irregular programs While much of the parallelization research of the last 30 years has focused on regular programs, parallelizing *irregular* programs is much harder. Irregular programs use pointer-based data structures such as lists, trees and graphs. Unfortunately, the techniques developed for parallelizing regular programs do not apply to irregular programs, and, indeed, it is not apparent that irregular programs have significant amount of coarse-grain parallelism to exploit.

We have performed several case studies and found that irregular programs often exhibit a form of *amorphous data-parallelism* that manifests itself as iterative computations over worklists of various kinds [77, 78, 79, 80]. Consider 2-D Delaunay mesh refinement, an important irregular code used in graphics and

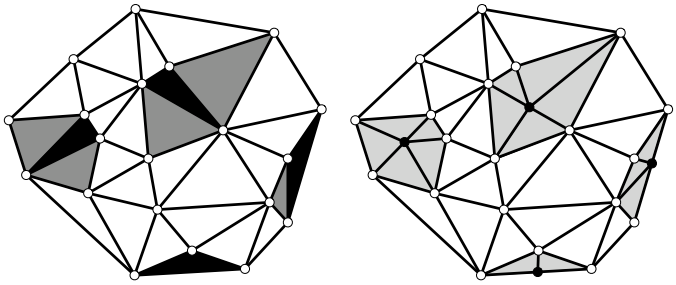


Figure 1.2: Mesh refinement

finite-element solvers[1]. The input to the algorithm is an initial triangulation of a region in the plane, as shown in Figure 1.2. Some of the triangles in this mesh may be badly shaped (these are shown in black in Figure 1.2(a)); if so, an iterative refinement procedure, shown in Figure 1.1, 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 1.2(a)), and (iii) re-triangulates that cavity (shown in light grey in Figure 1.2(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.

The key difference between data-parallelism as found in regular programs and the amorphous data-parallelism we see here is that *iterations in an amor-*

[1]This application is discussed in more detail in Section 2.1

phous data-parallel loop are not necessarily independent. In fact, the dependence or independence of two iterations (for example, whether or not two iterations in Delaunay mesh refinement produce overlapping cavities) in an amorphous data-parallel program is often dependent on the input data and cannot be determined statically.

We have found several instances of amorphous data-parallelism in our research, including mesh refinement, Delaunay triangulation (generating the initial mesh) [47], augmenting-paths maxflow [29], preflow-push maxflow [42], agglomerative clustering [125], SAT-solvers such as WalkSAT [117] and Chaff [88], as well as several other applications. We believe that this is a common pattern of parallelism in irregular programs and is the most promising type of parallelism to exploit.

1.3 Existing approaches to parallelizing irregular programs

Current approaches for parallelizing irregular applications are of varying applicability to programs exhibiting amorphous data-parallelism, and each technique has drawbacks, reducing its utility for such programs. These techniques can be divided into into static, semi-static, and dynamic approaches.

Static approaches. One approach to parallelization is to use a compiler to analyze and transform sequential programs into parallel ones, using techniques like *points-to analysis* [6, 26, 30, 33, 63, 81, 123, 135] and *shape analysis* [23, 32, 40, 49, 65, 70, 82, 112]. The basic approach is to leverage such static analyses to determine what portions of a program are independent of one another and hence can be executed in parallel [41, 54, 126].

The weakness of this approach is that the parallel schedule produced by the

compiler must be valid for all inputs to the program. This means that these compile-time techniques cannot handle data-dependent parallelism, which we have found to be prevalent in amorphous data-parallel applications such as Delaunay mesh refinement. The compile-time nature of these approaches will lead to their conservatively serializing the entire execution of an amorphous data-parallel loop.

This conclusion holds even if dependence analysis is replaced with more sophisticated analysis techniques such as commutativity analysis [109]. While this analysis allows for some dependences to be ignored by the parallelization analysis (because the operations executing in parallel commute), the static nature of the analysis still leads to conservative results, as it must consider all possible inputs to the program.

Hendren *et al.* proposed providing data structure abstractions to supplement a traditional static analysis, in an attempt to expose further opportunities for program transformations, including parallelization [55]. However, because the eventual parallelization transformations must be valid for all inputs, this approach nevertheless cannot handle the highly data-dependent parallelism found in amorphous data parallelism.

Semi-static approaches. In the *inspector-executor* approach, [102], the computation is split into two phases, an *inspector* phase that determines dependences between units of work, and an *executor* phase that uses the schedule to perform the computation in parallel.

For the inspector-executor approach to be useful, a program must obey two properties: (i) all the tasks that a program will execute must be known ahead of time, and (ii) it must be possible to determine the dependences between tasks without executing them. Unfortunately, in our applications, the data sets of-

ten change throughout execution, as in mesh generation and refinement (see Sections 2.1 and 2.2). Even when the data sets do not change, the work performed by the application is discovered dynamically, as in B-K maxflow and preflow-push maxflow (see Sections 2.4 and 2.5). In these scenarios, the work that needs to be scheduled cannot be determined until *after* the execution completes. Clearly, for our applications, the standard inspector-executor approach is not sufficient.

One option is to combine the inspector executor approach with bulk-synchronous parallelism (BSP) [130]. This approach can avoid the problem of dynamically generated work as follows. The workset (which represents all currently discovered work) is processed to determine which iterations may conflict with one another. A maximally independent subset of these iterations is then determined. These iterations can then be executed in parallel as a single, parallel "super-step." After execution, a new workset is generated (from the iterations that have not yet executed, as well as any newly generated work), which is then processed again. In essence, this approach utilizes the inspector-executor paradigm once per super-step. Recently, Gary Miller *et al*. performed a theoretical study of such an execution scheme for Delaunay mesh refinement [66]. While this approach is theoretically feasible for extracting parallelism from amorphous data-parallel applications, it may be too expensive to run the inspector phase in practice.

Dynamic approaches. In dynamic approaches, parallelization is performed at run-time, and is known as *speculative* or *optimistic* parallelization. The program is executed in parallel assuming that dependences are not violated, but the system software or hardware detects dependence violations and takes appropriate corrective action such as killing off the offending portions of the program and

re-executing them sequentially. If no dependence violations are detected by the end of the speculative computation, the results of the speculative computation are *committed* and become available to other computations.

Fine-grain speculative parallelization for exploiting instruction-level parallelism was introduced around 1970; for example, Tomasulo's IBM 360/91 fetched instructions speculatively from both sides of a branch before the branch target was resolved [128]. Speculative *execution* of instructions past branches was studied in the abstract by Foster and Riseman in 1972 [21], and was made practical by Josh Fisher when he introduced the idea of using branch probabilities to guide speculation [36]. Branch speculation can expose instruction-level (fine-grain) parallelism in programs but not the data-dependent coarse-grain parallelism in applications like Delaunay mesh refinement.

One of the earliest implementations of *coarse-grain* optimistic parallel execution was in Jefferson's 1985 Time Warp system for distributed discrete-event simulation [68]. In 1999, Rauchwerger and Padua described the LRPD test for supporting speculative execution of FORTRAN DO-loops in which array subscripts were too complex to be disambiguated by dependence analysis [108]. This approach can be extended to while-loops if an upper bound on the number of loop iterations can be determined before the loop begins execution [107]. More recent work has provided hardware support for this kind of coarse-grain loop-level speculation, now known as *thread-level speculation* (TLS) [27, 51, 76, 106, 108, 124, 131].

However, there are fundamental reasons why current TLS implementations cannot exploit the parallelism in our applications. One problem is that many of these applications, such as Delaunay mesh refinement, have unbounded while-loops, which are not supported by most current TLS implementations since they

target FORTRAN-style DO-loops with fixed loop bounds. A more fundamental problem arises from the fact that current TLS implementations track dependences by monitoring the reads and writes made by loop iterations to memory locations. For example, if iteration *i+1* writes to a location before it is read by iteration *i*, a dependence violation is reported, and iteration *i+1* must be rolled back.

For irregular applications that manipulate pointer-based data structures, this is too strict and the program will perform poorly because of frequent roll-backs. To understand this, consider the worklist in Delaunay mesh refinement. Regardless of how the worklist is implemented, there must be a memory location (call this location *head*) that points to a cell containing the next bad triangle to be handed out. The first iteration of the while loop removes a bad triangle from the worklist, so it reads and writes to *head*, but the result of this write is not committed until that iteration terminates successfully. A thread that attempts to start the second iteration concurrently with the execution of the first iteration will also attempt to read and write *head*, and since this happens before the updates from the first iteration have been committed, a dependence conflict will be reported (the precise point at which a dependence conflict will be reported depends on the TLS implementation). The manipulation of other data structures, such as the mesh, may also create such conflicts.

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

A second restriction of TLS is that it is tied to a particular loop ordering: speculatively executing loop iterations must commit in the order they would have executed when running sequentially. However, as we saw with De-

launay mesh refinement, these sequential ordering constraints are often over-determined: the application will produce the correct result regardless of the order of execution. We see in Chapters 4 and 5 how we can take advantage of this flexibility to dramatically improve the performance of speculative parallelization. As these techniques depend on being able to re-order speculative execution, they are not applicable to current TLS systems.

Finally, Herlihy and Moss have proposed to simplify shared-memory programming by eliminating lock-based synchronization constructs in favor of *transactions* [59]. There is growing interest in supporting transactions efficiently with software and hardware implementations of *transactional memory* [5, 52, 53, 59, 58, 84, 86, 104, 113, 118]. Most of this work is concerned with optimistic *synchronization* and not optimistic *parallelization*; that is, their starting point is a program that has already been parallelized (for example, the SPLASH benchmarks [52] or the Linux kernel [105]), and the goal is to find an efficient way to synchronize parallel threads. In contrast, our goal is to find the right abstractions for expressing amorphous data-parallelism in irregular applications, and to support these abstractions efficiently; synchronization is only one part of the bigger challenge of parallelizing irregular applications. Furthermore, most implementations of transactional memory track reads and writes to memory locations, so they suffer from the same problems as current TLS implementations. Open nested transactions [89, 90, 93] have been proposed recently as a solution to this problem, and they are discussed in more detail in Section 3.3.5.

1.4 Our approach

The preceding discussion of parallelization techniques leads us to consider several key features that an approach to parallelizing amorphous data-parallel ap-

plications should possess:

- *A reasonable sequential programming model.* In order for parallel programming to become widespread, we must reduce the barrier to entry. As programmers are used to thinking of algorithms sequentially, we would like to preserve sequential semantics as much as possible in our programming model. Programmers should be able to write programs with well-understood sequential semantics and rely on compilers and run-time systems to ensure that those semantics are maintained during parallel execution.
- *A dynamic approach to parallelization.* Clearly, static and semi-static approaches to parallelizing irregular programs are insufficient. The input-dependence and constantly changing data structures that we see in amorphous data-parallelism make dynamic parallelization techniques the only viable approach to parallelizing these types of programs.
- *A higher level of abstraction.* Most existing dynamic approaches operate at too low a level. By focusing on reads and writes to individual memory locations, these techniques can often miss the forest for the trees and unnecessarily restrict parallel execution. Instead, we should raise the level of abstraction; our goal should be to find a set of abstractions suitable for expressing amorphous data-parallelism, and correctly exploiting that parallelism.

This last point is crucial. Niklaus Wirth famously said that "algorithms + data structures = programs" [136]. Unfortunately, existing techniques, both static and dynamic, attempt to parallelize *programs* without considering the *algorithms* they implement or the *data structures* they use.

In a sense, the purpose of compiler analyses is to take an existing program, written in languages such as C++ or Java, and divine higher level semantics (such as shape properties), and then to take advantage of this information to parallelize a program. Unfortunately, these analyses are imprecise and thus cannot take full advantage of the semantics that a programmer understands about a program to parallelize an application; the requirement that a static analysis be conservative prevents such analyses from effectively parallelizing many programs, especially those that fundamentally require dynamic approaches such as speculative parallelism.

Dynamic approaches such as thread level speculation attempt to take an existing program and speculatively parallelize it. Unfortunately, these approaches only track low level memory accesses and so do not have a full picture of the algorithms and data structures that a programmer is using and hence can be too conservative in their parallelization.

By raising the level of abstraction, programmers can express their programs in terms of their component algorithms and data structures. We can then leverage information about the algorithm and semantic properties of the data structures to make more intelligent decisions about what parts of the algorithm to parallelize and how to parallelize it correctly and efficiently.

This thesis presents the *Galois system* to parallelize irregular programs. The Galois system consists of three major, interlocking components: *user code*, *Galois class libraries*, and the *Galois run-time*. The user code represents code that a programmer would write to express a particular algorithm such as Delaunay mesh refinement. We provide abstractions that allow programmers to naturally express the data-parallelism inherent in their algorithms, capturing key algorithmic properties (such as ordering constraints on loop iterations) while main-

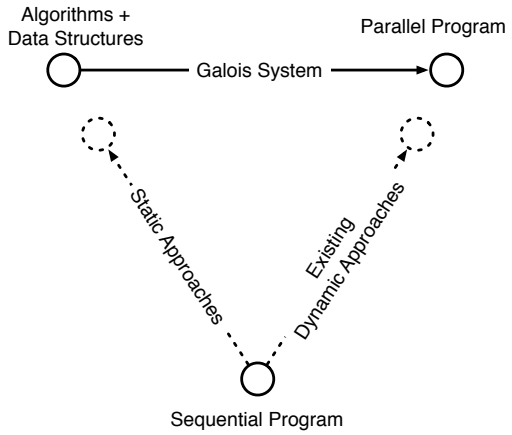


Figure 1.3: The goal of the Galois system

taining sequential semantics. Galois class libraries raise the level of abstraction of the shared data in a Galois program: rather than focusing on individual reads and writes, the class libraries capture important data structure semantics (such as the semantics of methods, or locality properties). The Galois run-time then uses optimistic parallelization to run the data-parallel sections of an application concurrently, leveraging the abstractions provided by the user code and the class libraries to ensure that the parallelization is both correct and efficient.

The goal of the Galois system is thus to allow programmers to write programs with a clean delineation between algorithms and data structures, and to then take this information and turn it into a parallel program. Figure 1.3 shows how this approach compares to static approaches, which attempt to reverse engineer programs before parallelizing them, and dynamic techniques such as thread level speculation, which attempt to directly parallelize "low-level" sequential programs.

We believe that this approach to tackling amorphous data-parallelism is promising, as it requires very little programmer effort to produce correct parallel code while still achieving significant scalability on small-scale multicore

systems.

1.5 Contributions and organization

This thesis makes several contributions in the areas of programming languages and speculative parallelization:

- **A general paradigm of parallelism**: Chapter 1 presents the first discussion of *amorphous data-parallelism*, which can provide a significant amount of parallelism. Chapter 2 surveys a number of irregular applications and demonstrates the prevalence of this paradigm across a wide variety of applications.
- **Language extensions for amorphous data-parallelism**: Chapter 3 presents novel language extensions which allow programmers to easily express programs containing amorphous data-parallelism. These constructs capture key algorithm semantics, such as ordering constraints, which allow a run-time system to automatically parallelize irregular, data-parallel applications.
- **Abstractions for data structures**: Chapter 3 also discusses how we can raise the level of abstraction of irregular data structures used in irregular applications. By allowing programmers to describe key semantic properties of objects, such as the commutativity of method invocations, we can develop systems which can exploit these semantics rather than viewing irregular structures as simple collections of pointers and objects.
- **A run-time system for exploiting abstractions**: Chapter 3 then presents the first run-time system which can leverage algorithmic and data structure abstractions efficiently, allowing us to extract significant amounts of

parallelism from programs exhibiting amorphous data-parallelism.

- **Abstractions for capturing locality properties**: Chapter 4 develops a set of abstractions for capturing locality properties of irregular data structures, based around *partitioning*. We show how to exploit this semantic information to improve locality and speculation accuracy, as well as to reduce contention and speculation overheads.
- **A scheduling framework for amorphous data-parallelism**: Chapter 5 presents a framework for scheduling the parallel execution of programs using the Galois system. This framework generalizes much of the previous work in the scheduling of data-parallel loops, and provides a general way of expressing the scheduling decisions a programmer must make when parallelizing irregular programs. We show how to use this framework to take advantage of data-structure and algorithm semantics to generate high-performing schedules.

We then conclude in Chapter 6 with a brief discussion of other models of parallelism, a summary of our contributions, and a discussion of future work.

CHAPTER 2

APPLICATION STUDIES

2.1 Delaunay mesh refinement

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

Although many types of meshes are used in practice, *Delaunay meshes* are particularly important since they have a number of desirable mathematical properties [25]. The Delaunay triangulation for a set of points in the plane is the triangulation such that no point is inside the circumcircle of any triangle (this property is called the *empty circle property*). An example of such a mesh is shown in Figure 2.1.

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose quality constraints governing the shape and size of the triangles. For a given Delaunay mesh, this is accomplished by *iterative mesh refinement*, which successively fixes "bad" triangles (triangles that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating. Figure 2.2 illustrates this process; the shaded triangle in Figure 2.2(a) is assumed to be bad. To fix this bad triangle, a new point is added at the center of this triangle's circumcircle. Adding this point may invalidate the empty circle property for some neighboring triangles, so all affected triangles are determined (this region is called the *cavity* of the bad triangle), and the cavity is re-triangulated, as shown in Figure 2.2(c) (in this figure, all triangles lie in the cavity of the shaded bad triangle). Re-triangulating a cavity may generate new bad triangles but it

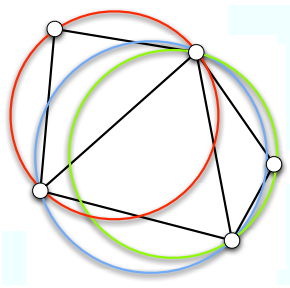


Figure 2.1: A Delaunay mesh

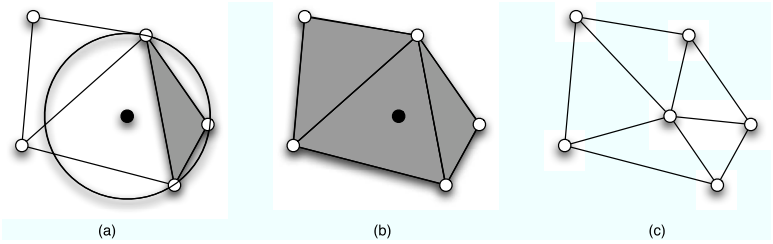


Figure 2.2: Fixing a bad element.

can be shown that this iterative refinement process will ultimately terminate and produce a guaranteed-quality mesh. Different orders of processing bad elements lead to different meshes, although all such meshes satisfy the quality constraints [25].

Figure 2.3 shows the pseudocode for mesh refinement. The input to this program is a Delaunay mesh in which some triangles may be bad, and the output is a refined mesh in which all triangles satisfy the quality constraints. There are two key data structures used in this algorithm. One is a worklist containing the bad triangles in the mesh. The other is a graph representing the mesh structure; each triangle in the mesh is represented as one node, and edges in the graph represent triangle adjacencies in the mesh.

Opportunities for exploiting parallelism. The natural unit of work for parallel execution is the processing of a bad triangle. Because a cavity is typically

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

Figure 2.3: Pseudocode of the mesh refinement algorithm

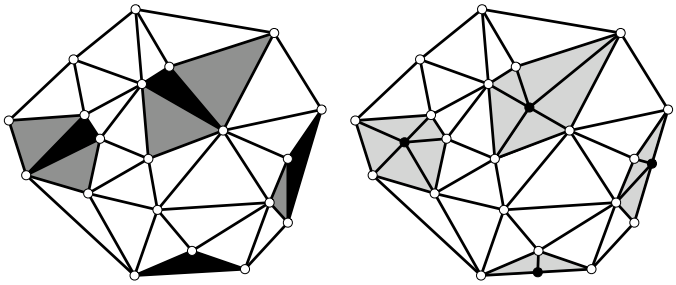


Figure 2.4: Processing triangles in parallel

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.

An example of processing several triangles in parallel can be seen in Figure 2.4. The left mesh is the original mesh, while the right mesh represents the refinement. In the left mesh, the *black* triangles represent the "bad" triangles, while the *dark grey* are the other triangles in the cavity. In the right mesh, the *black* points are the newly added points and *light grey* triangles are the newly created triangles.

Clearly, this algorithm is an example of a worklist algorithm where units of work from the worklist may be independent. This application is perhaps the canonical example of amorphous data-parallelism.

2.2 Delaunay triangulation

The second benchmark we studied is Delaunay *triangulation*, the creation of a Delaunay mesh, given a set of input points. In general, this mesh may have bad triangles, which can be eliminated using the refinement code discussed in Section 2.1.

Pseudocode for this algorithm is shown in Figure 2.5. The main loop iterates over the set of points, inserting a new point into the current mesh at each iteration to create a new mesh that satisfies the Delaunay property. When all the points have been inserted, mesh construction is complete. To insert a point p into the current mesh, the algorithm determines the triangle t that contains point p (line 6), then splits t into three new triangles that share point p as one of their vertices (line 7). These new triangles may not satisfy the Delaunay property, so a procedure called *edge flipping* is used to restore the Delaunay property. Edge flipping examines each edge of the newly created triangles (lines 9-15); if any edge does not satisfy the Delaunay property[1] (line 11), the edge is flipped, removing the two non-Delaunay triangles and replacing them with two new triangles (line 12). The edges of these newly created triangles are examined in turn (line 13). When this loop terminates, the resulting mesh is once again a Delaunay mesh.

To locate the triangle containing a given point, we use a data structure called

[1] An edge satisfies the Delaunay property if and only if the two triangles incident on the edge satisfy the Delaunay property.

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

Figure 2.5: Pseudocode for Delaunay triangulation

the *history DAG* [47]. Intuitively, this data structure can be viewed as a ternary search tree. The leaves of the DAG represent the triangles in the current mesh. When a triangle is split (line 7), the three new triangles are added to the data structure as children of the original triangle. The only twist to this intuitive picture is that when an edge is flipped (line 12), the two new triangles are children of both old triangles, so the data structure is a DAG in general, rather than a tree.

We can use this structure to efficiently locate which triangle contains a given point by walking down from the root of the DAG. If the DAG is more or less balanced, point location can be performed in $O(log(N))$ time where N is the number of triangles in the current mesh. However, if the data structure becomes long and skinny, point location can take $O(N)$ time, resulting in poor performance. To avoid this worst-case behavior, Guibas *et al* recommend inserting points in random order rather than in a spatially coherent order.

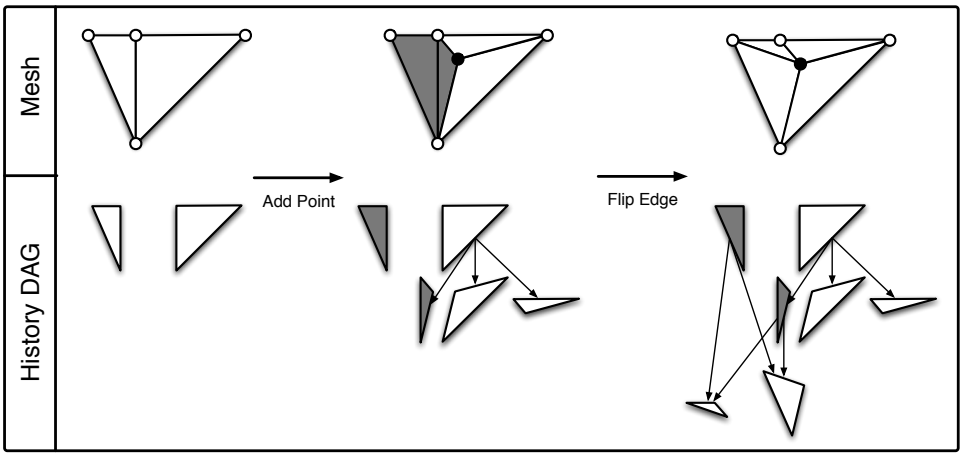


Figure 2.6: Delaunay triangulation: Adding a point to a mesh, then flipping an edge

By way of example, Figure 2.6 shows two steps of this algorithm, with the mesh at each step shown on top and a portion of history DAG shown at the bottom. In the first step, a single point is placed in the large triangle, which is then split into three, as is reflected in the history DAG. However, the triangles shaded light grey do not satisfy the empty-circle property. The Delaunay property is restored in the second step by flipping the edge shared between the two triangles. This is reflected in the history DAG by adding the two new triangles and having both old triangles point to both new triangles.

Opportunities for exploiting parallelism. This is yet another worklist algorithm, where the units of work are the points to be inserted into the Delaunay mesh. We can thus parallelize it by attempting to insert multiple points into the mesh in parallel. Inserting a new point only affects the triangles in its immediate neighborhood, so most points from the worklist can be inserted independently, as long as they are sufficiently far apart in the geometry. Conflicts can occur when two threads attempt to manipulate the same triangles in the mesh (lines 7 and 13). Note that unlike Delaunay mesh refinement, this algorithm does not

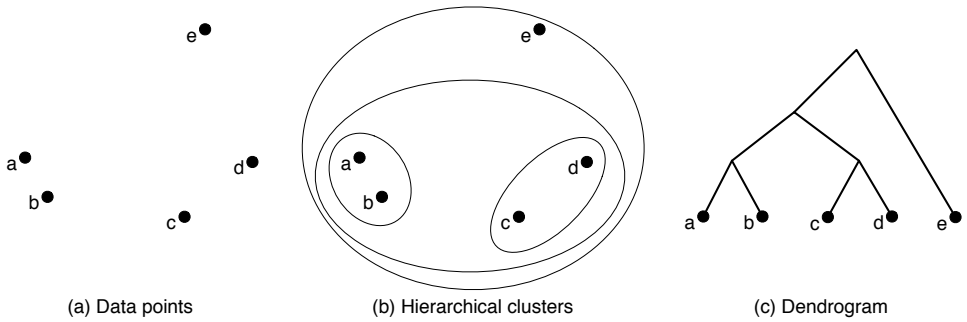


Figure 2.7: Agglomerative clustering

add new elements to the worklist.

2.3 Agglomerative clustering

Another interesting irregular application is *agglomerative clustering*, a well-known data-mining algorithm [125]. This algorithm is used in graphics applications for handling large numbers of light sources [133].

The input to the clustering algorithm is (1) a data-set, and (2) a measure of the "distance" between items in the data-set. Intuitively, this measure is an estimate of similarity—the larger the distance between two data items, the less similar they are believed to be. The goal of clustering is to construct a binary tree called a dendrogram whose hierarchical structure exposes the similarity between items in the data-set. Figure 2.7(a) shows a data-set containing points in the plane, for which the measure of distance between data points is the usual Euclidean distance. The dendrogram for this data set is shown in Figures 2.7(b,c).

2.3.1 Priority queue-based clustering

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 [125]. The algorithm terminates when there is only one point left in the data-set.

Pseudocode for the algorithm is shown in Figure 2.8. The central data structure is a priority queue whose entries are ordered pairs of points `<x, y>`, such that `y` is the nearest neighbor of `x` (we call this `nearest(x)`). In each iteration of the while loop, the algorithm dequeues the top element of the priority queue to find a pair of points `<p, n>` that are closer to each other than any other pair of points, and clusters them. These two points are then replaced by a new point that represents this cluster. The nearest neighbor of this new point is determined, and the pair is entered into the priority queue. If there is only one point left, its nearest neighbor is the point at infinity.

To find the nearest neighbor of a point, we can scan the entire data-set at each step, but this is too inefficient. A better approach is to sort the points by location, and search within this sorted set to find nearest neighbors. If the points were all in a line, we could use a binary search tree. Since the points are in higher dimensions, a multi-dimensional analog called a *kd-tree* is used [11]. The kd-tree is built at the start of the algorithm, and it is updated by removing the points that are clustered, and then adding the new point representing the cluster, as shown in Figure 2.8.

Opportunities for exploiting parallelism. Since each iteration clusters the two closest points in the current data-set, it may seem that the algorithm is in-

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

Figure 2.8: Pseudocode for agglomerative clustering

herently sequential. In particular, an item `<x,nearest(x)>` inserted into the priority queue by iteration `i` at line 17 may be the same item that is dequeued by iteration `(i+1)` in line 5; this will happen if the points in the new pair are closer together than any other pair of points in the current data-set. On the other hand, if we consider the data-set in Figure 2.7(a), we see that points `a` and `b`, and points `c` and `d` can be clustered concurrently since neither cluster affects the other. Intuitively, if the dendrogram is a long and skinny tree, there may be few independent iterations, whereas if the dendrogram is a bushy tree, there is parallelism that can be exploited since the tree can be constructed bottom-up in parallel. As in the case of Delaunay mesh refinement, the parallelism is very data-dependent. In experiments on graphics scenes with 20,000 lights, we have found that on average about 100 clusters can be constructed concurrently; thus,

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

Figure 2.9: Psuedocode for unordered agglomerative clustering

there is substantial parallelism that can be exploited.

We can view this application as yet another worklist algorithm. Essentially, the priority queue is a worklist which enforces some ordering constraints. Unlike in Delaunay mesh refinement and triangulation, the elements from the worklist cannot be executed in any order: they must respect the ordering constraints of the priority queue. However, as the previous discussion makes clear, it may still be possible to extract parallelism from an algorithm that operates over an ordered worklist.

2.3.2 Unordered clustering

The algorithm described above is a greedy algorithm using an ordered set, but under some mild conditions on the metric, there is an equivalent algorithm

using an unordered set iterator, shown in Figure 2.9. Intuitively, we can cluster two elements together whenever we can prove that the ordered greedy algorithm would also cluster them eventually. If the metric is non-decreasing with respect to set membership and if two elements agree that they are each other's best match then it is safe to cluster them immediately.

Opportunities for exploiting parallelism. This variant of agglomerative clustering operates over an unordered worklist: the resulting tree is not affected by the order in which the worklist is processed, allowing elements to be processed in parallel. Conflicts arise when one thread modifies the kdtree (lines 9 to 11), and this changes the result of another thread's ongoing nearest neighbor computations (lines 4 and 6).

2.4 Boykov-Kolmogorov maxflow

The Boykov-Kolmogorov algorithm is a maxflow algorithm used in image segmentation problems [17] (hereafter abbreviated as "B-K maxflow"). Like the standard augmenting paths algorithm [29], 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 maxflow algorithm outperforms other maxflow algorithms for graphs arising from image segmentation problems [17].

The B-K maxflow 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

```
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 2.10: Pseudocode for Boykov-Kolmogorov algorithm

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 2.10. 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 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 run-time depends on the structure of the graph and the evolution of the computation, so optimistic parallelization seems appropriate.

2.5 Preflow-push maxflow

Although experiments on uniprocessors have shown that the Boykov-Kolmogorov algorithm outperforms other maxflow algorithms for graphs arising from image segmentation problems [17], it is not known whether this holds for parallel implementations. Therefore, we also implemented the Goldberg-Tarjan preflow-push algorithm [42], 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 B-K maxflow algorithm, which maintains a valid flow at all times. Pseudocode for the algorithm is given in Figure 2.11.

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*

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

Figure 2.11: Pseudocode for preflow-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 exploiting 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.

As noted in previous work [7], the actual amount of parallelism at any given point of preflow-push is very data-dependent. It is affected by the structure of the flow graph. It is easy to construct cases where there is only ever one active node, which serializes the computation, or trivially parallel cases where there are always exactly N active nodes for the N processors available, and they never interfere with each other. Further, the parallelism is dependent on the stage of computation preflow-push is in. While perfect speedup is unrealistic, it is still possible to get significant speedup by working on nodes in parallel.

CHAPTER 3

THE GALOIS SYSTEM

3.1 Overview

In order to parallelize applications of the sort presented in Chapter 2, we developed the *Galois system*, a programming model and run-time system which enables the optimistic parallelization of amorphous data-parallel programs.

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

In this spirit, the Galois system is divided into three parts: (i) top-level *user code* which creates and manipulates shared objects (Section 3.2), (ii) a set of *Galois library classes* which provide implementations of the shared objects used by the user code (Section 3.3), and (iii) the *Galois run-time* which is responsible for detecting and recovering from potentially unsafe accesses to shared objects made by optimistic computation (Section 3.4).

Consider Delaunay mesh refinement. The relevant shared objects are the mesh and the worklist, and the Galois class libraries will provide implementations of these objects. The user code implements the mesh refinement algorithm described in Section 2.1. The Galois run-time is responsible for executing this

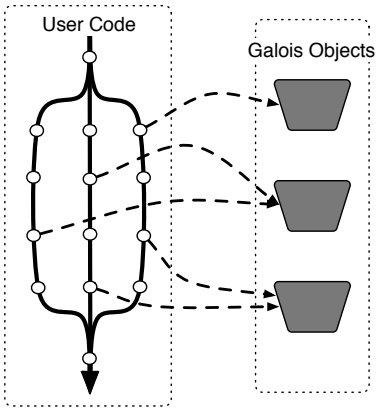


Figure 3.1: High-level view of Galois execution model

user code in parallel and ensuring that it behaves properly. Crucially, while the user code is executed concurrently by some number of threads (orchestrated by the run-time), *it is not explicitly parallel*, and makes no mention of threads or locks. Instead, the relevant information for ensuring correct parallel execution is contained in the Galois library classes and managed by the run-time, as we discuss below. Figure 3.1 is a pictorial view of this execution model.

This design allows for a clean *separation of concerns*. Rather than placing the entire parallelization burden on every programmer, programmers writing user code can focus on implementing their algorithm, while expert library programmers can focus on the difficulties of writing parallel code. This division of responsibilities is a key consideration in the design of the Galois system and its extensions.

For brevity, we refer to programmers who write user code as "Joe Programmers"[1]. These programmers may have a deep understanding of the algorithms they are implementing, but may not be well versed in parallel programming techniques and idioms. By way of contrast, we refer to programmers to design

[1] As in "average Joe."

and write the Galois libraries as "Steve Programmers"[2]. These programmers are knowledgeable in the domain of parallel programming, but need not have any domain-specific algorithmic knowledge.

Because the universe of programmers contains far more Joe Programmers than Steve Programmers, the Galois approach makes writing code that will ultimately run in parallel feasible for a larger class of programmers, as the difficult parallel code can be written once, encapsulated in libraries, and then used repeatedly by programmers less conversant in parallel programming techniques.

3.2 Programming model

3.2.1 Optimistic set iterators

As mentioned above, the client code is not explicitly parallel; instead parallelism is packaged into two constructs that we call *optimistic iterators*. In the compiler literature, it is standard to distinguish between *do-all* loops and *do-across* loops [73]. The iterations of a do-all loop can be executed in any order because the compiler or the programmer asserts that there are no dependences between iterations. In contrast, a do-across loop is one in which there may be dependences between iterations, so proper sequencing of iterations is essential. We introduce two analogous constructs for packaging optimistic parallelism.

- **Set iterator:** `for each e in Set S do B(e)`

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

[2] An arbitrary designation for "expert" programmers

```
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: }
```

Figure 3.2: Delaunay mesh refinement using set iterator

but any serial order of executing iterations is permitted. When an iteration executes, it may add elements to `S`.

- **Ordered-set iterator:** `for each e in Poset S do B(e)`

  This construct is an iterator over a partially-ordered set (Poset) S. It asserts that in a serial execution of the loop, the iterations must be performed in the order dictated by the ordering of elements in the Poset `S`. There may be dependences between iterations, and as in the case of the set iterator, elements may be added to `S` during execution.

The set iterator is a special case of the ordered-set iterator but it can be implemented more efficiently, as we see in section 3.4.3

Figure 3.2 shows the client code for Delaunay mesh generation. Instead of a work list, this code uses a set and a set iterator. The Galois version is not only simpler but also makes evident the fact that the bad triangles can be processed in any order; this fact is absent from the more conventional code of Figure 2.3 since it implements a particular processing order.

Note that the Galois program shown in Figure 3.2 can be viewed as a purely sequential program. Its semantics can be understood without appealing to a parallel execution model. The only additional effort a programmer must expend when writing Galois programs versus standard sequential programs is to understand the ordering constraints imposed by a particular algorithm.

3.2.2 Memory model

The Galois system uses an object-based, shared memory model. The system relies on cache coherence to communicate shared data between processors. All shared data is encapsulated in objects, and the only means of reading or writing shared data is by invoking methods on those objects. This approach is in contrast to that taken by TLS or word-based transactional memories, which allow threads to perform bare reads and writes to shared memory.

While limiting shared memory access to method invocations on shared objects may seem limiting, in practice this is a reasonable approach. Many programs are object-oriented, naturally performing all heap updates through method invocations on objects. Furthermore, the Galois system is able to leverage the semantics of shared objects to make more precise decisions about when parallel execution is safe (see Section 3.3.2) than if undisciplined reads and writes were allowed.

3.2.3 Execution model

Although the semantics of Galois iterators can be specified without appealing to a parallel execution model, these iterators provide hints from the programmer to the Galois run-time system that it may be profitable to execute the iterations

in parallel. Of course any parallel execution must be faithful to the sequential semantics.

The Galois concurrent execution model is the following. A master thread begins the execution of the program; it also executes the code outside iterators. When this master thread encounters an iterator, it enlists the assistance of some number of worker threads to execute iterations concurrently with itself. The assignment of iterations to threads is under the control of a scheduling policy implemented by the run-time 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 read and write to the objects in shared memory, and we must ensure that these reads and writes are properly coordinated. Section 3.3 describes the information that must be provided by the Galois class writer to enable this. Section 3.4 describes how the Galois run-time system uses this information to ensure that the sequential semantics of iterators are respected.

3.2.4 Discussion

Set iterators

Although the Galois set iterators introduced in Section 3.2.1 were motivated in this paper by the applications discussed in Chapter 2, they are very general, and we have found them to be useful for writing other irregular applications such as advancing front mesh generators [92], and WalkSAT solvers [117]. Many of these applications use "work-list"-style algorithms, for which Galois iterators

are natural, and the Galois approach allows us to exploit the amorphous data-parallelism in these irregular applications.

SETL was probably the first language to introduce an unordered set iterator [71], but this construct differs from its Galois counterpart in important ways. In SETL, the set being iterated over can be modified during the execution of the iterator, but these modifications do not take effect until the execution of the entire iterator is complete. In our experience, this is too limiting because work-list algorithms usually involve data-structure traversals of some kind in which new work is discovered during the traversal. The *tuple iterator* in SETL is similar to the Galois ordered-set iterator, but the tuple cannot be modified during the execution of the iterator, which limits its usefulness in irregular applications. Finally, SETL was a sequential programming language. DO-loops in FORTRAN are a special case of the Galois ordered-set iterator in which iteration is performed over integers in some interval.

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

Nested iterators

Languages such as NESL [13] support nested data-parallelism, with data-parallel operations exposing additional data-parallelism. In our programming model, this style of parallelism manifests itself as nested iterators: an iteration contains within it an ordered or unordered set iterator.

Although in our current applications, we have not found it necessary to use nested iterators, properly dealing with the multiple levels of parallelism afforded by nested iterators is an open question. There is no fundamental problem in supporting nested iterators, but there are many different approaches one might take to support multiple levels of parallelism.

Our current implementation takes a simple "flattening" approach to nested iterators: the thread encountering a nested iterator always executes it sequentially, to completion. The inner iterator's execution is considered part of the execution of the iteration which contains it.

In general, determining the appropriate action to take when encountering a nested iterator is a policy decision; a problem of performance rather than one of correctness. For example, should the nested iterator be executed in parallel or sequentially? If we choose to execute the nested iterator in parallel, should all threads draw work from the inner iterator, or should some threads continue executing work from the outer iterator? The answers to these policy questions have deep implications for the performance of an application. The problem of deciding what the appropriate policies are, and devising an appropriate mechanism for specifying them to the run-time system, is left to future work.

3.3 Class libraries

Because the user code (purposely) contains little information regarding the parallel execution of a Galois program, so as to facilitate writing by Joe Programmer, the burden of ensuring that the sequential semantics of the iterators in the program are respected fall on the Galois run-time. The run-time leverages information provided by the *Galois class libraries*, which specify information regarding how objects used in a Galois program can safely be manipulated in parallel. Writing these classes is the responsibility of Steve Programmer.

The key idea to preserving the sequential semantics of set iterators is to execute iterations *transactionally* [45]; this can produce a serializable schedule in which all data structures remain consistent. As long as this serializable schedule respects any ordering constraints imposed by the set being iterated over, the parallel execution will match the sequential semantics of the iterator.

To execute iterations transactionally, there are several key problems to be solved. We draw an analogy with the **ACID** properties: *atomcicity*, *consistency*, *isolation* and *durability*. In databases, these properties can be roughly defined as follows: *atomicity* — a transaction appears to execute entirely or not at all; *consistency* — the data structures in the database are in a consistent state before the transaction and after; *isolation* — transactions execute as if they were the only transaction running in the system; and *durability* — committed transactions persist in the face of system failure.

In the context of the Galois system, we are not concerned with system durability (which would require writing shared memory state to persistent storage and is an orthogonal problem to those we aim to solve). For the other three ACID properties, we can analogize database properties with attributes we want iterations in Galois programs to have (in the order we present them in the re-

mainder of this section):

- **Consistency** (Section 3.3.1) — All data structures must remain in a consistent state at all times; at any point an iteration might invoke a method on an object, it must see that object in a consistent state. We ensure this by requiring that all object methods be atomic[3].
- **Isolation** (Section 3.3.2) — An iteration must appear to execute as if it were executing by itself, without other iterations executing concurrently. In other words, the parallel execution of iterations must match some serial schedule of execution — this is the *serializability* property. We guarantee this by using *semantic commutativity* to ensure that iterations cannot view the uncommitted state of other iterations. If this happens, an iteration will no longer behave as if it is executing in isolation. When an iteration executes in a non-serializable way, it will not be allowed to commit.
- **Atomicity** (Section 3.3.3) — An iteration must run to completion and commit, or appear to have never made any changes to shared memory. We provide this through the use of *undo methods* to roll back changes made by an iteration that will not commit.

Each of these techniques, which together ensure the transactional behavior of iterations, are provided through the Galois class libraries, which provide atomic methods and the annotations required to support semantic commutativity and undo methods. We explain each component below, with reference to Figure 3.3. This figure shows set objects with methods `add(x)`, `remove(x)`, `get()` and `contains(x)` that have the usual semantics[4].

[3]Note that this is "atomic" in the standard computer science sense of being thread-safe, not in the databases sense being discussed here

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

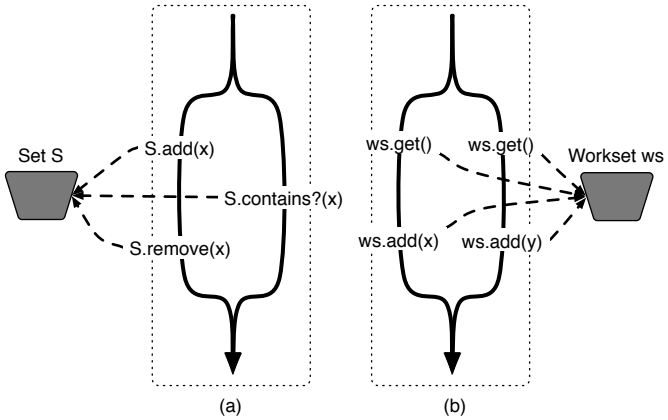


Figure 3.3: Interleaving method invocations from different iterations

3.3.1 Consistency through atomic methods

To ensure consistency, we require that every method of a shared object used in a Galois program be *atomic*. Formally, this means that every such shared object must be *linearizable* [61]. For an object to be linearizable, each method must have a single *linearization point* at some point between when the method is invoked and when it returns. The method must appear to execute instantaneously at that linearization point, behaving as if it were executed sequentially.

This property ensures that the object always remains in a consistent state. No matter how methods are invoked concurrently on the object, the linearizability property means that we can view each method as having occurred at a distinct time (the linearization point). Thus, the overall set of invocations on the object have some equivalent serial schedule of execution and the invariants of the object are never violated. For example, multiple threads can simultaneously execute methods such as `add` and `remove` on Set S in Figure 3.3(a), and as long as S is linearizable, these invocations will always see S in a consistent state, and will leave S in a consistent state when they have completed.

Providing linearizable objects can be done using any technique desired. One

solution is to use a lock on object S; if this inhibits concurrency, we can use fine-grain locks within object S. These locks are acquired before the method is invoked, and released when the method completes. Alternately, one can use transactional memory, with each method of a class enclosed in an `atomic` section [19, 53]. In this case, transactions start when a method is invoked and commit when the method completes, ensuring that each method invocation appears atomic. In our current implementation, methods are made atomic through the use of locks.

Using linearizable objects dramatically simplifies the challenge of ensuring that two iterations concurrently manipulating shared state are independent. Normally, to determine whether two iterations executing in parallel are independent, one must consider all possible interleavings of the instructions that the two iterations execute. However, because each method call can be considered to have executed at a single point, we can reduce this problem to considering interleavings of *method invocations*, rather than interleavings of all instructions.

3.3.2 Isolation through semantic commutativity

Given linearizable objects, the key issue becomes: which method interleavings can we allow while maintaining sequential semantics? Not all method interleavings will produce results that are valid under sequential semantics. Consider Figure 3.3(a). If S does not contain `x` before the iterations start, notice that in any sequential execution of the iterations, the method invocation `contains(x)` will return false. However, for one possible interleaving of operations—`add(x)`, `contains(x)`, `remove(x)`—the invocation `contains(x)` will return true, which is incorrect.

The fundamental problem in the incorrect execution is that the second iter-

ation is able to see the intermediate state of the first iteration (and thus sees `x` in the set, even though it will eventually be removed). Crucially, *the erroneous execution can only happen when the iterations are running concurrently*. When one iteration sees and depends on the intermediate state of another, no sequential execution of the iterations can produce the same result, and hence the sequential semantics of the set iterator are violated.

This problem is essentially one of ensuring the *isolation* of iterations: iterations must appear to execute as if no other iterations were executing concurrently. If all iterations are isolated from one another, this is equivalent to the execution of the iterations' being *serializable*: regardless of the parallel execution of iterations, it will appear as if they executed sequentially in some order. Note that serializability is a sufficient condition to ensure the sequential semantics of the unordered set iterator; matching the sequential semantics of the ordered set iterator requires restricting the valid sequential schedules that iterations could execute in.

Often, the consistency of data structures and the isolation of iterations are guaranteed by the same mechanism. For example, if an iteration acquires locks on objects to ensure consistency, it can release those locks only at the end of the iteration. This will ensure that no other iteration touches objects that the current iteration has touched, guaranteeing isolation. The well-known two-phase locking algorithm used in databases is an optimized version of this simple idea.

Recall that transactional memory implementations ensure the consistency of method invocations by placing each method in an `atomic` block and executing each method as a transaction. All memory locations accessed within this block are added to the transaction's read/write set, and the transactional memory hardware (or software run-time) will ensure that no other transaction (*i.e.* no

other method invocation) will interfere. To provide for iteration isolation, all the method calls in an iteration can be composed into a single atomic block; essentially the entire iteration is executed as a transaction. All the reads and writes of the entire iteration are tracked and these sets are only released at the end of the iteration. Thread-level speculation (TLS) systems operate in a similar manner.

Both of these approaches solve the problem in Figure 3.3(a). In the locking approach, the first iteration will acquire a lock on `S` when it calls `add(x)`, which it will not release until after it has executed `remove(x)`. Thus, the second iteration will only be able to execute `contains(x)` before or after the first iteration; the interleaving shown in Figure 3.3(a) will be disallowed.

In the transactional approach, the first iteration's executing `add(x)` will necessarily modify a memory location that `contains(x)` will attempt to read. Thus, the second iteration will conflict with the first iteration and again the incorrect interleaving will be disallowed.

These approaches suffice to guarantee the isolation of concurrently executing iterations—they disallow all method interleavings which break isolation. However, they can be too restrictive, forbidding interleavings which do not break isolation. Consider the program in Figure 3.3(b), which is motivated by Delaunay mesh refinement: each iteration gets a bad triangle at the beginning of the iteration, and may add some bad triangles to the work-set at the end. Because each iteration gets different bad triangles from the work-set at the beginning, and adds different triangles at the end, the interleaving shown in Figure 3.3(b) is obviously a valid interleaving; the two iterations remain isolated from one another. However, both the locking and the transactional approaches to isolation forbid this interleaving.

The locking approach prevents multiple concurrently executing iterations from accessing the same object, obviously disallowing the interleaving. The situation is more subtle for the transaction approach. Regardless of how the set object is implemented, there must be a location (call it `head`) that points to a cell containing the next triangle to be handed out. The first iteration to get work will read and write location `head`, and it will lock it for the duration of the iteration, preventing any other iterations from getting work. Most current implementations of transactional memory will suffer from the same problem since the `head` location will be in the read and write sets of the first iteration for the duration of that iteration. The crux of the problem is that the *abstract* set operations have useful semantics that are not available to an implementation that works directly on the *representation* of the set and tracks reads and writes to individual memory locations. The problem therefore is to understand the semantics of set operations that must be exploited to permit parallel execution in our irregular applications, and to specify these semantics in some concise way.

Semantic commutativity

The solution we have adopted exploits the commutativity of method invocations. Two methods commute with one another if they can be executed in either order without changing the semantic state of the object (*i.e.* the state of the object visible through its interface).

Consider two iterations, A and B. If all the methods invoked by A commute with every method invoked by B, and *vice versa*, then A and B can execute in parallel in isolation. Intuitively, the commutativity of the methods means that, regardless of how the method invocations are interleaved between the two iter-

ations, they can be "pushed past" one another until the invocations from each iteration occur contiguously, without any intervening invocations from the other iteration; this is a serial schedule of execution. By commutativity, the results of this execution are equivalent to any interleaved execution of A and B; hence, A and B are isolated from one another. This property can be trivially extended to any number of iterations, giving us the following: *If the method invocations from one iteration commute with the method invocations of all other simultaneously executing iterations, the first iteration is isolated from all other iterations*.

Turning to our running example, we see that in Figure 3.3(a), the invocation `contains(x)` does not commute with the operations from the other thread—it will return a different result depending on whether it is executed before or after `add(x)`—so the invocations from the two iterations cannot be interleaved. In Figure 3.3(b), (1) `get` operations commute with each other, and (2) a `get` operation commutes with an `add` operation provided that the operand of `add` is not the element returned by `get`. This allows multiple threads to pull work from the work-set while ensuring that sequential semantics of iterators are respected.

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

semantic commutativity (that is, commutativity with respect to the abstract data type of the class). We also note that commutativity of method invocations may depend on the arguments of those invocations. For example, an `add` and a `remove` commute only if their arguments are different.

Semantic commutativity reduces the burden placed on a programmer to ensure that iterations executing in parallel do so safely. Traditionally, guaranteeing the isolation of two concurrently executing pieces of code required making sure that all possible interleavings of instructions between those two pieces were safe. Even with atomic methods, one must consider all possible interleavings of method invocations. This leads to an exponential state space to explore.

However, with semantic commutativity, one needs to consider only pairs of methods of a given class. Each method must be checked against every other method of the class to determine if, and under what conditions, they commute. However, this is a dramatically reduced state space; a programmer need only consider $O(n^2)$ different possibilities.

Related work

The use of commutativity in parallel program execution was explored by Bernstein as far back as 1966 [12]. Conceptually, one can view commutativity conditions as a particular type of predicate locks, which are used in databases to provide logical locks on database tables (rather than locks on actual entries) [34]. In this setting, Weihl described a theoretical framework for using commutativity conditions for concurrency control [134]. Herlihy and Weihl extended this work by leveraging ordering constraints to increase concurrency but at the cost of more complex rollback schemes [60].

In the context of parallel programming, Steele described a system for ex-

ploiting commuting operations on memory locations in optimistic parallel execution [122]. However, in that work, commutativity is still tied to concrete memory locations and does not exploit properties of abstract data types like Galois does. Diniz and Rinard performed static analysis to determine *concrete* commutativity of methods for use in compile-time parallelization [109]. Semantic commutativity, as used in Galois, is more general but it must be specified by the class designer. Wu and Padua have proposed to use high level semantics of container classes [137]. They propose making a compiler aware of properties of abstract data types such as stacks and sets to permit more accurate dependence analysis.

3.3.3 Atomicity through undo methods

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

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

Note that when an iteration rolls back, all of the methods which it invokes during roll-back must succeed. Thus, we must never encounter conflicts when invoking undo methods. When the Galois system checks commutativity, it also checks commutativity with the associated undo method. Because this check has already succeeded by the time a rollback may occur, we can guarantee that the undo methods will execute without conflict, and hence the rollback will be safe.

3.3.4 Object wrappers

A key design feature of the Galois system is its ability to take any thread-safe (*i.e.* linearizable) object and use it in a transactional manner during parallel execution. To do this, we introduce *Galois wrappers*, a simple example of which can be seen in Figure 3.4.

We utilize two common design patterns in the formulation of Galois wrappers, *delegation* and *strategies* [39]. Because the thread-safe object (in this case, `Foo`) does not support transactional composition of its calls, simply invoking methods on it in an iteration is unsafe. We must thus protect it by commutativity checks to ensure isolation. This is done by using the thread-safe object as a delegate, and passing all calls to it through the Galois wrapper, as we see in Figure 3.4 with the method `bar`. Note that the delegate pattern makes it easy to replace simple data structures with clever, hand-tuned concurrent data structures [114] if necessary, without changing the rest of the program: one merely needs to change the delegate object of the Galois wrapper.

Isolation and atomicity are ensured by a `ConflictDetection` strategy object. This object is passed information about the method and its arguments, which it then uses to perform the commutativity checks and set up any associated undo methods. These checks are encapsulated in a `ConflictDetection`

```
class GaloisFoo {

   static final int METHOD_BAR = 1;

   public GaloisFoo(Foo delegate, ConflictDetection cd) {
      _cd = cd;
      _delegate = delegate;
   }

   public int bar(int a, int b) {

      _cd.prolog(METHOD_BAR, {a, b});
      int retval = bar(a);
      _cd.epilog({retval});

      GaloisRuntime.addUndo(/* ... */);

      return retval;
   }

   ConflictDetection _cd;
   Foo _delegate;

}
```

Figure 3.4: Generic Galois wrapper

object for two reasons: (i) to allow multiple objects which share semantics (*e.g.* a `HashSet` and a `TreeSet` in Java) to share the same commutativity properties; and (ii) to allow Galois wrappers to be instantiated with different conflict detection schemes (see Section 4.3 for an example of an alternate scheme).

Because both commutativity checks and undos rely on the semantics of objects, it is necessary for the class designer to provide this information. This is done through an *interface* specfication, which provides three pieces of information:

- *returns*: This gives a name to the return value of a method.

- *commutes*: This section specifies which other interface methods the current method commutes with, and under which conditions. For each method specified in this section, we provide a *side condition*. The two methods commute whenever the side condition evaluates to `true`. For example, `remove(x)` commutes with `add(y)` as long as the elements are different.
- *undo*: This section specifies the inverse of the current method. It is used to construct the semantic undo used in the Galois wrapper.

Figure 3.5 provides an example of this specification information for `set` objects. A few points of interest: the two read only methods (`contains` and `findRandom`) commute with all other read only methods (obviously). This is shown by having the side condition simply be `true`—the methods commute under all possible invocations. Note also that we are using simple object equality to define commutativity (rather than a deeper notion of equality such as Java's *.equals()*). While this may appear to be unsafe, it is simply based on the semantics of the set in question. Sets are *unique associative containers* (in STL parlance [121]), and hence only allow one of any particular object to be in the set. If this uniqueness is enforced by reference equality,[5] then using reference equality in the commutativity conditions is correct. If, however, this uniqueness is enforced by deeper, semantic equality, then the commutativity conditions would take this into account.

These specifications are merely a declarative statement of the semantics of a given data structure—turning these specifications into correct, efficient code is another problem entirely. In this thesis, we do not deal with a formal system for transforming specifications into conflict detection objects, but Section 3.4.4 describes how conflict detection objects are typically implemented to check com-

[5]Formally, for Java `Sets` [37], this is true when, for two elements $e1$ and $e2$ in the set, $(e1 == e2) \Leftrightarrow (e1.equals(e2))$

```
interface Set {
   void add(Object x);
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {y != x}
         - findRandom() : y {y != x} //findRandom call returning y
      [undo] remove(x)
   void remove(Object x);
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {y != x}
         - findRandom() : y {y != x}
      [undo] add(x)
   bool contains(Element x);
      [returns] bool b;
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {true} //all calls commute
         - findRandom(): y {true} //all calls commute
   Object findRandom();
      [returns] Object x;
      [commutes]
         - add(y) {y != x}
         - remove(y) {y != x}
         - contains(y) {true} //all calls commute
         - findRandom(): y {true} //all calls commute
}
```

Figure 3.5: Example commutativity specification for a `Set`

mutativity conditions.

In the applications we have looked at, most shared objects are instances of *collections*, which are variations of sets, so specifying commutativity information and writing undo methods has been straightforward.

Incremental development of Galois classes

One appealing feature of the object-oriented nature of Galois classes is that they lend themselves to incremental development as needed to improve parallelism. We provide a "baseline" conflict detection strategy, which provides no commutativity information. Thus, all methods are assumed to conflict with one another. This means that an object can be accessed by at most one iteration at a time, and that iteration shuts out other iterations until it commits. In this case, undo methods can be implemented automatically using shadow copies, as in software transactional memories, as there are no method interleavings which require semantic undos. Note that this approach is equivalent to using two-phase locking to ensure serializability.

However, if it turns out that locking out other iterations has a deleterious effect on parallel performance, a programmer can begin inserting commutativity checks incrementally, as he or she determines that two methods do, in fact, commute. This can gradually increase the parallelism afforded by an object until it reaches a satisfiable amount.

A similar approach can be taken for ensuring the consistency of a shared object (*i.e.* providing atomic methods). Because atomic methods are provided by the wrapped object, a programmer can begin with a coarse-grain locking implementation of the object (which can be be implemented in a straightforward manner using the monitor idiom). If this does not provide enough concurrency, it is easy to replace the wrapped object with a different implementation, providing the same interface, which performs fine-grain locking, or uses transactional memory.

3.3.5 Discussion

The design of the Galois class libraries exposes a multitude of avenues for further investigation and research. In this section, we briefly discuss the following issues:

- **Semantic commutativity vs. transactional memory**: A comparison between semantic commutativity and transactional memory, focusing on how the two approaches differ when detecting conflicts between concurrently executing iterations.
- **Semantic commutativity vs. open nesting**: A discussion of how semantic commutativity compares to recent proposals to augment transactional memory with open nesting [93].
- **Issues with semantic commutativity and return values**: An explanation of how to correctly handle methods whose commutativity is conditional on their return values.
- **Semantic undo vs. shadow copies**: A comparison of the Galois approach to undoing speculative execution with existing approaches.
- **Eliding wrappers and sharing** `ConflictDetection` **objects**: A description of the conditions under which multiple shared objects can share a single Galois wrapper and/or the same `ConflictDetection` object.

Semantic commutativity vs. transactional memory

By leveraging data structure semantics, semantic commutativity is strictly more precise than the read/write sets of transactional memory for detecting conflicts between concurrently executing iterations. One can view reads and writes to memory as "methods" which act on memory locations: `read(addr)` and

`write(addr)`. A single iteration can then be broken up into a (long) sequence of `read` and `write` operations. Note that the semantics of these operations are such that `read`s to a particular address commute, but no other combination of operations on a given address commute[6].

Thus, by semantic commutativity, two iterations will be declared independent if the set of addresses either writes to is disjoint from the set of addresses the other reads from and writes to. Note that this is equivalent to a transactional memory's tracking of read/write sets. Hence, any iterations that a transactional memory would declare independent would also be found independent by semantic commutativity.

This property holds even if semantic commutativity tracks higher-level methods (such as additions and deletions from a set). This is because if transactional memory sees two iterations as independent, the read/write sets generated by any methods the two iterations invoke must not overlap. If two methods have non-overlapping read/write sets, they necessarily commute with one another in a concrete sense (since they could execute in either order and produce exactly the same set of reads and writes) and hence commute in a semantic sense as well.

Concrete commutativity vs. transactional memory Interestingly, the standard approaches to transactional memory are strictly less precise than concrete commutativity. Consider the sorted linked list in Figure 3.6(a). Then consider two iterations, the first adding `W` to the list, the second adding `H`. The state of the linked list each iteration would see after executing the operation is shown in Figures 3.6(b) and 3.6(c), with the locations in their read sets shaded light grey

[6] "Silent" writes, where two `write` operations write the same value to a particular location, commute, but we disregard this for simplicity

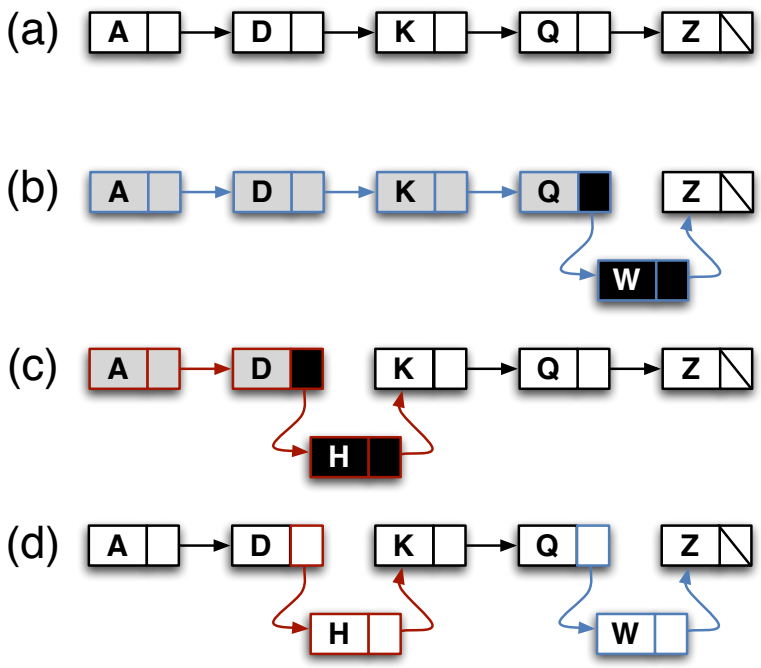


Figure 3.6: Transactional memory vs. concrete commutativity

and the locations in their write sets shaded black.

Clearly, regardless of which order the two iteration perform their additions to the list, the resulting list will be as shown in Figure 3.6(d), so the invocations commute both semantically and concretely. However, the read/write sets of both iterations conflict, and hence a transactional memory would not allow these iterations to proceed in parallel.

This particular problem is dealt with in the transactional memory literature by providing *early release* functionality [58]. This allows transactional memories to remove addresses from transactions' read-sets, in the interest of reducing conflicts. In the case of linked list traversals, it would be used to remove the first three nodes from the read-set in Figure 3.6(b), and the first node from the read-set in Figure 3.6(c). By doing so, the two invocations no longer conflict, and the iterations performing them can continue without aborting.

Early release has generally been presented as an unsafe optimization, which

can break the isolation of transactions. It is apparent from this discussion that early release in transactional memories is safe as long as it preserves concrete commutativity.

Semantic commutativity vs. open nesting

Several recent transactional memories have provided *open nesting* [89], where a nested transaction can commit even as its parent transaction maintains its read/write sets [86, 93]. This is accomplished is by maintaining separate read/write sets for open nested transactions. Rather than merging the nested transaction's sets with the parent transaction upon completion, as in closed nesting, the read/write sets are discarded, effectively committing the open nested transaction.

Open nesting allows certain amounts of information to "escape" the isolation boundary of the parent transaction, increasing concurrency. To ensure that transactional semantics are not violated, Ni *et al.* have proposed the notion of "abstract locks," to detect conflicts between open nested transactions and other transactions [93]. However, they provide no systematic methodology for using these abstract locks. In [20], Carlstrom *et al.* used abstract locks to transactionalize the Java collections classes, but likewise did not provide a general methodology for using abstract locks.

In general, open nesting is a *mechanism* for providing semantic conflict detection, not a *model* for detecting semantic conflicts. It specifies the changes that must be made to a traditional transactional memory to support semantic conflict checking, but does not provide the programming model for safely using open nesting. One can view semantic commutativity as providing such a methodology: much as early release is safe as long as it preserves concrete com-

mutativity, open nesting is safe as long as it preserves semantic commutativity. Thus, the appropriate locking protocols for open nested transactions are exactly those which enforce semantic commutativity.

Our approach to semantic commutativity is agnostic to the mechanism used to implement it. The implementation of semantic commutativity in the Galois system uses linerizable objects and semantic commutativity checks in software (later work by Herlihy and Koskinen discusses how this approach can be integrated with transactional memory [57]). However, we could also use a purely transactional approach, using open nesting and abstract locks to capture semantic commutativity.

Issues with semantic commutativity and return values

Much of the power of semantic commutativity arises due to its ability to provide conditional commutativity (*i.e.* methods commute only under certain conditions). These conditions can be based on either the method arguments, or on a method's return value. While there are no issues with having commutativity conditional on arguments, there are several problems which arise when commutativity is conditional on return values. We call methods whose commutativity is dependent on return values *return-dependent methods*.

First, an iteration executing a return-dependent method can cause other iterations to lose isolation. It is impossible to determine the safety of a return-dependent method invocation until *after* it executes. Unfortunately, this means that other iterations can see the modified object state, before the commutativity check occurs, and can thus lose isolation.

It is easiest to demonstrate this by example. Consider a shared set, which supports `add`, `contains` and `removeRandom`, with the last removing and re-

Iteration A	**Iteration B**
{	{
`s.add(x)`	`...`
`...`	`s.removeRandom() //returns x`
`s.contains(x)`	`...`
}	}

Figure 3.7: Problems with return-dependent methods

turning a random element from the set. Now consider two iterations operating on the set, performing the operations shown in Figure 3.7. Iteration B's call to `removeRandom` is unsafe, as it does not commute with Iteration A's call to `add`. However, before the commutativity check on Iteration B fails, Iteration A proceeds to call `contains`. This commutes with everything executed so far (as there is no record yet that Iteration B called `removeRandom`), so the invocation proceeds, and returns false. Thus, Iteration B has clearly broken the isolation of Iteration A.

This is not the only problem with return-dependent methods. As we will see in Section 3.3.3, undo methods that are executed when an iteration is rolled back must be able to execute safely. However, because we cannot tell whether a return-dependent method is unsafe until after it is executed, we cannot guarantee that the associated undo method can be executed safely.

We avoid this problem by requiring that all return-dependent methods be read only. This makes it impossible for an iteration to affect the isolation of another iteration, as the return-dependent method cannot modify shared state. Furthermore, no undo methods are required, addressing the issue of unsafe rollbacks. Note that this means that methods such as `removeRandom` must be implemented in two phases—a method such as `findRandom` followed by `remove`.

Iteration A	**Iteration B**
`{`	`{`
`...`	`s.findRandom() //returns x`
`s.remove(x)`	`...`
`}`	`}`

Figure 3.8: Example demonstrating race due to return-dependent methods

However, this restriction is not enough to make return-dependent methods safe. Iterations executing return-dependent methods can lose their *own* isolation, even when the methods are read only. This is due to a subtle race condition. Consider a set supporting the operations `findRandom` and `remove`, and the two iterations shown in figure 3.8. Iteration B executes `findRandom`, and prepares to perform the commutativity check. Before the check is performed, Iteration A executes `remove` (which appears to be safe, as B has not performed its commutativity check yet), and commits. Unfortunately, this means the record of A's executing `remove` no longer exists, and the ongoing commutativity check performed by B will now pass, claiming that `findRandom` is safe. Thus, iteration A caused iteration B to lose isolation. A similar problem can occur when an iteration aborts.

This problem occurs because of an "atomicity gap" between when a method completes and when its commutativity is checked; the two operations do not occur as a single atomic action. This gap allows other iterations to execute unsafe operations which can break isolation. One solution to the problem is to attempt to eliminate the atomicity gap—for example, by holding any locks acquired by a return-dependent method until after the commutativity check completes. However, this breaks the clean separation between method atomicity and isolation checking, reducing the modularity of the system. We would no longer be able

to wrap any linearizable type in an object wrapper and treat it as a Galois object. Rather, the implementation of an object becomes closely coupled with the conflict checking required for isolation.

There are several alternate solutions which do not incur this programmability penalty. First, several return-dependent methods can be "checked" after they execute, to ensure that isolation was not broken. For example, after executing `findRandom`, we can call `contains` on the returned element to ensure that it is still a valid result for `findRandom`. For return-dependent methods that do not have a simple check, we can instead re-execute the method and ensure that we produce the same result. In either case, if the check fails, we can simply treat this as a commutativity violation and trigger a roll-back.

A second solution is to track all iterations that have committed during the execution of a return-dependent method, and save their execution records. These can then be checked as part of the commutativity check of the return-dependent record, to ensure that we have not lost isolation.

Our current system uses a combination of these two approaches, performing post-execution checks when they can be calculated cheaply, and tracking commits in other cases.

Semantic undo vs. shadow copies

Other approaches to speculative parallelization do not require semantic undo methods, as we do. Instead, they use a variety of approaches to allow rollbacks:

- **Speculative caches** Many optimistic parallelization and synchronization techniques buffer speculative state in a processor's cache, only committing changes to main memory after speculation completes [52, 59, 76]. In some software transactional memories, such as [53], an equivalent soft-

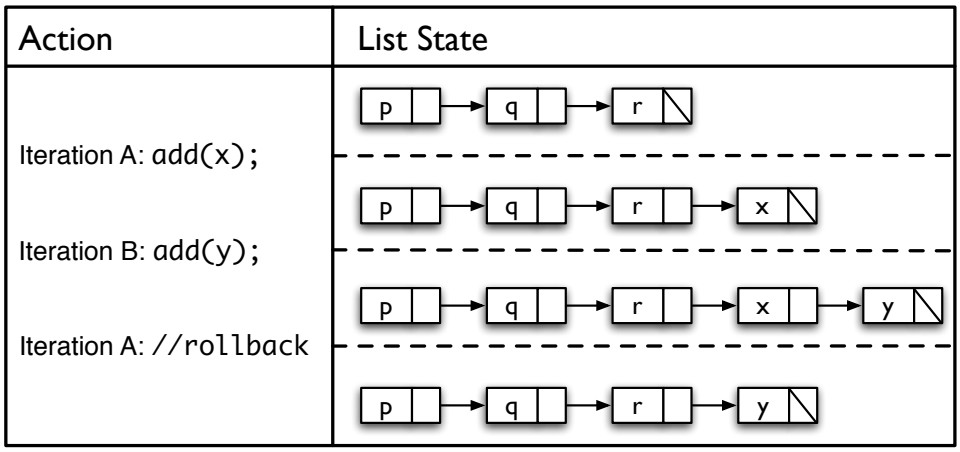


Figure 3.9: Using semantic undo for rollback

ware approach is taken. Because changes are only made public to main memory at commit time, rollback can be accomplished simply by clearing the speculative cache. In the transactional memory literature, this policy is known as "lazy update."

- **Logged writes** Other implementations [86] make changes to shared memory as speculative execution progresses, saving a log of all writes in local memory. Rollbacks are accomplished by processing the log in a last-in, first-out manner, undoing all changes to shared memory. In the transactional memory literature, this policy is known as "eager update."
- **Shadow copies** Object-based software transactional memories [58, 84] make a "shadow" copy of an object when it is first accessed speculatively. All speculative modifications to an object are made to the object, with the shadow copy representing the state of the object before speculative execution. Thus, rollback can be performed by replacing all objects with their shadow copies.

The most direct comparison to our rollback technique can be made with shadow copies. We preserve a complete record of actions to be undone (much as

in the logged writes approach), instead of simply using shadow copies. This is due to the interleavings of iterations allowed by semantic commutativity. Consider the example shown in Figure 3.9. Iteration A adds `x` to the linked list, then Iteration B adds `y`. At some point in the future, Iteration A rolls back. After rollback, the linked list should contain `y`. However, at no point during forward execution does the list exist in a state containing `y` but not `x`. Thus, there is no point where a valid shadow copy could be made—the only safe rollback mechanism is to perform a semantic undo, removing `x` from the list.

Eliding wrappers and sharing `ConflictDetection` objects

When providing Galois wrappers for a set of library classes, it is immediately apparent that not all objects require wrapping. For example, objects which are *immutable* can never be changed in an unsafe manner. These objects can be accessed directly, without performing conflict detection, as accesses to them, by definition, always commute. For example, the triangles contained in the Delaunay mesh are actually immutable objects (new triangles may be created, but existing triangles are never modified) and hence, we do not need to provide wrappers for them.

We can also provide a single wrapper for all objects which, collectively, make up the *representation* of a single data structure.[7] For example, a `Set` backed by a linked list is comprised of a series of linked nodes. Because all of these nodes are treated collectively as a single set, with a single interface, we need only provide one wrapper for the objects.

Care must be taken when eliding a wrapper, though. Because the `ConflictDetection` object in the wrapper is the location where relevant in-

[7] While we do not provide a rigorous definition of "representation," the following intuition suffices: an object is part of a data structure's representation if changes to the object affect the invariants of the data structure.

formation is kept regarding the optimistic state of shared objects, it is necessary that these logs remain consistent with respect to the shared objects they reference. In general, we require that any changes to a shared object be recorded by a single `ConflictDetection` object (*i.e.* there cannot be active invocations on an object which are seen by more than one `ConflictDetection` object). This ensures that there is a single point of reference for an object's state, and that it remains consistent.

This is a problem of controlling *representation exposure* [28], where the internal representation of an object is exposed beyond its encapsulation boundary. Intuitively, Galois wrappers and the associated `ConflictDetection` objects protect all accesses initiated at the encapsulation boundary, but have no record of accesses made at other points. So if, for example, a linked list node from the previous example is aliased and can be accessed without going through the `Set`'s wrapper, then we can no longer guarantee isolation.

A concrete example of this problem occurs in the the `ConcurrentHashMap` class in the Java collections classes [37]. This object is thread-safe, and hence can be safely wrapped in a Galois wrapper; a `ConflictDetection` object that handles maps can be used to perform commutativity checks. This object implements a method called `keySet`, which returns a set of all the keys in the map. Naïvely, this returned object would have its own wrapper, with its own `ConflictDetection` object. However, the `keySet` and the underlying `ConcurrentHashMap` share representation (removing a key from the `keySet` removes the key from the map, and *vice versa*), so having separate Galois wrappers for the two objects is unsafe. In this case, one iteration can add a key-value pair `<K, V>` to the `Map`, while a second iteration concurrently attempts to remove `K` from the `keySet`, clearly violating isolation.

We would be unable to detect this isolation violation, as the two invocations are tracked by separate `ConflictDetection` objects, and thus no commutativity check would fail. We can fix this problem by having the two object wrappers use the same `ConflictDetection` object. The general principle we adhere to in order to avoid issues with shared representation is: *objects which share representation should share conflict detection*.[8]

At the moment, this principle is simply a guideline that programmers must follow; it is not enforced or verified in any way. It may be possible to enforce the guideline by controlling aliasing through annotations [3, 15, 16, 28, 64, 94], or to verify that it is followed through program analyses that ensure similar restrictions [46, 83, 91]. We leave this to future work.

3.3.6 A small example

Iteration A	**Iteration B**	**Iteration C**
`{`	`{`	`{`
`  ...`	`  ...`	`  ...`
`  a.accumulate(5)`	`  a.accumulate(7)`	`  a.read()`
`  ...`	`  ...`	`  ...`
`}`	`}`	`}`

Figure 3.10: Example accumulator code

Consider a program written using a single shared object, an integer accumulator. The object supports two operations: `accumulate` and `read`, with the obvious semantics. It is clear that `accumulate`s commute with other `accumulate`s, and `read`s commute with other `read`s, but that `accumulate`

[8] It is important to note that this principle does not prevent multiple containers from holding the same object (for example, the Nodes in Delaunay mesh refinement appear in both the Mesh and the Worklist). This is because the objects held by containers are not part of the containers' representations; the containers' invariants depend on *immutable* state of the objects they holds (such as the address of the object).

does not commute with `read`. The methods are made atomic with a single lock which is acquired at the beginning of the method and released at the end.

There are three iterations executing concurrently, as seen in Figure 3.10. The progress of the execution is as follows:

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

In [132], von Praun *et al* discuss the use of ordered transactions in parallelizing FORTRAN-style DO-loops, and they give special treatment to reductions in such loops to avoid spurious conflicts. Reductions do not require any special treatment in the Galois approach since the programmer could just use an object like the accumulator to implement reduction.

3.4 Run-time system

The Galois run-time system comprises three global structures: a *scheduler*, which is responsible for creating iterations, an *arbitrator*, which is responsible for aborting iterations, and *commit pool*, which is responsible for committing iterations. In the baseline Galois system, the run-time also interacts with per-object *conflict logs* which are the `ConflictDetection` objects responsible for detecting commutativity violations.

At a high level, the run-time systems works as follows. The commit pool maintains an *iteration record*, shown in Figure 3.11, for each ongoing iteration in the system. The status of an iteration can be `RUNNING`, `RTC` (ready-to-commit) or `ABORTED`. Threads go to the scheduler to obtain an iteration. The scheduler 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), creates an entry for the iteration in the commit pool, and sets the status field of the iteration record to `RUNNING`. When an iteration invokes a method of a shared object, (i) the conflict log of that object and the `local_log` of the iteration record are updated, as described in more detail below, and (ii) a callback to the associated undo method is pushed onto the undo log of the iteration record. If a commutativity conflict is detected, the arbitrator 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.

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

Figure 3.11: Iteration record maintained by run-time system

3.4.1 Scheduler

The first component of the manager is the *scheduler* object, whose job it is to assign work to threads as they need it. In the default Galois system, the scheduler simply assigns work randomly from the worklist to threads. However, it may be beneficial to use more intelligent scheduling policies to improve performance. In this case, the scheduler object can be replaced with a different scheduler more appropriate to the application. The various intricacies of choosing an appropriate scheduling policy are discussed in detail in Chapter 5.

3.4.2 Arbitrator

The second component of the manager is the *arbitrator*, an object whose job 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 from a correctness perspective. There have been several policies proposed in the transactional memory literature for choosing which transaction to roll back when faced with a conflict. These policies may utilize one of several metrics, including the age of the conflicting transactions, which transaction has a larger memory footprint, or random selection [115]. Conceptually, there is no obstacle

to implementing similar contention management policies in the Galois run-time (it simply requires a different arbitrator). However, in our experience we have not found it necessary to use anything other than the Galois default policy: the arbitrator rolls back the iteration which detected the conflict.

Unlike in the ordered case, when iterating over an ordered set the default arbitration policy raises the possibility of deadlock: if iteration A and B conflict and the higher priority iteration, A, is rolled back, B still cannot commit (as that will break sequential semantics). Unfortunately, when A re-executes it will still conflict with B and the default arbitrator will roll back A again. Thus, no forward progress will be made, and the system will deadlock.

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

This situation is complicated when i_2 is the iteration that must be rolled back. Because the Galois run time systems functions purely at the user level, there is no simple way to abort an iteration running on another thread. To address this problem, each iteration record has an *iteration lock* as shown in Figure 3.11. When invoking methods on shared objects, each iteration must own the iteration lock in its record. Thus, the thread running i_1 does the following:

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

4. It then resumes its execution of i_1, which can now proceed as the conflict has been resolved.

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

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

3.4.3 Commit pool

The isolation property of transactions means that they can be serialized. The observed serial schedule that they represent is determined by the order in which the transactions commit. When iterating over an unordered set, because the

order of iterations does not matter, transactions can commit in any order. However, when iterating over an ordered set, there is a specific serial order that must be respected (*e.g.* when iterating over a priority queue, the observed serial execution must be in priority order). Thus, transactions can no longer commit in any order. It is the responsibility of the *commit pool* to ensure that transactions commit in the appropriate order.

Intuitively, the commit pool functions much as a *reorder buffer* in a modern out-of-order execution (OOE) processor [56]. In an OOE processor, instructions are executed out of order, but can only be retired, and hence have their results committed, in order; the reorder buffer allows this mix of out-of-order execution and in-order committing to happen. Similarly, when pulling iterations from an ordered set iterator, the scheduler is given the freedom to execute iterations in any order, and even to complete in any order (and hence a lower priority iteration can execute completely even while a higher priority iteration is still running). However, the commit pool ensures that iterations only commit their state (and hence release isolation) in order.

The commit pool contains a queue, called the *commit queue* which is a list of all iterations current in the `RUNNING`, `ABORTED` or `RTC` states, sorted by priority. Thus, the highest priority `RUNNING` iteration is at the head of the queue.

When an iteration attempts to commit, the commit pool checks two things: (i) that the iteration is at the head of the commit queue, and (ii) that the priority of the iteration is higher than all the elements left in the set/poSet being iterated over[9]. 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

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

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. This can be done efficiently by scanning forward through the commit queue. 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.

3.4.4 Conflict logs

The *conflict log* is the implementation of a `ConflictDetection` object (see Section 3.3.4) which performs commutativity checks. In general, we do not prescribe a particular implementation of commutativity checks—some objects may have semantics which lend themselves to more efficient implementations than others—but here we describe a typical implementation.

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.

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

As an optimization, each iteration caches its own portion of the conflict logs in a private log called its `local_log`. This local log stores a record of all the methods the iteration has successfully invoked on the object. When an iteration makes a call, it first checks its local log. If this local log indicates that the invocation will succeed (either because that same method has been called before or other methods, whose commutativity implies that the current method also commutes, have been called before[10]), the iteration does not need to check the object's conflict log.

3.5 Case studies

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

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

3.5.1 Delaunay mesh refinement

We first wrote a sequential Delaunay mesh refinement program without locks, threads etc. to serve as a *reference* implementation. We then implemented a Galois version (which we call *meshgen*), and a fine-grain locking version (*FGL*) that uses locks on individual triangles. The Galois version uses the set iterator, and the run-time system described in Section 3.4. In all three implementations, the mesh was represented by a graph that was implemented as a set of triangles, where each triangle maintained a set of its neighbors. This is essentially the same as the standard adjacency list representation of graphs. For meshgen, code for commutativity checks was added by hand to this graph class; ultimately, we would like to generate this code automatically from high level commutativity specifications like those in Figure 3.5. We used an STL queue to implement the workset [121]. We refer to these default implementations of meshgen and FGL as *meshgen(d)* and *FGL(d)*.

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

The input data set was generated automatically using Jonathan Shewchuk's Triangle program [119]. It had 10,156 triangles and boundary segments, of which 4,837 triangles were bad.

Execution times and speed-ups. Execution times and self-relative speed-ups for the five implementations on the Itanium machine are shown in Figures 3.12 and 3.13 respectively. The reference version is the fastest on a single processor. On 4 processors, *FGL(d)* and *FGL(r)* differ only slightly in performance. *mesh-*

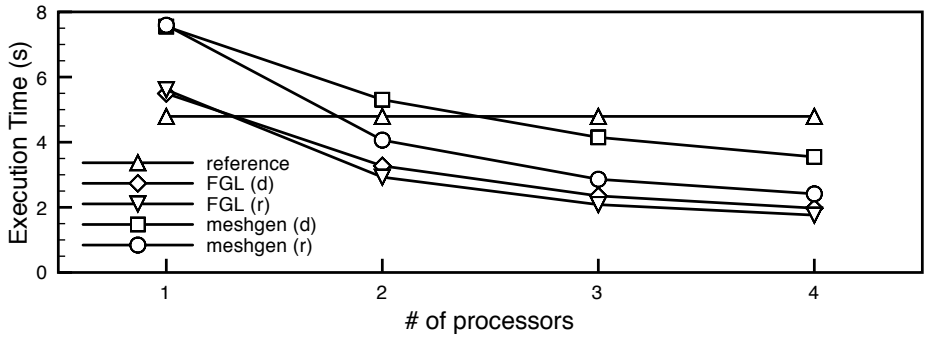


Figure 3.12: Mesh Refinement: execution times

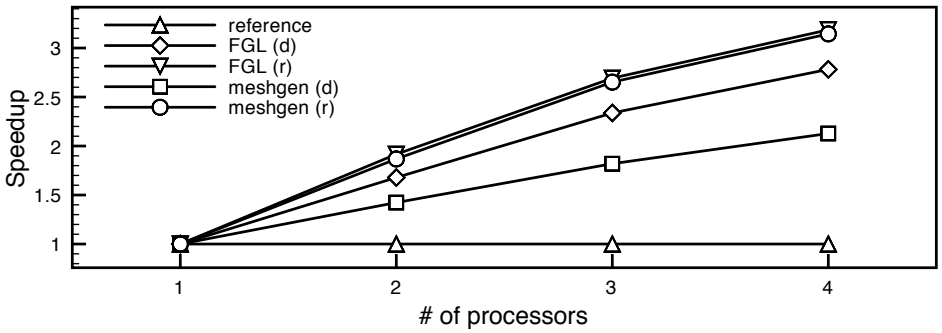


Figure 3.13: Mesh refinement: self-relative speed-ups

gen(r) performed almost as well as *FGL*, although surprisingly, *meshgen(d)* was twice as slow as *FGL*.

Table 3.1: Mesh refinement: committed and aborted iterations for meshgen

	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

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 Table 3.1. On 1 processor, *meshgen* executed and committed 21,918 iterations. Because of the inherent nondeterminism of the set iterator, the number of iterations executed by meshgen in parallel varies from run to run (the same effect will be seen on one processor if the scheduling policy is varied). Therefore, we ran the codes a large number of times, and determined a distribution for the numbers of committed and aborted iterations. Table 3.1 shows that on 4 processors, *meshgen(d)* committed roughly the same number of iterations as it did on 1 processor, but also aborted almost as many iterations due to cavity conflicts. The abort ratio for *meshgen(r)* is much lower because the scheduling policy reduces the likelihood of conflicts between processors. This accounts for the performance difference between *meshgen(d)* and *meshgen(r)*. Because the *FGL* code is carefully tuned by hand, the cost of an aborted iteration is substantially less than the corresponding cost in meshgen, so *FGL(r)* performs only a little better than *FGL(d)*.

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

One conclusion from these experiments is that domain knowledge is invaluable for implementing a good scheduling policy. We present a deeper investiga-

Table 3.2: Mesh refinement: instructions per iteration on a single processor

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

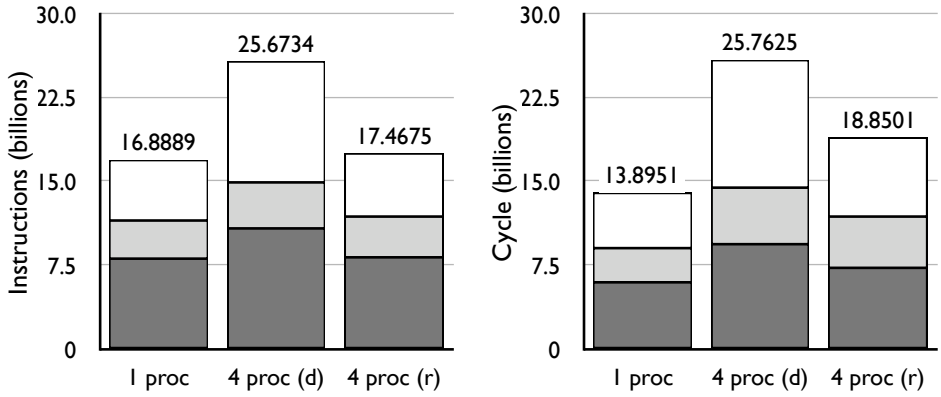


Figure 3.14: Mesh refinement: breakdown of instructions and cycles in meshgen

tion of scheduling policies for Dealaunay mesh refinement in Chapter 5.

Instructions and cycles breakdown. Table 3.2 shows the breakdown of different types of instructions executed by the reference and meshgen versions of Delaunay mesh refinement when they are run on one processor. The numbers shown are per iteration; in sequential execution, there are no aborts, so these numbers give a profile of a "typical" iteration in the two codes. Each iteration of meshgen performs roughly 10,000 floating-point operations and executes almost a million instructions. These are relatively long-running computations.

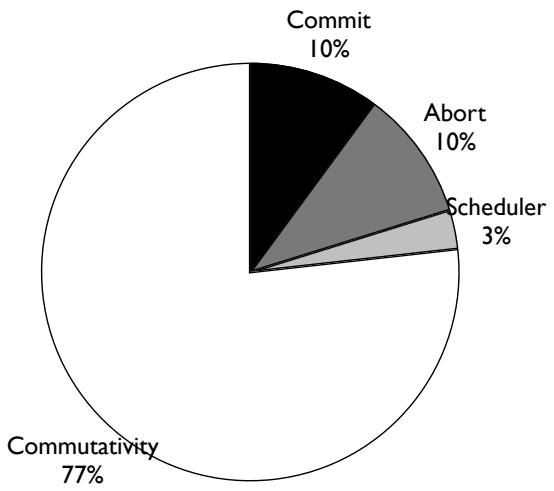


Figure 3.15: Mesh refinement: breakdown of Galois overhead

Meshgen executes almost 80% more instructions than the reference version. To understand where these extra cycles were being spent, we instrumented the code using the Performance Application Programming Interface (PAPI) [18]. Figure 3.14 shows a breakdown of the total number of instructions and cycles between the client code (the code in Figure 3.2), the shared objects (graph and workset), and the Galois run-time system. The 4 processor numbers are sums across all four processors. The reference version performs almost 9.8 billion instructions, and this is roughly the same as the number of instructions executed in the client code and shared objects in the 1 processor version of meshgen and the 4 processor version of *meshgen(r)*. Because *meshgen(d)* has a lot of aborts, it spends substantially more time in the client code doing work that gets aborted and in the run-time layer to recover from aborts.

We further broke down the Galois overhead into four categories: commit and abort overheads, which are the time spent committing iterations and aborting them, respectively; scheduler overhead, which includes time spent arbitrating conflicts; and commutativity overhead, which is the time spent performing

Table 3.3: Mesh refinement: L3 misses (in millions) for meshgen(r)

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

conflict checks. The results, as seen in Figure 3.15, show that roughly three fourths of the Galois overhead goes in performing commutativity checks. It is clear that reducing this overhead is key to reducing the overall overhead of the Galois run-time.

The 1 processor version of *meshgen* executes roughly the same number of instructions as the 4 processor version. We do not get perfect self-relative speedup because some of these instructions take longer to execute in the 4 processor version than in the 1 processor version. There are two reasons for this: contention for locks in shared objects and the run-time system, and cache misses due to invalidations. Contention is difficult to measure directly, so we looked at cache misses instead. On the 4 processor Itanium, there is no shared cache, so we measured L3 cache misses. Table 3.3 shows L3 misses; the 4 processor numbers are sums across all processors for *meshgen(r)*. Most of the increase in cache misses arises from code in the shared object classes and in the Galois run-time. An L3 miss costs roughly 300 cycles on the Itanium, so it can be seen that over half of the extra cycles executed by the 4 processor version, when compared to the 1 processor version, are lost in L3 misses. The rest of the extra cycles are lost in contention.

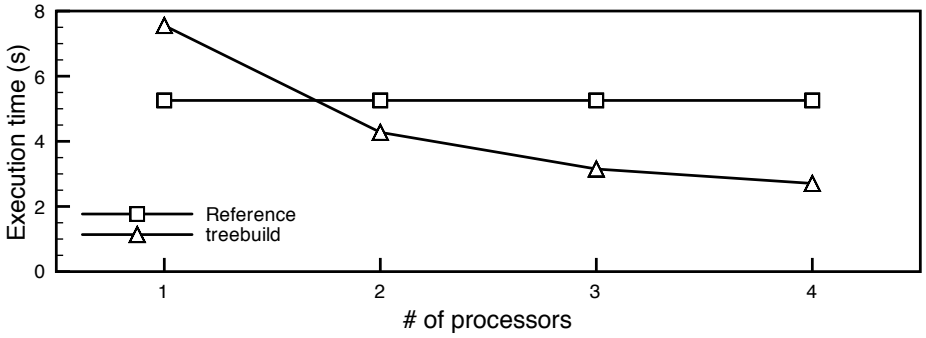


Figure 3.16: Agglomerative clustering: execution times

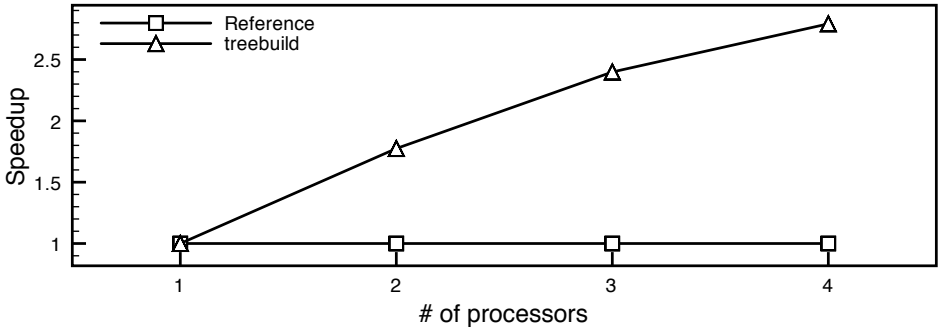


Figure 3.17: Agglomerative clustering: self-relative speed-ups

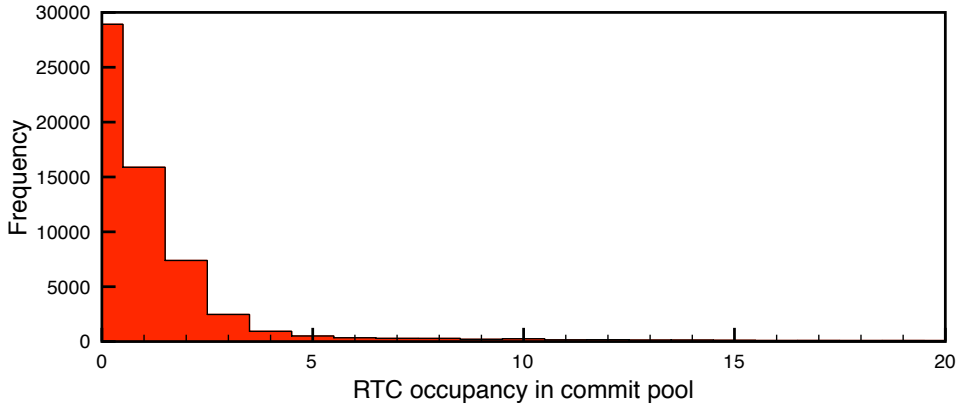


Figure 3.18: Agglomerative clustering: commit pool occupancy by `RTC` iterations

Table 3.4: Agglomerative clustering: committed and aborted iterations in treebuild

	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

Table 3.5: Agglomerative clustering: instructions per iteration on a single processor

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

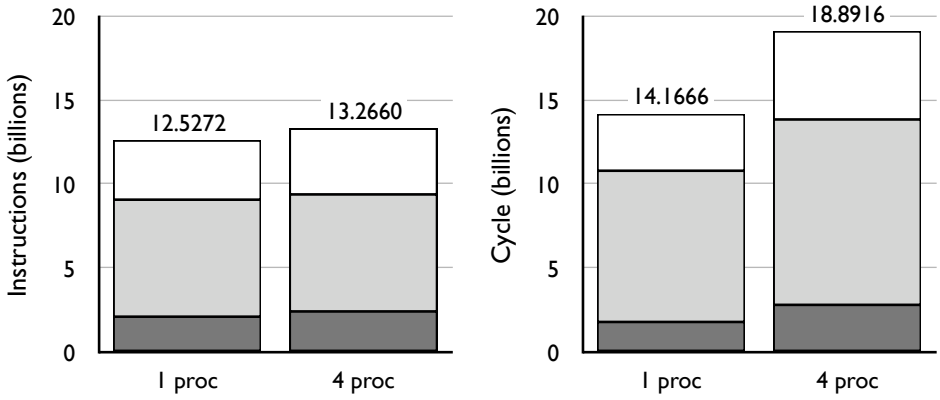


Figure 3.19: Agglomerative clustering: breakdown of instructions and cycles

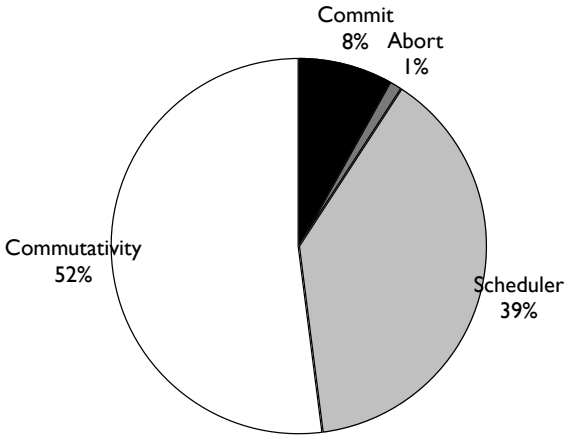


Figure 3.20: Agglomerative clustering: breakdown of Galois overhead

Table 3.6: Agglomerative clustering: L3 misses (in millions)

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

3.5.2 Priority queue-based 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 an ordered set. 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 [133]. This code builds a light hierarchy based on a distance metric that factors in Euclidean distance, light intensity and light direction. We modified the objects used in the light clustering code to use Galois interfaces and

the ordered set iterator for tree construction. The overall structure of the resulting code was discussed in Figure 2.8. We will refer to this Galois version as *treebuild*. We compared the running time of *treebuild* against a *reference* version which performed no threading or locking.

Figures 3.16–3.20 and Tables 3.4–3.6 show the results on the Itanium machine. These results are similar to the Delaunay mesh generation results discussed in Section 3.5.1, so we describe only the points of note. The self-relative speed-ups in Figure 3.17 shows that despite the serial dependence order imposed by the priority queue, the Galois system is able to expose a significant amount of parallelism. The mechanism that allows us to do this is the commit pool, which allows threads to begin execution of iterations even if earlier iterations have yet to commit.

To understand the role of the commit pool quantitatively, we recorded the number of iterations in `RTC` state every time the commit pool created, aborted or committed an iteration. This gives an idea of how deeply into the ordered set we are speculating to keep all the processors busy. Figure 3.18 shows a histogram of this information (the x-axis is truncated to reveal detail around the origin). We see that most of the time, we do not need to speculate too deeply. However, on occasion, we must speculate over 100 elements deep into the ordered set to continue making forward progress. Despite this deep speculation, the number of aborted iterations is relatively small because of the high level of parallelism in this application, as discussed in Section 2.3.1. We present a further study of commit pool behavior in Section 3.5.2.

Note that commit pool occupancy is not the same as parallelism in the problem because we create iteration records in the commit pool only when a thread needs work; the priority queue is distinct from the commit pool. We also see

that due to the overhead of managing the commit pool, the scheduler accounts for a significant percentage of the overall Galois overhead, as seen in Figure 3.20.

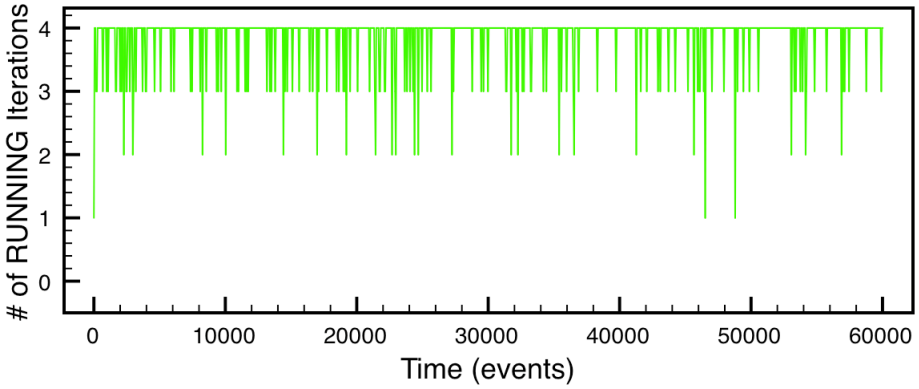
Table 3.6 shows that most of the loss in self-relative speedup when executing on 4 processors is due to increased L3 cache misses from cache-line invalidations.

Commit pool occupancy

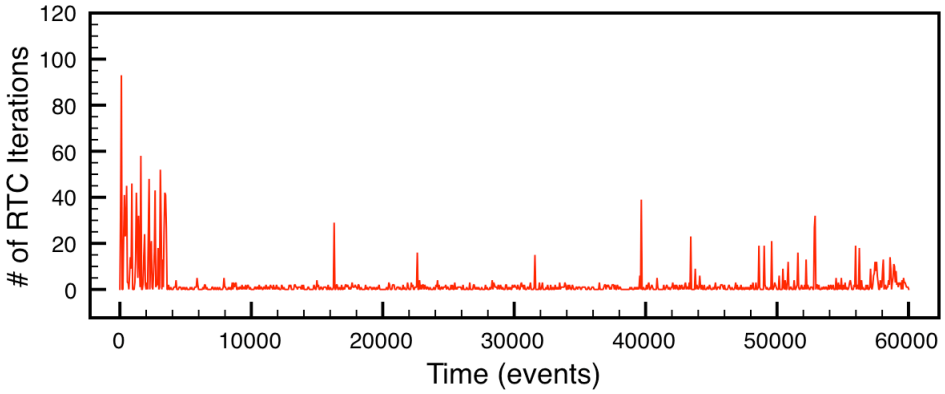
We performed a further study of the commit pool, to investigate its effects on parallelism of ordered-set programs. We instrumented the Galois run-time to report commit pool occupancy at every "event," where events are defined as iterations attempting to commit. This instrumentation tracked the number of iterations in the `RUNNING`, `RTC` and `ABORTED` states. To gather data, we ran agglomerative clustering on four cores; the results of this study are shown in Figures 3.21(a)-(c).

First, consider the number of running iterations in the commit pool (Figure 3.21(a)). As we would expect, there are fairly consistently four iterations in the RUNNING state, indicating that the system is able to maintain parallel execution throughout the program run (even if it is not always beneficial, as we shall see shortly). Figure 3.21(b)) shows the number of `RTC` iterations in the commit pool over time. This reflects the histogram results shown in Figure 3.18; most of the time there are not too many RTC iterations in the commit pool, but occasionally we have to speculate quite deeply into the iteration space in order to find work to do.

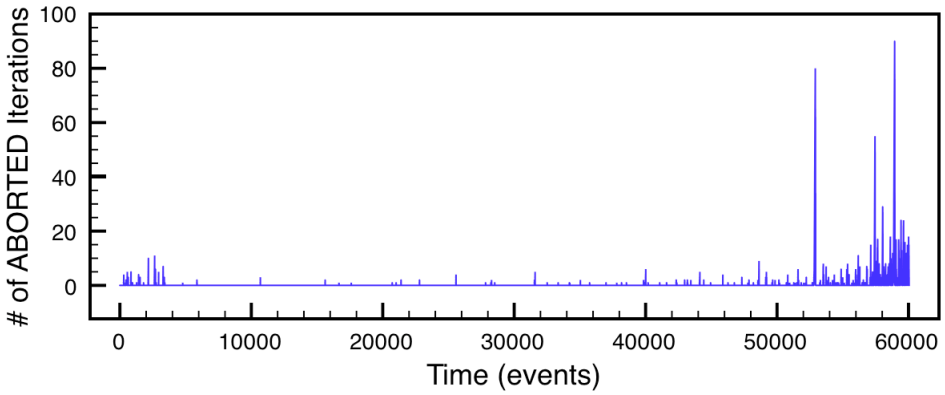
Interestingly, even when the degree of speculation gets very high (as it does towards the beginning of execution), the number of aborts that we see does not increase significantly, as we can see in Figure 3.21(c). This means that the high



(a) RUNNING iterations in commit pool



(b) RTC iterations in commit pool



(c) ABORTED iterations in commit pool

Figure 3.21: Commit pool occupancy over time

degree of speculation is warranted; we are able to continue making forward progress even while higher priority iterations stall. However, towards the end of execution, this ceases to be the case: the number of aborted iterations spikes. This is to be expected, as the number of potential clusters to look at shrinks, increasing the likelihood of conflict.

Clearly the commit pool is a useful structure when executing ordered-set iterators. Without it, we would not be able to speculate beyond higher priority iterations to find useful work to do. We once again draw an analogy to a reorder buffer. In an in-order processor, when a single instruction (*e.g.* a multiply) is high latency, execution must stall until the instruction completes. However, with the use of a reorder buffer, other, independent iterations from later in the instruction stream can execute to hide the latency of earlier instructions, maintaining performance. Similarly, the commit pool allows later iterations to execute to account for slow, high priority iterations, maintaining concurrency.

3.5.3 Performance on 4-core Xeon

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

Table 3.7 shows the performance of the two programs when run on a single core and on two cores. We see that when the two cores are on the same package,

Table 3.7: Results on dual-core, dual-processor Intel Xeon

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

we achieve near-perfect speedup, but the speedup is much less when the two cores are on separate packages. This confirms that a substantial portion of efficiency loss arises from cache line invalidations due to data sharing, so further improvements in performance require attending to locality.

3.6 Summary

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

use collections of various sorts.

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

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

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

CHAPTER 4

PARTITIONING FOR PERFORMANCE

4.1 Overview

The Galois system as presented in the previous chapter allows for the optimistic parallelization of programs exhibiting amorphous data-parallelism. However, while it can successfully parallelize these programs, it was not developed for performance or scalability. In this chapter, we investigate approaches to improving the scalability of the Galois system.

4.1.1 Scalability issues

There are several issues to consider when pursuing scalability in any parallel system. The first is *locality*: a program should maintain good cache locality, even as its data is shared among multiple processors. If a program can be parallelized but cache locality is destroyed, it is very difficult to achieve acceptable performance.

A second issue constraining scalability is *contention*. In a shared memory parallel program, there can be several data structures which are shared between multiple processors. As such, access to these shared structures must be synchronized to ensure correct behavior, and this synchronization often enforces mutual exclusion. As a result, contention for shared resources can lead to excessive serialization of execution, dramatically reducing scalability (*cf.* Amdahl's Law [4]).

While the previous two issues affect all parallel programs, when dealing with optimistic parallelization, we must contend with two further issues which can impact scalability. First, we must consider the effects of *mis-speculation*. Essentially, in order to obtain the benefits of optimistic parallelism, the optimism

must be warranted. If most speculative execution must be rolled back, then there is little effective parallelism in the program. Achieving scalability thus requires minimizing the amount of mis-speculation in optimistic execution.

The final obstacle to scalability arises because optimistic parallelization systems must perform run-time checks to detect when speculative execution is correct. These checks can be a source of significant overhead—in the baseline Galois system, they account for 75% of the overall run-time overhead, as we saw in Section 3.5.1—and often create serial bottlenecks. Thus, minimizing the overhead of run-time dependence checks is necessary to achieve reasonable performance.

4.1.2 Locality vs. parallelism

In many data-parallel applications, there exists a fundamental tension between maintaining good cache locality and achieving high levels of parallelism. Achieving locality requires scheduling execution such that iterations which access the same regions of the data structures are run in close temporal proximity. This schedule of execution will promote temporal locality. However, if that schedule of execution is blindly applied to parallel execution, it is highly likely that iterations executing simultaneously on separate processors will conflict with one another.

Delaunay mesh refinement provides an illustrative example of this. In mesh refinement, work is newly created when a retriangulated cavity contains new bad triangles. Processing these new bad triangles will obviously access the same regions of the mesh as the iteration that generated them. Thus, the best sequential schedule for computation is to treat the worklist as a stack. As a result, any newly created work will be executed immediately, leading to excellent temporal

Table 4.1: Performance of random worklist vs. stack-based worklist for Delaunay mesh refinement

	Random		Stack-based	
# of cores	Time (s)	Abort rate (%)	Time (s)	Abort rate (%)
1	18.438	—	14.437	—
2	10.268	0.031	9.847	76.440
4	7.682	0.073	9.563	85.674

locality.

Unfortunately, if we naïvely use the same stack-based worklist in a parallel setting, we find that performance will suffer due to high amounts of mis-speculation. This is because when new bad triangles are created through a cavity's retriangulation, they are necessarily near each other in the mesh. Furthermore, these bad triangles are all added to the stack at the same time and hence will be adjacent in the worklist. Thus, when multiple processors retrieve new work from the worklist they will likely receive triangles that are nearby on the mesh. As a result, the likelihood that two processors will process triangles whose cavities overlap will be high, leading to significant mis-speculation.

To avoid this problem, the default scheduling policy of the Galois run-time is to assign work to processors at random from the worklist. This has the benefit of reducing the likelihood of mis-speculation, as processors will be working on triangles from all over the mesh, but at the cost of cache locality, as we can no longer exploit the temporal locality afforded by the stack.

Table 4.1 shows the execution time and abort rates for mesh refinement using both the stack-based worklist and the random worklist. Figure 4.1 shows the same results pictorially. We see that using the random worklist runs 27% slower than the stack-based worklist on a single core, due to the lack of locality.

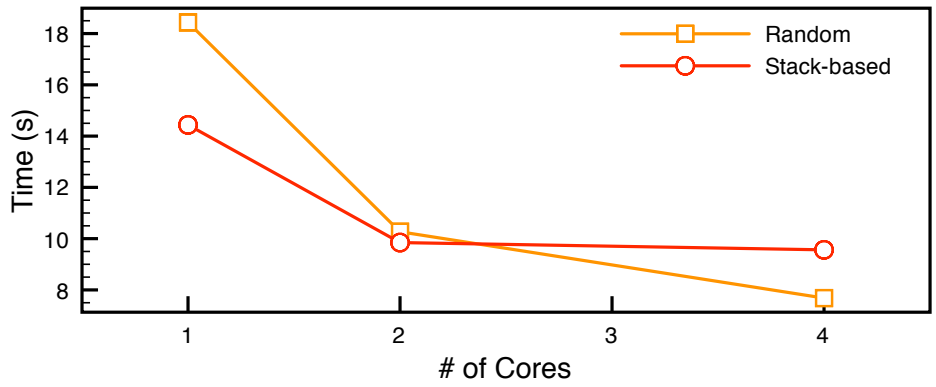


Figure 4.1: Execution time of random worklist vs. stack-based worklist for Delaunay mesh refinement

However, on four cores, the stack-based worklist has an abort rate of over 85%, while the random worklist has an abort rate of nearly 0%. The better speculation afforded by the random worklist thus allows it to outperform the stack-based worklist.

Interestingly, cache coherence can also lead to degradation of performance. If a particular region of the mesh is accessed by multiple cores, each core will attempt to cache triangles from that region. Unfortunately, the mesh changes throughout execution, so that region will be continuously written to by each core. This can result in a significant number of cache invalidations as newly written regions of the mesh ping-pong between the cores that are accessing them. We have some experimental evidence that this behavior does, indeed, affect the performance of mesh refinement, as seen in Section 3.5.3. Eliminating these excessive coherence events can be seen as improving *inter-core locality*: the tendency of each core to access disjoint regions of memory.

4.1.3 Achieving scalability

Clearly, the problem of achieving scalability in an optimistic parallelization system is difficult. Beyond the standard scalability issues of locality and contention, we must also contend with the problems of mis-speculation and run-time overheads. Furthermore, as the preceding discussion elucidates, these scalability issues interact in ways which make it difficult to address them all simultaneously.

In this chapter, 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 chapter is organized as follows. In Section 4.2, 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 4.3, we discuss how we reduce the overhead of conflict detection. Section 4.4 discusses how programmers can take advantage of the partitioning capabilities of the Galois system. In Section 4.5, 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) [17], a graph-cuts code that uses the preflow-push algorithm [42], and agglomerative clustering [125]. For each application, we describe the algorithm and key data structures as well as opportunities for exploiting parallelism and data partitioning. We summarize in Section 4.6.

4.2 Partitioning

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. This requires distributing data to cores and assigning work in a way that sustains cache performance. We target not only positive cache effects such as temporal locality, but also negative cache effects, such as invalidations caused by data sharing.

In programs that operate over regular data structures such as dense matrices and arrays, this is relatively straightforward. Languages such as HPF [75, 110] and ZPL [22] leverage small amounts of user-direction to distribute data structures among multiple processors and schedule computation in a way that maintains cache locality while providing parallelism. This problem is significantly more difficult for irregular data structures such as trees, lists and graphs.

To see why this is the case, we draw a distinction between *semantic locality*, which is the locality inherent to data structure semantics (for example, neighboring cells in a matrix, or connected nodes in a graph are semantically local) and actual locality, which is related to how the data structure is laid out in memory. Regular data structures have a close connection between semantic locality and actual locality. The semantics of matrix and array indices perfectly correspond with the actual locality inherent in the data structures' representation. It is thus apparent to see how to achieve locality in a parallel setting: ensure that iterations which touch the same indices of arrays and matrices are assigned to the same processor.

Unfortunately, with irregular data structures there is often very little correspondence between their semantic locality and their actual locality. There is clearly little spatial locality; neighboring nodes in a graph may be allocated to completely different regions of memory. But it can also be hard to find temporal locality: due to the variety of ways in which the nodes can be accessed (directly; through a chain of neighbors, etc.) it is hard to tell which iterations access the same regions of a data structure. It is thus difficult to simply examine a program and determine how to distribute data and computation to ensure that locality is maintained. This problem is compounded by the fact that there are numerous irregular data structures, each with differing definitions of semantic locality.

By way of example, consider a graph data structure, and iterations which each examine a given node and then travel to its neighbors. There is obviously locality exhibited by this program, as a node and its semantically local neighbors are accessed together. If it were possible to determine which iterations accessed a particular region of the graph, they could be assigned to the same core, improving temporal and inter-core locality. Unfortunately, making this transformation requires understanding the semantics of the graph data structure, and in particular its locality properties.

Our goal, then, is to determine how programmers can expose the semantic locality inherent in irregular data structures, and then leverage that information to improve cache performance. We do this by introducing *data and computation partitioning* to the Galois system, where a programmer can specify partitioning information for the irregular data structures used by a program. This partitioning information captures data-structure specific semantic locality in a more general form, as discussed in Section 4.2.2. By providing this information to a compiler or run-time system, we can effectively distribute irregular data structures among multiple processors and assign work to those processors in a way that ensures locality.

In this section, we describe how this data and computation partitioning is done to promote inter-core locality. As an extra benefit, this partitioning can also reduce the probability of speculative conflicts.

Figure 4.2 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 2.4. 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 4.2.1, 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 [75]. Section 4.2.2 describes the mapping of data structures to abstract domains. Finally, Section 4.2.3 describes how the run-time system performs data-centric assignment of work to cores.

4.2.1 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 4.2, 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 4.3.

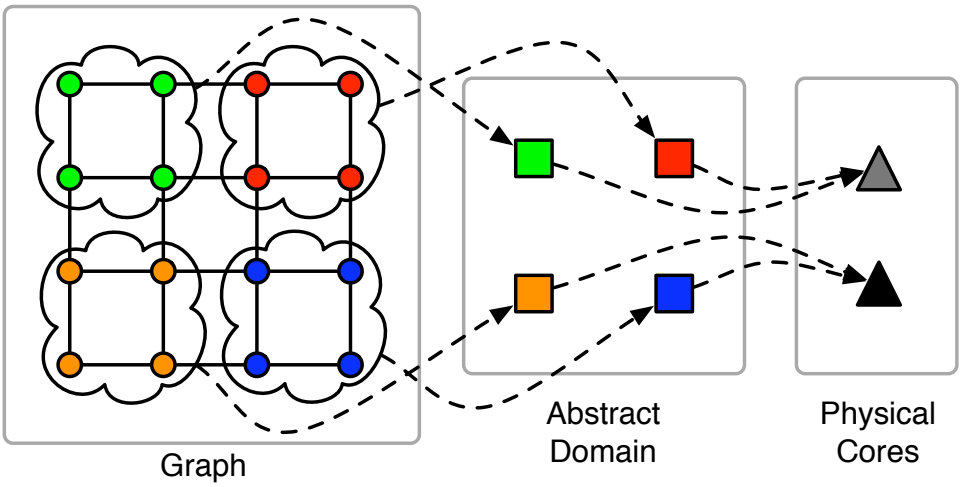


Figure 4.2: Data partitioning in the Galois system

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). The benefits of this topology are discussed in detail in Section 4.2.2.

4.2.2 Data partitioning

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

Logical 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 4.2.

Logical partitioning makes explicit the latent semantic locality properties of

data structures. For example, the semantic locality of a mesh in Delaunay mesh refinement (where neighboring triangles are semantically local) is captured by placing contiguous regions of the mesh into a single partition. We have thus transformed a data-structure specific locality property (determined by which triangles an iteration touches) into a more general locality property (determined by which *partitions* an iteration touches).

This information can be leveraged by the Galois run-time in many ways. For example, the run time can be used to perform data-centric scheduling of iterations in Delaunay mesh refinement. When a core goes to the run-time to get a bad triangle to work on, the scheduler can examine the worklist of bad triangles and return a bad triangle mapped to that core. Because mesh partitions are contiguous regions of the mesh, cores 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 run-time system need to be modified to implement this approach. This transformation is discussed in further detail in Section 4.2.3

Physical partitioning

Physical partitioning takes the logical partitioning 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 which can lead to 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. Because we are in a cache-coherent shared memory setting, every processor can access every partition of a data structure without any explicit communication.

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 4.5 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 [72, 74, 120]. 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*) [74]. At the other extreme in complexity are spectral methods that perform eigenvalue computations to determine good

graph partitions [120]. 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 4.3. 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.

Dynamic data structures

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.

Leveraging the topology of abstract domains

If an abstract domain specifies a certain topology, this information can be exploited to achieve even better locality. The abstract domain's topology can be used to capture the relationship between different partitions of a data structure. Figure 4.2 demonstrates this: partitions of the graph that are adjacent to one another are assigned to abstract processors that are adjacent in the abstract domain's grid topology. Much as the grouping of data structure elements into partitions captured the semantic locality of elements, the choice of particular abstract processors to assign partitions to captures the locality relationship between partitions.

An abstract domain's topology can then be used by the run-time when determining how to map abstract processors to physical cores. For example, if the abstract domain has a grid topology, and more abstract processors than cores, adjacent abstract processors are mapped to the same physical core (as in Figure 4.2). In essence, the abstract domain is also partitioned, and this second level of partitioning exploited, to improve locality.

Taking advantage of the multi-level partitioning afforded by an abstract domain's topology can also be useful in the presence of hierarchical architectures with multiple levels of shared and private caches. Adjacent abstract processors can be assigned to physical cores which share caches at some level of the hierarchy, increasing the likelihood that operations which access both partitions will still exhibit some cache locality.

Baskaran *et al.* looked at multi-level tiling of *regular programs* for multi-level parallel architectures (such as the Cell processor) [10]. Leveraging the abstract

domain construct allows us to perform similar multi-level "tiling" for irregular data structures in a general way.

4.2.3 Computation partitioning

Combining data structure partitioning with the topology of an abstract domain allows a programmer to capture the locality information inherent in an irregular data structure. This information is then exposed to the run-time, which can utilize it in a number of ways. The first is by carefully mapping abstract processors to physical cores, as described earlier. The second is to take the semantic locality captured by the partitioning and to turn it into temporal locality.

To do this, we 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 worklist, 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. We explore a number of alternate scheduling policies in Chapter 5.

Related work

Bai *et al.* examined a similar computation partitioning approach when dealing with multiple transactions performing operations on a hash table [9]. Their application- and data structure-specific approach examined the keys that incoming transactions operate on and dynamically assigned transactions to processors based on a run-time partitioning of the key space. Our approach, while relying on a user-specified partitioning rather than a run-time partitioning, is significantly more general, as it performs computation partitioning for any application which uses partitioned data structures.

In the context of task-parallelism, Chen *et al*. [24] 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.

Philbin *et al.* transformed sequential, loop-based programs into fine-grained parallel programs and used a similar computation partitioning approach to re-

order iterations and improve cache locality [98]. Their approach required the determination of which memory locations a loop iteration accessed; they restricted themselves to programs accessing dense matrices for this reason. It may be possible to apply our partitioning approach to extend their technique to irregular programs.

4.3 Reducing conflict detection overhead

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.

The key insight is that, while commutativity checks capture the necessary and sufficient restrictions on which methods can be invoked by simultaneously executing iterations, we can "relax" the commutativity checks to express sufficient, but overly restrictive, conditions. Iterations executing under the relaxed conditions remain isolated, but we will lose precision in our conflict detection, and trigger "false positive" conflicts.

Consider, for example, the method `add(x)` in a set. Under normal Galois commutativity, `add(x)` would commute with `add(y)`, provided that `y` and `x` are distinct. Under "relaxed" commutativity, `add(x)` commutes with `add(y)` when `x` is *in a different partition* than `y`.

This is still a sufficient condition (because if two invocations commute under the relaxed conditions, they clearly still commute under the more precise con-

dition). However, because the partitions are much coarser granularity than individual elements of the set, the coarsened conditions will allow fewer method calls to execute concurrently, restricting parallelism. On the other hand, we can support these partition-based conditions with very low overhead, as we see below.

4.3.1 Partition locks

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 run-time 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.

Restrictions on lock coarsening

Lock coarsening cannot always be performed. For partition locking to be a safe implementation of relaxed commutativity, the relaxed conditions must consider only the equality or inequality of sets of partitions. However, not all commutativity conditions can be correctly relaxed to obtain this form.

Consider the kd-tree of agglomerative clustering [11]. It supports a `findNearest(x)` method, which takes as an argument a cluster `x` and returns the nearest cluster, `y` in the kd-tree, as well as a method `add(z)` which inserts a new cluster into the tree. The commutativity relation between the two methods is as follows: `findNearest(x)` commutes with `add(z)` as long as

the cluster `z` inserted by `add` is not closer to `x` than the cluster `y` (returned by `findNearest`) is. This condition cannot be expressed in terms of the partitions that `x`, `y` and `z` lie in, and hence there is no relaxed commutativity condition that both safely captures the actual commutativity properties of the kd-tree and can be implemented using partition locking.

4.3.2 Overdecomposition

While lock coarsening can lead to a significant improvement in run-time 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.

Consider a problem being run with two threads and two logical processors. If an iteration from the first thread accesses both logical processors, it will control the entire data structure. No other iteration can be started by the second thread, as it will immediately find that it cannot acquire the necessary logical processor. In general, if many iterations cross partition boundaries, the system can experience reduced core utilization, and its effective parallelism is constrained.

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 suf-

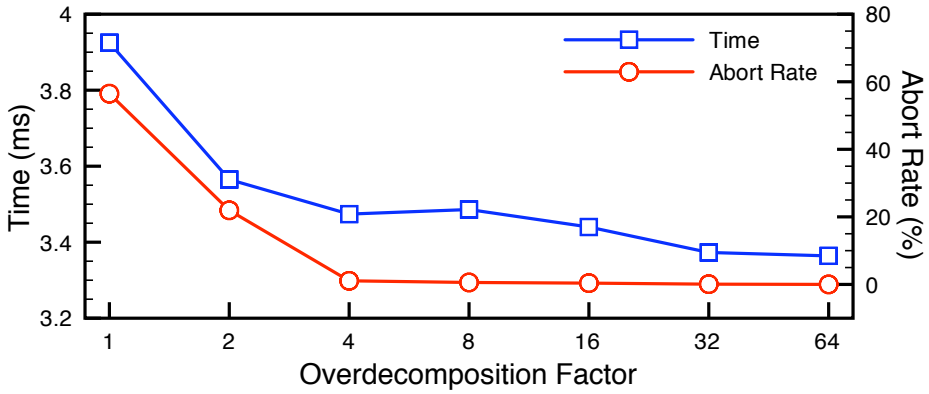


Figure 4.3: Relationship between overdecomposition factor and performance

ficiently 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.

To gain some insight on the effect of overdecomposition on performance, we investigated the behavior of Delaunay mesh refinement using various overdecompostion factors. An overdecomposition factor of one means four partitions on four cores; two means eight partitions; four means 16, *et cetera*. Figure 4.3 shows the performance of Delaunay mesh refinement, running on 4 cores (the experimental setup was as described in Section 4.5), with varying overdecomposition factors.

We immediately see the correlation between abort rate and performance. As the abort rate monotonically decreases as the overdecomposition factor increases, there is a rough correlation between performance and overdecomposi-

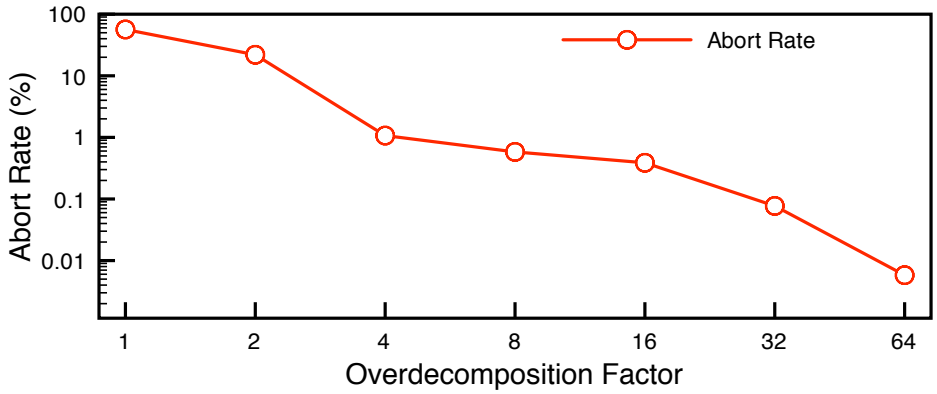


Figure 4.4: Relationship between overdecomposition factor and log(abort rate)

tion factor. However, while abort rate decreases dramatically as the overdecomposition factor increases (Figure 4.4 plots the abort rate on a log scale to make this more evident), the execution time does not change significantly beyond a certain overdecomposition factor. Essentially, once the abort rate decreases to a certain level, its effect on performance becomes negligible. Hence, further overdecomposition is not useful.

In the case of Delaunay mesh refinement, the size of the mesh is large enough that even high overdecomposition factors do not lead to increased synchronization. With an overdecomposition factor of 64 on 4 cores, there are 256 partitions. In these experiments, the input mesh contains roughly one hundred thousand triangles, so each partition has roughly four hundred triangles. As the average cavity contains only 6 triangles, the total number of triangles that are accessed by an iteration is fairly low, and hence the likelihood that an iteration touches multiple partitions is low even with such small partitions. Furthermore, as the program executes, the mesh, and hence each partition, grows in size. Thus, the chance of a iteration's touching multiple partitions decreases with time.

This is not the case for B-K maxflow (augmenting paths), however. Here,

the size of the graph is fixed throughout execution. However, the amount of the graph an individual iteration touches is fairly small (the average iteration will touch a single node and its immediate neighbors), so it requires very high overdecomposition factors for synchronization to become an issue. In the scenario where each node resides in its own partition (for a 1024 x 1024 graph, this requires an overdecomposition factor of 262144 on four cores), however, performance does suffer. In this case, the program runs roughly three times slower than it does with an overdecomposition factor of 16.

In general, the appropriate level of overdecomposition is dependent on algorithmic behavior and input characteristics. We leave a complete study of overdecomposition and its various effects to future work.

4.4 Implementation

There are a number of requirements that an implementation of the partitioning techniques described above 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 writer of user code (*i.e.*, Joe 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 user code should change as little as possible when a non-partitioned data structure is replaced with a partitioned data structure (compare this

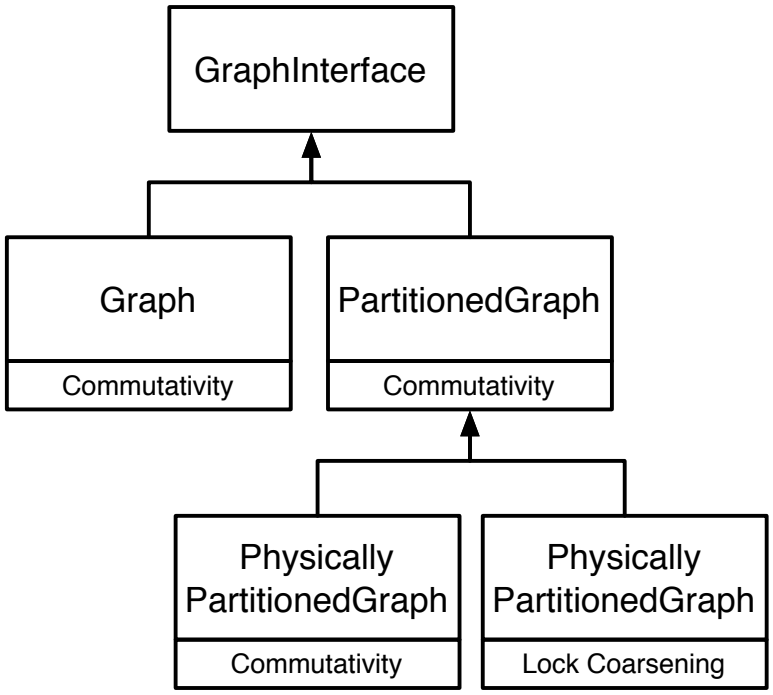


Figure 4.5: Class hierarchy for graphs

with distributed-memory programming). This allows Joe Programmer to more easily integrate partitioned data structures into an existing program.

Implementing abstract domains; logical and physical partitioning; computation partitioning; and lock coarsening while meeting these requirements is straightforward in the Galois system thanks to its object-oriented nature. We now address how each of these techniques are implemented in detail.

Abstract domains

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. This distribution can take advantage of the topology of the abstract domain, as described in Section 4.2.2.

Data partitioning

The implementation of partitioning in the Galois system is straightforward, and is the responsibility of expert programmers (*i.e.*, Steve Programmer). 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.

It's important to note that performing logical and physical partitioning can be accomplished without changing the interfaces to the data structure. Thus, any user program which is written against a common interface can use any of the partitioned data structures which provide the same interface. Figure 4.5 gives an example of this. Several classes, providing unpartitioned, logically partitioned and physically partitioned graphs implement the same `GraphInterface`. Thus, if user code is written against the `GraphInterface`, switching from an unpartitioned to a partitioned graph simply requires chang-

ing the object instantiation. No other changes are required to user code.

Computation partitioning

Computation partitioning is accomplished purely by a change to the Galois run-time system. Recall that the scheduling policy used by the system is under the control of the *Scheduler* object in the Galois run-time. Thus, different scheduling policies can be implemented simply by changing the scheduling object. When iterating over a partitioned worklist, the Galois run-time automatically chooses the scheduler object implementing data-centric scheduling. Thus, the run-time automatically performs computation partitioning with no user intervention.

Lock coarsening

Recall that commutativity checks in the Galois system are implemented by wrapping shared objects in *Galois wrappers* which contain `ConflictDetection` objects (see Section 3.3.4). The `ConflictDetection` object 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 `ConflictDetection` object for a data structure. Rather than performing commutativity checks, the new wrapper uses the lock coarsening approach for conflict detection. Because both commutativity checks and lock coarsening are implemented using `ConflictDetection` objects, and calls to these objects are hidden within the Galois wrapper, the user code remains agnostic to which form of conflict detection is used. As in the case of data partitioning, lock coarsening places an additional burden on Steve Programmer (to provide new `ConflictDetection`

objects) but does not increase the difficulty of writing user code.

Figure 4.5 shows an example of this. There are two different partitioned graphs in the hierarchy, one using standard commutativity checks, and the other using partition locks. As before, the interface to the graph remains the same regardless of which `ConflictDetection` object is used, so Joe Programmer merely needs to change object instantiation to allow the run-time to use different means of conflict detection.

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

4.5 Case studies

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 chapter, in Java 1.6[1]. 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

[1]This is a port of the original Galois system, written in C++, used in Chapter 3.

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.

4.5.1 Delaunay mesh refinement

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 4.2.2.

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 4.2.3.

- $meshgen_{lco}$ — 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 [119]. It had 100,364 triangles and boundary segments, of which 47,768 were bad.

Table 4.2 shows the wallclock time (in seconds) for the 5 benchmarks. Figure 4.6 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. These results are different than those presented in Section 3.5.1, as the worklist is implemented as a stack, rather than the randomized set presented previously. $meshgen_{ovd}$, 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 4.2). Table 4.3 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 ex-

pensive, as the overheads show.

Table 4.2: Execution time (in seconds) for Delaunay mesh refinement.

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

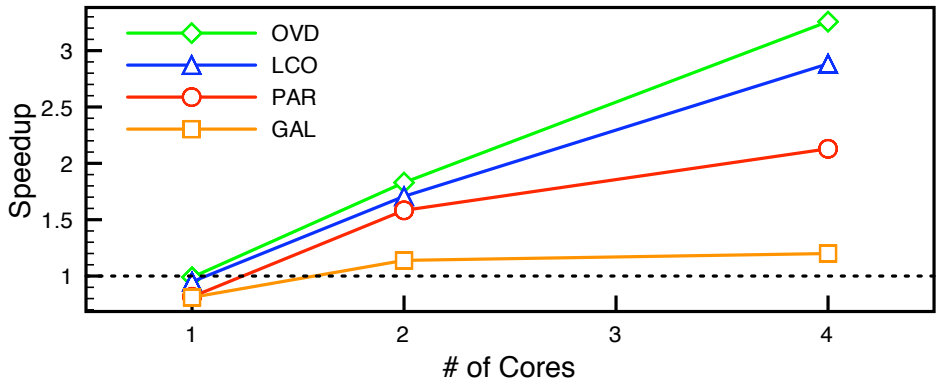


Figure 4.6: Speedup vs. # of cores for Delaunay mesh refinement

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 4.3 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 work-

Table 4.3: Uniprocessor overheads and abort ratios

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%

list is implemented as a stack, which leads to high abort ratios for this application, as mentioned in Section 4.1.2. 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 itera-

tion 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.

4.5.2 Boykov-Kolmogorov maxflow

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 ver-

sions, the worklist is implemented as a queue, matching the C implementation. The input data is a 1024x1024 grid representing a checkerboard pattern. Table 4.4 shows the wallclock time of the 5 benchmarks. Figure 4.7 shows speed-ups relative to the sequential version. Table 4.5 shows the uniprocessor overheads and abort ratios of the four parallel versions on 4 cores.

Table 4.4: Execution time (in milliseconds) for B-K maxflow.

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

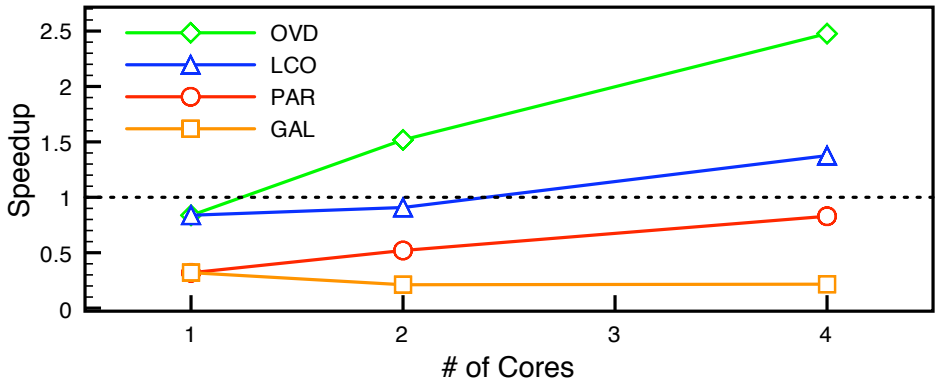


Figure 4.7: Speedup vs. # of cores for B-K maxflow

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

Table 4.5: Uniprocessor overheads and abort ratios

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%

though the 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 4.5, keep this implementation from scaling (as in Section 4.5.1, 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.

4.5.3 Preflow-push maxflow

Partitioning strategy For image processing applications, input graphs typically have a grid-like structure. Therefore, as in the B-K maxflow 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 4.6 gives wallclock execution times for the five benchmark versions (in seconds), while Figure 4.8 shows speedups over the sequential code. Table 4.7 gives the overheads for the four parallel versions running on a single core, and the abort ratios on four cores.

Table 4.6: Execution time (in seconds) for preflow-push.

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

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.

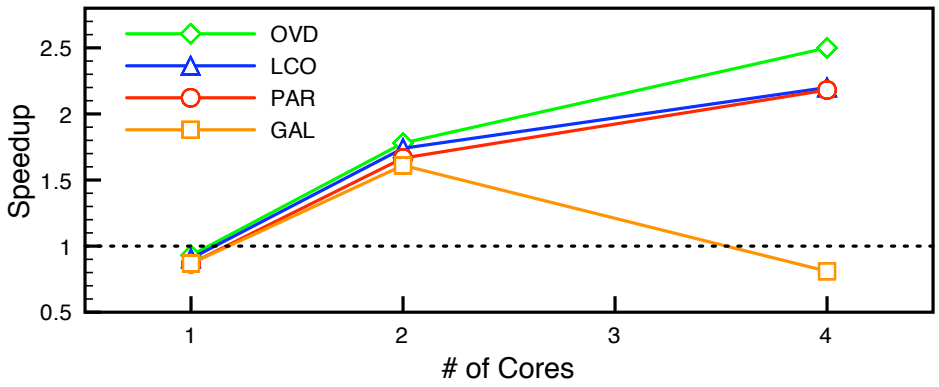


Figure 4.8: Speedup vs. # of cores for preflow-push

Table 4.7: Uniprocessor overheads and abort ratios

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%

Unlike the other benchmarks we evaluated, preflow-push does not require optimistic *parallelization*, but only optimistic *synchronization*. Work performed early in an iteration remains valid even if work done later in the iteration conflicts with another, concurrent iteration. Thus, we use Galois to detect these conflicts and maintain the consistency of the solution, but we do not have to roll back iterations that abort. The measured abort ratio therefore has a different effect on performance than in other benchmarks. However, it is still broadly indicative of concurrency.

Table 4.7 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 4.8. 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.

4.5.4 Unordered agglomerative clustering

Partitioning strategy 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 [133] to use Galois iterators and commutativity checks[2]. 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 4.8 gives wallclock execution times for the three

[2]Unlike in Chapter 3, here we use the unordered version of agglomerative clustering, described in Section 2.3.2.

benchmark versions (in seconds), while Figure 4.9 shows speedups over the sequential code. Table 4.9 gives the overheads for the two parallel versions running on a single core and the abort ratios on four cores.

Table 4.8: Execution time (in seconds) for agglomerative clustering.

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

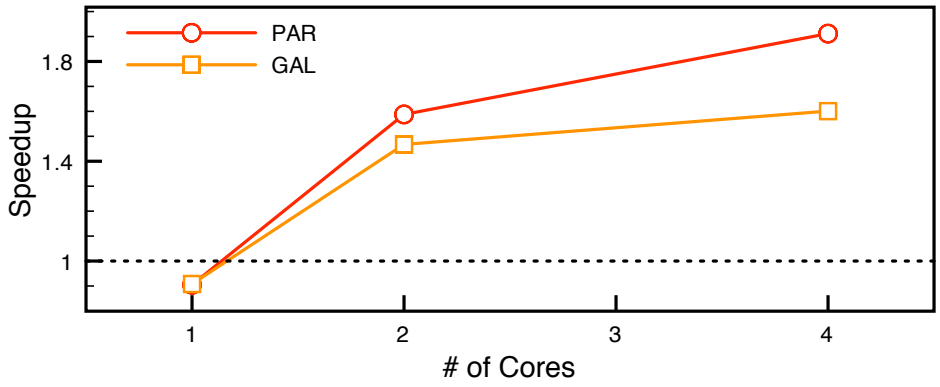


Figure 4.9: Speedup vs. # of cores for agglomerative clustering

Table 4.9: Uniprocessor overheads and abort ratios

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

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

4.6 Summary

In this chapter, we described several key optimizations to the Galois system which improve scalability with low programmer overhead:

First, we showed 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. This allows irregular programs to exploit many of the same techniques exploited by regular programs to achieve parallelism while maintaining 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.

Partitioning has always been a common approach to parallelizing irregular applications. Antonopoulos *et al.* showed how to parallelize Delaunay mesh refinement through partitioning [8], and Scott *et al.* presented a parallel version of Delaunay triangulation using transactional memory for synchronization [116]. There are several common threads running through applications parallelized through partitioning: first, they require application-specific partitioning schemes; second, they require careful synchronization and application specific code at the boundaries between partitions. These restrictions have limited the writing of partitioned parallel programs to domain experts who understand the appropriate partitioning to perform and the necessary actions to take at boundaries. Furthermore, it is difficult to share knowledge and techniques between two applications parallelized in this way. The upshot of this parallelization process is that it allows for significant parallelism; no synchronization is necessary except at partition boundaries.

The partitioning techniques in the Galois system allow irregular applications to be parallelized in much the same way, without significant programmer overhead. By using data-structure specific partitioning, we allow partitioners to be written once, and then used across multiple programs. Because the scheduling of computation automatically leverages the partitioning of data structures, a programmer does not need to intervene to appropriately assign work to processors. And, by using lock coarsening, synchronization can largely be eliminated (as long as iterations remain within a single partition), and programs correctly and naturally handle partition boundaries, without any application specific code.

The Galois system, extended with partitioning, is thus the first approach we know of to allow programmers to exploit data partitioning to promote locality, reduce mis-speculation and lower overheads and provide scalable, parallel implementations of irregular programs in a general way.

CHAPTER 5

FLEXIBLE SCHEDULING

5.1 Overview

In this chapter, we address the problem of scheduling the iterations of Galois set iterators for parallel execution. In principle, these iterations can be assigned arbitrarily to different cores and each core has the freedom to execute the iterations mapped to it in any order. In practice, we have found that even for sequential execution, the performance of the program is affected dramatically by the scheduling policy. Consider a sequential execution of Delaunay mesh refinement. If newly created bad triangles (line 10 in Figure 2.3) are processed immediately, we get the benefits of exploiting temporal and spatial locality. For this reason, hand-written implementations of Delaunay mesh refinement use a stack to implement the worklist. A scheduling policy that picks a bad triangle at random from the current worklist will not exploit locality and may therefore perform poorly. Just how much performance is lost depends on the size and shape of the mesh, cache parameters, etc. but the experiments reported in Section 5.6.1 show that the slow-down over the LIFO schedule can be more than 33% for even moderately sized meshes. Paradoxically, other applications such as Delaunay *triangulation* [47] suffer enormous slow-downs if the schedule tries to exploit locality. In Section 5.6.2, we show that using a locality-aware schedule for this problem can triple the execution time compared to using a random schedule!

Even for the same application, a good sequential scheduling strategy may be bad for parallel execution. For *parallel* Delaunay mesh refinement, using a stack (with atomic push and pop operations) to implement the worklist can

double execution time compared to using the randomized scheduling policy, as we show in Section 5.6.1. This has nothing to do with the overhead of accessing the global worklist since both scheduling strategies involve the same number of accesses; instead, it turns out that there is significant mis-speculation if the worklist is implemented as a stack.

This discussion shows that the problem of scheduling iterations of Galois set iterators is considerably more complex than the more familiar problem of scheduling iterations of DO-ALL loops in regular programs. In particular, we have found that the relatively simple scheduling policies in OpenMP for supporting scheduling of DO-ALL loops [96] are not adequate for scheduling iterations of Galois set iterators. To ease the implementation of application-specific schedules, we have designed a general scheduling framework and have used it to implement a number of specific scheduling policies in the Galois system.

The rest of this chapter is organized as follows. In Section 5.2, we describe our scheduling framework and discuss how it is integrated with the Galois system. In Section 5.3, we present a number of instantiations of this scheduling framework, and discuss their properties and when they may be useful. We then discuss how programmers can implement these scheduling policies within the Galois system in Section 5.4. We also discuss a scheduling-related optimization we perform called *iteration coalescing* in Section 5.5. In Section 5.6, we describe experimental results for several real-world irregular applications: Delaunay mesh refinement [25], Delaunay triangulation [47], the Boykov-Kolmogorov maxflow algorithm (used in image segmentation) [17], the preflow-push algorithm [42] for maxflow, and agglomerative clustering [133]. We use our scheduling framework to evaluate a number of different schedules for each application, illustrating the effects of scheduling on performance and the efficacy of our

framework. We summarize in Section 5.7 with a discussion of lessons learnt.

5.2 Scheduling framework

In principle, the iterations of an unordered set iterator can be executed in any order, and the run-time system has complete freedom in how it assigns iterations to processors for execution. This is true even for an ordered set iterator, although in that case the run-time system must ensure that iterations commit in the right order. However, the performance of the program may depend critically on the scheduling policy used to execute the loop for the following reasons.

1. *Algorithmic effects*: In some irregular applications, the scheduling policy can affect the efficiency of an algorithm or data structure used by the application. For example, a commonly used algorithm for Delaunay triangulation, described in Section 2.2, uses a data structure called the history DAG whose operations have good expected-case complexity but bad worst-case complexity[1]. Rewriting the application to use a different algorithm or data structure is one option, but this may not always be possible.

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

[1] *cf.* the behavior of a binary search tree.
[2] *cf.* inserting sorted elements into a binary search tree

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

4. *Load-balancing*: The assignment of work to cores should attempt to balance the computation load across cores. This can be difficult in irregular programs because work is often dynamically created, and because load-balancing may conflict with locality exploitation. For example, in Delaunay mesh refinement, load-balancing can be accomplished by assigning each core a randomly chosen bad triangle whenever the core needs work [80]. However, this policy limits locality.

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

5.2.1 Comparison with scheduling of DO-ALL loops

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

For example, OpenMP supports three scheduling policies for DO-ALL loops:

static, *dynamic*, and *guided*. Static schedules assign iterations to cores in a cyclic (round-robin) fashion before loop execution begins; to exploit locality, the programmer can specify that the assignment be done in a block-cyclic fashion in *chunks* of c contiguous iterations at a time. Static schedules can lead to load imbalance if the execution times of iterations vary widely. In dynamic scheduling [101], the system assigns iterations to cores whenever the core needs work; this is good for load-balancing, but if each iteration does only a small amount of work, the overhead of assigning iterations dynamically can be substantial. To ameliorate this problem and to permit locality exploitation, the programmer can ask the system to hand out chunks of c contiguous iterations at a time. Guided self-scheduling [100] is a more sophisticated form of dynamic scheduling in which the chunk size is decreased gradually towards the end of loop execution.

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

5.2.2 Our approach

A fully defined schedule for a set iterator requires the specification of three policies (see Figure 5.1).

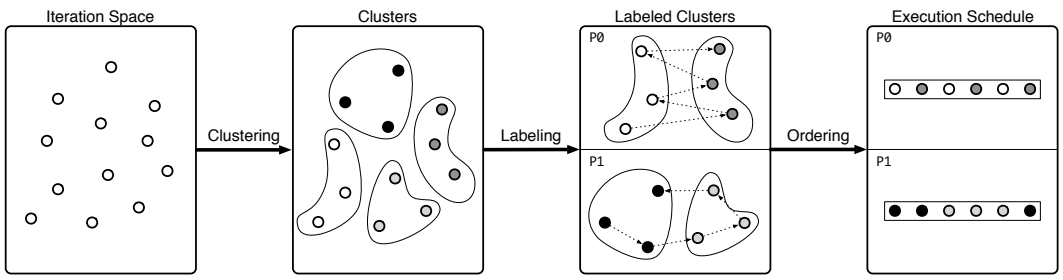


Figure 5.1: Scheduling framework

1. *Clustering:* A cluster is a group of iterations all of which are executed by a single core. The clustering policy maps each iteration to a cluster.
2. *Labeling:* The labeling policy assigns each cluster of iterations to a core. A single core may execute iterations from several clusters, as shown in Figure 5.1.
3. *Ordering:* The ordering policy maps the iterations in the clusters assigned to a given core to a linear order that defines the execution order of these iterations.

To understand these policies, it is useful to consider how the static and dynamic scheduling schemes supported by OpenMP map to this framework. For a static schedule with chunk size c, the clustering policy partitions the iterations of the DO-ALL loop into clusters of c contiguous iterations. The labeling policy assigns these clusters to cores in a round-robin fashion, so each core may end up with several clusters. The ordering policy can be described as *cluster-major* order since a core executes clusters in lexicographic order, and it executes all iterations in a cluster before it executes iterations from the next cluster. Notice that for static schedules of DO-ALL loops, the iteration space, clusters and the three scheduling policies are known before the loop begins execution. For dynamic schedules on the other hand, some of these policies are defined incrementally

as the loop executes. Consider a dynamic schedule with chunk size c. As in the case of static schedules, the clustering policy partitions iterations into clusters of c contiguous iterations, and this policy is defined completely before the loop begins execution. However, the labeling policy is defined incrementally during loop execution since the assignment of clusters to cores is done on demand. The ordering policy is cluster-major order, as in the static case. In general therefore, Figure 5.1 should be viewed as a post-execution report of scheduling decisions, some of which may be made before loop execution, while the rest are made during loop execution.

Scheduling in irregular programs can be viewed as the most general case of Figure 5.1 in which even the iteration space and clusters are defined dynamically. In applications like Delaunay mesh refinement, elements can be added to the worklist as the loop executes, and this corresponds abstractly to the addition of new points to the iteration space of the loop during execution. It is convenient to distinguish between the *initial iterations* of the iteration space, which exist before loop execution begins, and *dynamically created iterations*, which are added to the iteration space as the loop executes. The initial iterations may be clustered before loop execution begins, but the run-time system may decide to create new clusters for dynamically created iterations, so both the iteration space and clusters may be defined dynamically.

5.3 Sample policies

We now describe a number of policies for clustering, labeling and ordering that we have found to be useful in our application studies.

5.3.1 Clustering

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

- *Chunking:* This policy is defined only for *ordered-set* iterators, and it is a generalization of OpenMP-style chunking of DO-ALL loops. The programmer specifies a chunk size c, and the policy clusters c contiguous iterations at a time.
- *Data-centric:* In some applications, there is an underlying global data structure that is accessed by all iterations. Partitioning this data structure between the cores often leads to a natural clustering of iterations; for example, if the mesh in Delaunay mesh refinement is partitioned between the cores, the responsibility for refining a bad triangle can be given to whichever core owns the partition that contains that bad triangle[3] [79]. The data-centric policy is similar in spirit to what is done in High Performance FORTRAN (HPF) [75, 110]. The number of data partitions is specified by the programmer or is determined heuristically by the system.
- *Random:* In some applications, it may be desirable to assign initial iterations to clusters randomly. The number of initial clusters is specified by the programmer or is chosen heuristically. Alternately, a programmer can choose to leave unspecified the number of clusters, but instead specify a desired cluster size.
- *Unit:* Each iteration is in a cluster by itself. This can be considered to be a degenerate case of random clustering in which each cluster contains exactly one iteration. This is the default policy.

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

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

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

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

Comparison with owner-computes

It is interesting at this point to compare the presented clustering policies to the well-known *owner-computes* rule for assigning work to processors [110]. In owner computes, work is assigned based on which processor owns the data the work accesses. Intuitively, the *partitioned* clustering policy captures this rule. When a data structure is partitioned among many abstract processors, which are then mapped to physical cores, each core effectively owns several partitions of the data structure. By assigning work to cores based on the partition that work belongs to, the *partitioned* clustering policy is equivalent to owner computes.

The *inherited* clustering policy, on the other hand, differs from the *partitioned* policy (and hence from owner-computes) in a key way: newly generated work is handled by the core which generates the work. In many cases, this is equivalent to the *partitioned* policy. For example, in Delaunay mesh refinement, newly generated work is, because of the partition assignment policy discussed in Section 4.2.2, always part of a partition mapped to the current core.

However, it is possible for newly generated work to belong to a partition mapped to a different core. For example, BK maxflow (Section 2.4) generates new work as part of the breadth-first-search phase of execution. New nodes added to the frontier may be on a different partition than the node currently being processed. According to the *partitioned* policy, that new node should be processed by the core it is mapped to. The *inherited* policy will instead cause the current core to process the node, even though it is "owned" by a different core. This breaks the owner compute rule, but can provide better locality, as the newly discovered node will be in the current core's cache. We present a comparison of the *inherited* and *partitioned* schedulers in Section 5.6.3.

5.3.2 Labeling

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

We have implemented the following static labeling policies.

- *Round-robin:* For ordered-set iterators, clusters can be assigned to cores in a round-robin fashion. This is similar to what is done in OpenMP.
- *Data-centric:* If clustering is performed using a data-centric policy, the cluster can be assigned to the same core that own the corresponding data par-

tition. This promotes locality and also reduces the likelihood of conflicts because cores work on disjoint data for the most part.

- *Random:* Clusters are assigned randomly to cores.

We have also implemented the following dynamic policies.

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

5.3.3 Ordering

The ordering policy specifies a sequential execution order for all the iterations assigned to a processor. We have found that it is intuitive to split the ordering policy into two sub-policies, *inter-cluster ordering* and *intra-cluster ordering*.

Inter-cluster ordering specifies the order in which the clusters are executed, and when execution may switch from one cluster to another. We have implemented the following policies:

- *Random:* Clusters are randomly chosen, and execution switches between clusters randomly.

- *Lexicographic:* This is applicable to ordered-set iterations. Clusters are executed according to their ordering in the iteration space.
- *Cluster-major:* A single cluster is fully executed before switching to a second cluster.
- *Switch-on-abort:* Iterations are executed from one cluster till some iteratino aborts. At that point, execution switches to a different cluster.

Once a cluster is picked, intra-cluster ordering specifies in what order the iterations within that cluster should be executed. Sample policies include:

- *Random:* Iterations are executed at random.
- *Lexicographic:* Iterations are executed in the order specified by the ordered-set.
- *LIFO/FIFO:* If iterations are dynamically added to a cluster, they are executed in a LIFO/FIFO manner.

5.4 Applying the framework

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

Recall that the object which controls scheduling in the Galois run-time is the *Scheduler*. It provides methods which specify how a thread obtains a new piece

of work (essentially performing the actions of the labeling policy and ordering policy), and how a core adds new work to the iterator (performing the actions of the clustering policy). By subclassing the scheduler and selectively overriding these methods, it is straightforward for Steve Programmer to implement any schedule he or she desires. Our implementations of the policies described above use this technique.

Given a set of scheduler objects which implement different scheduling policies, the question then becomes how a programmer can specify which policy to use for his or her program. One way of setting this policy is through compiler directives similar to OpenMP pragmas: the data-parallel loop is annotated to indicate the policy the programmer desires. Unfortunately, this is too restrictive for our purposes. By limiting the policies to those provided by compiler directives, programmers lose the ability to specify custom scheduling policies (which are often necessary, as we see in Section 4.5.4).

Instead, we take a programmatic approach. When instantiating the Galois run-time, Joe Programmer can pass in a particular scheduler object to set a scheduling policy. This requires minimal changes to the user code, comparable to compiler directives, but retains the full flexibility of the scheduling framework. The Galois system provides several common policies which provide acceptable performance for a number of applications (see Section 5.6.6).

5.5 Iteration granularity

One significant source of overhead in the Galois system is that of *iteration granularity* (*i.e.*, the amount of work done by a single iteration). If a single iteration is short, the overheads of obtaining work, as well as the overheads of tracking that iteration's execution, can often far outweigh the benefits of executing

work in parallel (this is especially evident in the results for Boykov-Kolmogorov maxflow, in Section 5.6.3, and for Preflow-push maxflow, in Section 5.6.4).

Several of the scheduling policies presented in the previous section attempt to address the overhead of obtaining work for fine-grained iterations. A clustering policy that groups multiple iterations together allows them to be assigned to a processor as a group, which can result in significantly lower overhead (*cf.* chunked scheduling in OpenMP). However, clustering policies cannot address the other source of overhead that overwhelms fine-grained iterations: synchronization overhead. We address this with an optimization called *iteration coalescing*

5.5.1 Iteration coalescing

Recall that in the Galois programming model, iterations of the optimistic iterators represent the basic unit of transactional execution during parallel execution. In other words, each iteration appeared to execute atomically, and in isolation, but there could be arbitrary interleavings of iterations (these interleavings can be restricted by the scheduling policy, but there is no issue of correctness involved). Iteration coalescing breaks this connection between iterations and transactional execution. Instead, we can treat multiple iterations as a single *super-iteration*, a transactional piece of work which as a whole appears to be atomic and isolated. This means that rather than allowing individual iterations to appear to interleave arbitrarily in the serialized schedule, iterations forming a super-iteration will appear as a chunk—they must appear to execute one after another.

This model of execution can be implemented in the run-time by separating the tasks of creating a new iteration record (which maintains the information

required to ensure isolation) from getting a piece of work from the scheduler (which represents a single iteration of the optimistic iterator).[4] Execution then proceeds as follows.[5]

When a core needs work, the run-time creates a new iteration record, and then the scheduler provides a new iteration to work on. The core executes the iteration, acquiring locks and recording undo methods as necessary. When the iteration completes, rather than committing the iteration by releasing all the locks, the core simply returns to the scheduler to obtain a new iteration, while still holding all its locks; this new iteration is associated *with the same iteration record as the first*. Thus, the two iterations are combined into a single super-iteration. The next iteration then executes, potentially acquiring new locks. Eventually, according to some policy, the core commits all of the speculative iterations in the super-iteration by releasing its locks and clearing its undo logs. If at any point a conflict is detected, *all* of the speculative iterations currently being executed by the core are rolled back.

Tradeoffs

Decoupling transactional execution from iterations has several advantages. First, there is lower overhead as fewer iteration records need to be created; this can be especially advantageous for fine-grained iterations. Second, and more interestingly, because the second iteration is associated with the same iteration record as the first, all the locks acquired by the first iteration are still "owned" by the second iteration, which can then "reuse" them without reacquiring the

[4]Because there is no longer a one-to-one correspondence between iteration records and iterations from the worklist, "iteration record" is a bit of a misnomer. However, we retain the nomenclature for the sake of consistency.

[5]For the sake of simplicity, the following discussion assumes a two-phase locking approach to maintaining isolation, but the run-time behaves similarly when using commutativity conditions.

locks. Thus, *the average number of locks acquired per iteration decreases*, lowering overhead. Consider Delaunay mesh refinement, using partition locking for conflict detection. When a core begins speculative execution, it will acquire a lock on the partition the first iteration accesses. If a second iteration is associated with the same iteration record, and it accesses the same partition, the core will not need to acquire any additional locks; thus the total amount of synchronization decreases. This is again especially useful for short iterations where synchronization overheads are relatively larger.

The primary downside to this approach is that it lengthens the amount of time iterations remain speculative. As a result, locks are held for longer (as they cannot be released until the iterations are no longer speculative), which increases the likelihood of aborts. Unfortunately, aborts are more expensive with iteration coalescing, as there is more speculative work to roll back, and hence more wasted work. In short, iteration coalescing is a balancing act between lowering synchronization overhead and maintaining low mis-speculation rates.

Reducing wasted work with checkpointing

One potential technique to reduce the amount of wasted work on rollback (and hence to tolerate the higher mis-speculation rate) is to implement a form of *checkpointing*. Recall that even though iteration coalescing means that we treat groups of iterations as a single, atomic whole, the sequential semantics of the program still allow the iterations to execute individually. Consider a sequence of n iterations which are coalesced together. If iteration i detects a conflict and thus should be rolled back, the standard coalescing semantics require that all the iterations in the super-iteration be rolled back. However, iterations 1 through $i - 1$ have successfully and safely executed in isolation already. Rolling back

only iteration i leaves the overall program in a consistent, safe state. In essence, iteration boundaries serve as checkpoints for a super-iteration, demarcating safe points to roll back to. Thus, iteration i can be rolled back and iterations 1 through $i - 1$ can be committed. When iteration i is re-executed, it does so within a new super-iteration.

Note that if a different core detects a conflict with a super-iteration, we cannot be sure which iteration within the super-iteration is responsible for the conflict. Thus, we must conservatively roll back the entire super-iteration, rather than simply rolling back to a safe checkpoint.

This checkpointing rollback system is similar to *partial rollback* of nested transaction [87, 90, 113]. In that setting, when a nested transaction encounters a conflict, it is possible to only roll back the nested transaction, rather than the parent transaction as well. One can think of a super-iteration as a parent transaction, with the iterations that comprise it each being a nested transaction; rollback to a checkpoint is analogous to a partial rollback of a nested transaction. One key difference is that when performing a partial rollback of a nested transaction, the parent transaction continues to maintain isolation (essentially, it continues to hold any locks that it acquired) and the nested transaction will eventually have to re-execute. Thus, in the face of persistent conflicts, the parent transaction must roll back as well. However, in our system, when an iteration is rolled back, we can safely commit the previously executed work (releasing any held locks), avoiding persistent conflicts. We have found that the additional overhead required to implement the checkpointing system outweighs the benefits in reduced rollback costs—however, further study is required, which we leave to future work.

5.5.2 Discussion

In general, the appropriate amount of coalescing is highly application dependent. It is also affected by scheduling policy, as some scheduling policies result in a lower probability of aborts, and hence can benefit more from iteration coalescing. Iteration coalescing is also more useful with "coarse grain" conflict detection, such as partition locking, as coalesced iterations are more likely to use the same locks. In contrast, fine-grained commutativity checks may not lead to much reuse, reducing the benefits of iteration coalescing. Finally, although iteration coalescing can be beneficial for many applications, the benefits are most pronounced when the average iteration is short-running, as the synchronization and bookkeeping overheads are higher relative to the amount of work to be done.

A useful default policy when using partition locking is to coalesce all newly generated work together with the iteration which generated the work, under the assumption that these iterations are likely to lie in the same partition, and unlikely to conflict with other concurrently executing iterations. We use this coalescing policy when using the *inherited* clustering policy in all of the case studies we present in Section 5.6, with the exception of agglomerative clustering, which did not coalesce iterations. We leave a study of the effects of various coalescing policies on performance to future work.

5.6 Case studies

We have evaluated our scheduling approach on five irregular applications. The scheduling framework described in Section 5.2 can be used to implement a vast number of policies and it is both infeasible and pointless to evaluate all these

policies on all benchmarks. Instead, we studied the algorithms and data structures in these applications, and determined a number of interesting scheduling policies for each one. We then implemented these policies in the Galois system and measured the performance obtained for that application using each of these policies.

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

5.6.1 Delaunay mesh refinement

Scheduling issues. As discussed in Section 5.2, the scheduling policy can affect performance because of algorithmic effects, locality, conflicts, load balancing, and overhead. In mesh refinement, algorithmic effects are minor. The final mesh depends on the order in which bad triangles are refined, and although different orders perform different amounts of work, the variation in the amount of work is small. Furthermore, the cost of getting work from the worklist is relatively small compared to cost of an iteration, so the effect of scheduling overhead is small. Therefore, the main concerns are locality, conflicts, and load

balancing.

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

Evaluation. The baseline sequential implementation, called *seq* in this discussion, uses a LIFO scheduling policy, implemented using a stack as the worklist to exploit spatial and temporal locality. In all the parallel implementations discussed in this section, the mesh is partitioned between the cores, and conflict detection is performed using partition locking rather than commutativity checks, as described in Section 4.3. This ensures that all parallel versions use the same mechanism for conflict checks, so the only difference between them is the scheduling policy[6].

We evaluated four different parallel schedules:

- *default* — This is the default schedule used by the base Galois system: the worklist is centralized, and a core is given one bad triangle, chosen at random, on demand. In terms of the scheduling framework introduced in Section 5.2, we can describe this schedule as follows: it uses the *unit* clustering policy for both initial and dynamically generated iterations, and the labeling policy is dynamic and random. Obviously, there are no ordering concerns in this policy.

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

- *stack* — This policy is similar to *default*, except that the worklist is stack-like, so the labeling policy is dynamic and LIFO rather than random. This policy mimics the scheduling policy of *seq*.
- *part* — This schedule uses data-centric clustering for both initial iterations and dynamic iterations, with 4 times as many partitions as processors. The labeling policy is also data-centric. The cluster interleaving used is *switch-on-abort*. Within a cluster, iterations are ordered in a LIFO manner, processing newer work first. The mesh is partitioned using breadth-first search, a simplified version of the Kernighan-Lin method [74].
- *hist* — This schedule uses a *random* clustering policy for initial iterations, with each cluster containing 16 elements. Dynamically created iterations use *inherited* clustering; newly created work is assigned to the cluster that is currently being processed. The labeling policy is the same as in *default*. Because the labeling policy is dynamic, there is no cluster interleaving. Iteration ordering within a cluster is LIFO.

We compared the parallel implementation using these schedules with the sequential implementation. Figure 5.1(a) gives the wallclock time, in seconds, for *seq* as well as the four parallel versions on different numbers of cores. Figure 5.1(b) shows the speedup of the five parallel versions relative to sequential execution time. We see that *stack* has the worst performance, achieving a speedup of 1.2 on 4 cores, while *hist* performs the best, achieving a speedup of 3.3.

There are a number of interesting points to note in these results. First, we note that *seq*, which is the sequential implementation that uses LIFO scheduling, and *stack*, which is the parallel implementation of LIFO-like scheduling, perform almost identically on one core; since *stack* is a parallel code, it has a small additional overhead even when run on a single core. Both versions ex-

Table 5.1: Execution time (in seconds) and abort ratios for Delaunay mesh refinement

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

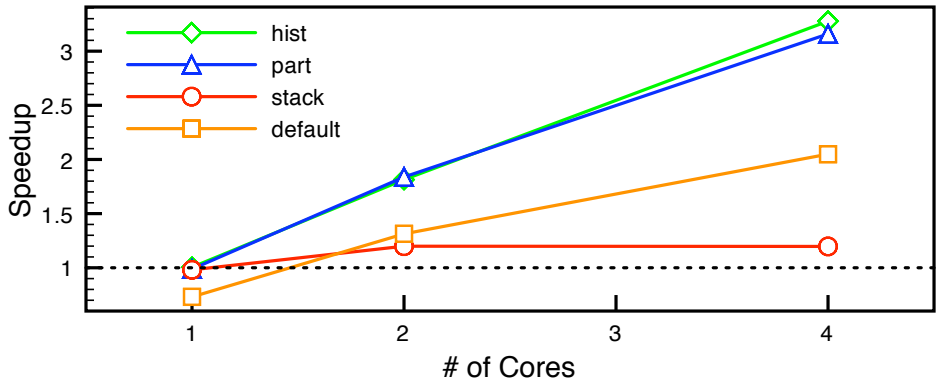


Figure 5.2: Speedup vs. # of cores for Delaunay mesh refinement

ploit locality and therefore outperform the *default* version, which uses randomized scheduling. The fact that *hist* also performs well on one core shows that most of the locality benefits can be obtained by focusing on one bad triangle from the original mesh at a time, and repeatedly eliminating all new bad triangles created in its cavity before moving on to a different bad triangle from the original mesh.

Interestingly, single-core performance does not always translate to parallel

performance. While *default* is slower than *stack* on a single core, it is faster on 4 cores. This is likely due to speculation conflicts, as discussed before. To investigate this, we measured the *abort ratio*, the percentage of executed iterations which are rolled back. A high abort ratio is indicative of significant mis-speculation in the program, which may reduce performance. There is no direct correlation between abort ratio and performance: some iterations abort soon after starting (essentially a busy-wait), while others abort towards the end, resulting in more lost work. The rightmost column of Figure 5.1(a) shows the abort ratio for the parallel schedules on 4 cores. From these numbers, we see that *stack* has a very high abort ratio, as expected. By processing triangles chosen at random, *default* avoids this problem, and the gain in concurrency outweighs the cost in lost locality.

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

5.6.2 Delaunay triangulation

Scheduling issues. For this application, algorithmic effects are the most important, since it is critical to avoid worst-cast behavior of the history DAG. The

best sequential implementation (*seq*) uses a random worklist [47]. In the parallel implementation, we can exploit temporal and spatial locality if points are inserted in sorted order. Unfortunately, this can lead to worst-case behavior of the history DAG.

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

We evaluated three different parallel schedules:

- *default* — The default Galois schedule. Note that this approximates the schedule used by the sequential implementation.
- *part* — The points are partitioned geometrically. The schedule uses data-centric clustering for the initial iterations, with 8 times as many clusters as cores. The labeling policy is also data-centric. The ordering is cluster-major; within each cluster, the ordering is random. Intuitively, this scheduling policy tries to exploit locality when clustering iterations, but it does not try to exploit locality when executing a given cluster.
- *sorted* — This schedule is similar to the *part* schedule except that the ordering within each cluster is the (geometrically) sorted order rather than random. Intuitively, this scheduling policy tries to exploit locality when creating clusters and also when executing iterations in each cluster.

Figure 5.3(a) gives the overall execution time on different numbers of cores, as well as the abort ratio on four cores. Figure 5.3(b) shows the speedup relative to *seq* for the various schedules.

The *sorted* schedule exhibits the worst performance of all the evaluated schedules. Although sorting the points achieves better locality in the mesh, these results illustrate the tradeoff described earlier: sorting the points is good

Table 5.2: Execution time (in seconds) and abort ratios for Delaunay triangulation

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

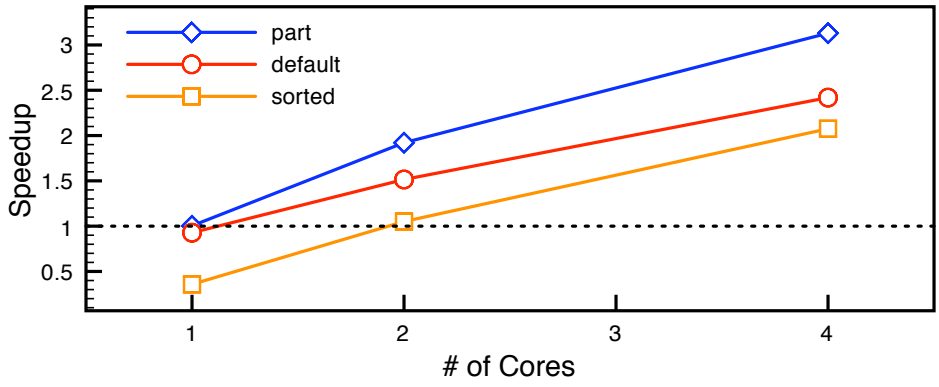


Figure 5.3: Speedup vs. # of cores for Delaunay triangulation

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

5.6.3 Boykov-Kolmogorov maxflow

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

Like mesh refinement, B-K maxflow involves a graph traversal, so the locality concerns are similar. The amount of dynamically generated work is relatively small compared to mesh refinement, so ensuring that the original distribution of work exploits locality and concurrency is important.

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

- *default* — the default schedule.
- *queue* — this is the same policy as *default*, except the labeling policy uses FIFO labeling. This schedule approximates the sequential schedule of execution since it will perform an approximate breadth-first traversal.
- *chunked* — this is the same policy as *queue*, except it uses the *random* clustering policy, and aims to create clusters of size 16. Dynamically generated iterations are assigned to new, unlabeled clusters. The ordering within a cluster is random.

- *hist* — this is the same policy as *chunked*, except the clustering function uses the *inherited* rule for dynamically generated iterations. Intra-cluster ordering is LIFO ordering. This is essentially the same schedule as *hist* in mesh refinement.
- *part* — This schedule is the same as *part* in mesh refinement, except the ordering policy uses *cluster-major* ordering and the clustering policy uses the *inherited* rule for new work. This clustering policy is key, because in this application, data-centric clustering may not assign newly created work to the current cluster. As the input graph has a grid structure, the partitioning used for data-centric clustering is block-based.

Figure 5.4(a) gives the execution time on different numbers of cores, as well as the abort ratio on four cores. Figure 5.4(b) shows the speedup relative to *seq* for the parallel schedules. We see that *default* performs the worst, actually slowing down compared to sequential execution by a factor of 10 on four cores, while *part* performs the best, achieving a speedup of 2.67x over *seq*.

Table 5.3: Execution time (in ms) and abort ratios for B-K maxflow

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

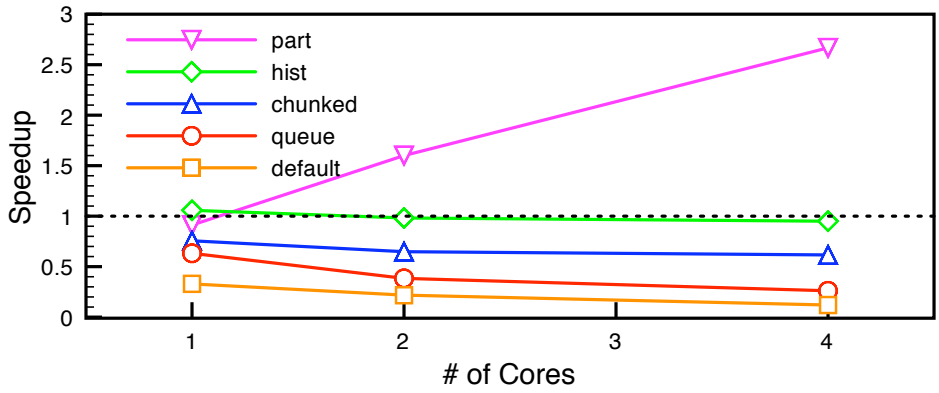


Figure 5.4: Speedup vs. # of cores for B-K maxflow

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

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

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

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

***inherited* vs. *partitioned* clustering** As discussed in Section 5.3.1, scheduling using the *inherited* clustering policy produces a different schedule than the traditional owner-comnputes heuristic. We evaluated the performance of *inherited* clustering versus the owner-computes approach of *partitioned* clustering to determine what difference, if any, existed between the two schedules. We thus compared the performance of two variants of B-K maxflow, one using the *inherited* policy, and the other using the *clustered* policy while keeping the labeling and ordering policies fixed.

In order to produce a fair comparison between *inherited* and *partitioned* clustering, we rewrote the *inherited* scheduler to use a similar implementation as the *partitioned* scheduler (rather than the fully optimized local-worklist implementation described in Section 5.3.1). This meant that the performance difference between the two systems measured just scheduling differences, rather than including implementation effects. It is important to note, in light of the results presented earlier, that this scheduler modification had the effect of slowing down the *inherited* scheduler.

Figure 5.5 shows the performance of the two clustering policies on multiple cores. On four cores, we found that *inherited* clustering ran 19% faster than

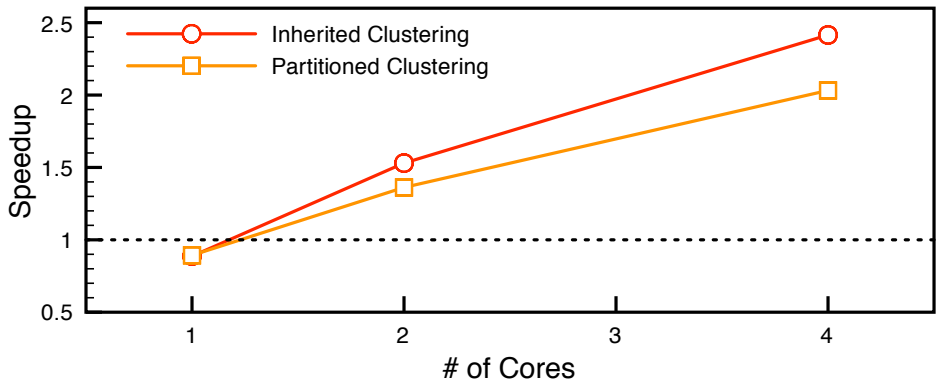


Figure 5.5: Performance of *inherited* vs. *partitioned* clustering for B-K maxflow

partitioned clustering. This indicates that the locality gains from using *inherited* clustering can be significant when compared to the owner-computes heuristic, and these benefits are independent of the performance gains obtained through a more optimized implementation.

5.6.4 Preflow-push maxflow

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

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

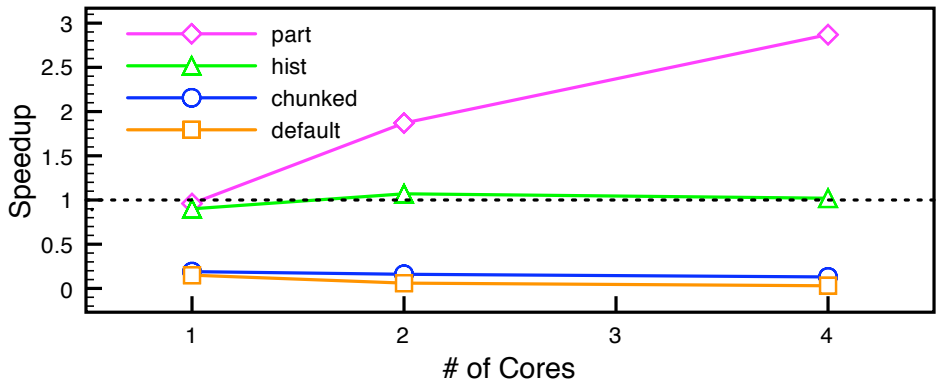


Figure 5.6: Speedup vs. # of cores for preflow-push maxflow

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

Table 5.4: Execution time (seconds) and abort ratios for preflow-push maxflow

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

Preflow-push, even more than B-K maxflow, shows tremendous sensitivity to scheduling overheads. Computing a maxflow on the input data requires

about 30 million iterations of the main loop. Unsurprisingly, *default* performs very poorly, and although *chunked* does show improvement due to larger iteration clusters, newly generated work is still handled too slowly to result in speedup.

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

5.6.5 Unordered agglomerative clustering

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

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

- *default* — The default schedule.
- *chain* — This schedule improves locality using a programmer-specified dynamic labeling policy. If an iteration does not successfully form a data

cluster between *a* and *b*, the labeling policy assigns the iteration associated with *b* to the processor next, based on a scheduler hint inserted into the iteration at line 15.

- *hist* — This schedule is the same as *chain*, except when *a* and *b* are successfully combined in a data cluster (line 13). In this case, the schedule assigns the newly created iteration to the current iteration cluster (as in the *inherited* clustering policy), using another scheduler hint. This does not affect iterations generated by line 15, otherwise the loop would never terminate.

Table 5.5: Execution time (in seconds) and abort ratios for agglomerative clustering

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

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

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

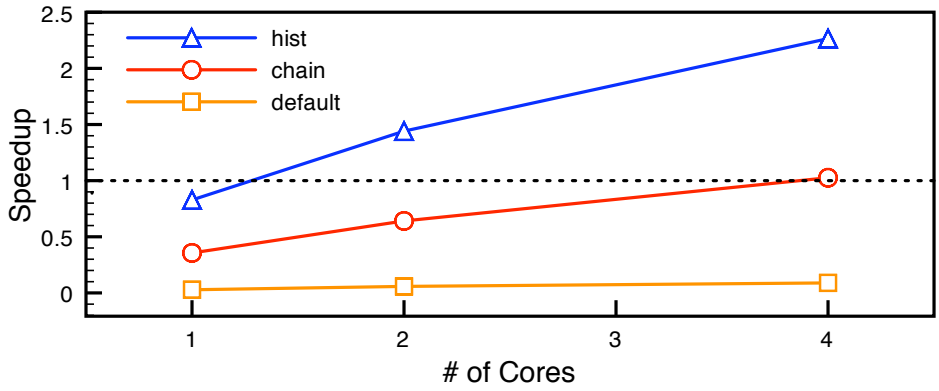


Figure 5.7: Speedup vs. # of cores for agglomerative clustering

Table 5.6: Highest-performing scheduling policies for each application

	Scheduling Policy		
Application	Clustering	Labeling	Ordering
MR	random / inherited	dynamic / random	— / LIFO
DT	data-centric / —	static / data-centric	cluster-major / random
BK	data-centric / inherited	static / data-centric	cluster-major / LIFO
PP	data-centric / inherited	static / data-centric	cluster-major / LIFO
AC	unit / custom	dynamic / custom	— / —

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

5.6.6 Summary of results

Our experimental results clearly demonstrate it is beneficial to provide scheduling flexibility across applications—the default Galois scheduling policy tends to

perform poorly. Furthermore, across different applications the optimal scheduling policy can differ. Table 5.6 summarizes the scheduling policies that we found to perform the best for each of our applications. The applications are abbreviated as follows: MR — Delaunay mesh refinement; DT — Delaunay triangulation; BK — B-K maxflow; PP — preflow-push maxflow; AC — unordered agglomerative clustering. The policies are presented as follows: clustering shows first the policy for initial work, then the policy for dynamically generated work; labeling specifies dynamic or static labeling, then the specific policy; and ordering shows first the cluster interleaving policy, then the intra-cluster ordering policy.

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

It is possible that even for a single application there is not a particular scheduling policy that performs the best. In irregular programs, behavior can be very input dependent. For lack of space, we have only evaluated each application on a single input set, and have not investigated this input-dependent

variability in scheduling. However, for the general *types* of inputs we have considered for each application (*e.g.* image segmentation problems for B-K maxflow and preflow-push maxflow), we feel that there is likely only small amounts of variability in the optimal scheduling policy across inputs. We leave a full study, which would also consider other types of inputs which may have significantly different behavior, to future work.

5.7 Summary

In this chapter we presented a general framework for scheduling data-parallel computation, suited for both regular and irregular applications. We described how a schedule can be defined through three policies: clustering, labeling and ordering. We also showed how the object-oriented nature of the Galois system can be leveraged to easily implement our framework. This framework subsumes the scheduling policies of data-parallel systems such as OpenMP [96], and it affords the Galois system significantly more flexibility than toolkits such as Intel's Thread Building Blocks [67], which does not provide control over scheduling for general iterators.

Through an evaluation of the framework on several real-world applications, we demonstrated that different schedules can exhibit widely varying performance on a given application, and that there is no single, best-performing schedule across all applications. We did, however, discover a combination of scheduling policies which gives acceptable performance across a range of applications. By extending the Galois system with our scheduling framework, we can give programmers a set of reasonable default schedules, as well as allow them to explore the space of possible schedules, arriving at the particular schedule that best suits their application.

CHAPTER 6

CONTEXT AND CONCLUSIONS

6.1 Other models of parallelism

Generalized data-parallelism is not the only model of parallelism that exists in programs; for some programs, alternate approaches to parallelization may be more appropriate. We briefly discuss other approaches to parallelism, and how they compare with amorphous data-parallelism. This is not meant to be a comprehensive review of other approaches to parallelism, but rather to place amorphous data-parallelism, and the Galois system in particular, in the broader context of parallel programming models.

6.1.1 Decoupled software pipelining

Ottoni *et al.* proposed a method for parallelizing loops called Decoupled Software Pipelining (DSWP) [97]. Data-parallel loops are parallelized by executing loop iterations concurrently. DSWP takes a fundamentally different approach to parallelizing loops, drawing an analogy with software pipelining [2]. In software pipelining, different instructions from multiple iterations are executed in parallel on multiple functional units. Similarly, in DSWP, different tasks from multiple iterations are executed in parallel on multiple processors.

The basic DSWP algorithm for parallelizing loops is fairly straightforward. A compiler examines the loop and produces a dependence graph, capturing both the data and control dependences in the loop. Strongly connected components (SCCs) of the dependence graph are then condensed, and the resulting condensed graph is sorted topologically. SCCs are then assigned to different

processors, with communication instructions inserted to communicate the appropriate dependence information between them. Thus, when executing a loop in parallel, each processor will be responsible for a portion of instructions in a loop, and it will execute those instructions for each iteration, forwarding the results to the next processor in the pipeline.

If this approach were to be applied to amorphous data-parallel programs such as those we have studied, the dependence graph of a loop would consist of a single SCC containing all the instructions of the loop. This is because of the dependences between iterations induced by updates to shared structures such as the graph. In order to avoid such situations, Vachharajani *et al.* introduced speculation into DSWP [129]. This allows some edges to be speculatively ignored. This may allow DSWP to find parallelism in applications such as Delaunay mesh refinement.

A more significant problem with DSWP is that, like thread-level speculation, it is fundamentally tied to the sequential loop ordering to guide its parallelization. Thus, as with TLS, we cannot apply the improved scheduling techniques proposed in Chapters 4 and 5. This may unnecessarily restrict the performance of parallel execution by limiting locality and increasing the likelihood of misspeculation.

6.1.2 Task parallelism

Unlike in data-parallelism, where the same operations are applied to different pieces of work, *task parallelism* envisions *different* parts of a program ("tasks") executing simultaneously. In some sense, task parallelism is a generalization of data-parallelism (as each unit of work in a data-parallel program can be viewed as a separate task), but the programming models for each tend to be significantly

different.

The main programming question in task parallelism is how to identify tasks that can be executed in parallel. One approach is to use *fork-join* parallelism to spawn additional threads of execution which run in parallel with the spawning thread. A well-known implementation of this style of parallelism is Cilk [14, 38], which provides non-blocking function calls, called *spawns*, that allow a thread to invoke a method which will be run concurrently while the thread continues to execute. These methods can then make additional non-blocking calls, producing yet more parallel computation. Each of these asynchronous methods can be viewed as a task, thus the execution can be viewed as a tree, with spawn edges linking tasks. To account for data dependences (for example, return values from an asynchronously executed method that the calling thread needs to consume), additional data dependence edges are inserted, creating a directed acyclic graph (DAG) representing the program.

An advantage of this style of parallelism is that the DAG completely captures all the dependences that a program must respect, and available parallelism is immediately evident; nodes in the DAG which are not connected can be executed in parallel. This requires that parallel tasks execute in isolation from one another. The Cilk run-time system exploits this to dynamically schedule tasks in parallel without regard to synchronization between them. This approach breaks down if data dependences cannot be determined ahead of time; it is no longer apparent which tasks can be executed in parallel. Agrawal *et al.* extended Cilk to allow tasks to be executed in parallel as transactions and rolled back if they are not independent, thus dynamically enforcing isolation between concurrent tasks [1]. This extension can be analogized with our progression from data-parallelism (with independent parallel tasks) to amorphous data-parallelism

(with *speculatively* independent parallel tasks).

Task parallelism is most useful when a program can be readily broken down into a DAG of interrelated tasks. This approach is not as appropriate for amorphous data-parallelism: if a task DAG were to ignore data dependences it would appear as a single-level tree, providing very little useful information; if the DAG were to include dependences, then all tasks would conservatively depend on one another, leading to no apparent parallelism (except through speculative techniques).

From a philosophical perspective, the greatest downside to task parallelism is that it requires an explicit notion of parallel tasks. Programmers must consider parallelism at all times when writing their applications. Cilk requires programmers to specify which methods can be executed asynchronously, and explicitly specify synchronization points in the program. This makes writing Cilk programs (and task parallel programs in general) more difficult than writing sequential programs. We feel that to make parallel programming accessible to more programmers, it must be possible to easily transition from sequential programs to parallel programs with little programmer effort, as in Galois user code, which maintains sequential semantics while delegating parallelism to the Galois class libraries.

6.1.3 Stream parallelism

Another approach to parallelism is *stream parallelism* [48, 127], where a program is broken up into several units, or "actors", which operate on "streams" of data. These actors can be connected together to form a dependence graph, with some actors consuming the output of other actors. These actors can then be distributed among several processors. Stream programs are relatively straight-

forward to parallelize. The only communication between actors is through the data streams, as they share no other state. Thus, synchronization is straightforward. Parallelism can be obtained in many ways [43, 44]:

- **Task parallelism:** Independent actors can be executed simultaneously on different processors.
- **Pipelined execution:** Dependent actors can operate on a stream in a pipelined manner. Consider two actors, `A` and `B`, with `B` consuming the stream output of `A`. If `A` and `B` are mapped to different processors, as `A` produces output, it can immediately be sent to `B` for processing, even as `A` continues to process the input stream.
- **Data-parallelism:** If an actor performs "stateless" data-parallelism (*i.e.*, the data-parallel operations require no updates to the actor's state), the actor itself can be distributed among several processors to exploit the data-parallelism.

While stream programming is an attractive approach to parallelization as it requires very little effort to synchronize, and compilers and run-time systems can automatically produce parallel code from a sequential program, it is a restrictive programming model. It only applies to programs such as image and video processing which operate on data streams. The model is not general enough to handle the highly "state-ful" loops that we see in amorphous data-parallelism.

6.1.4 Functional and data-flow languages

All the models of parallel programming to this point are based on imperative languages. However, non-imperative languages can often support parallelism

in a much more natural way. For example, functional programs lend themselves to parallelization [50]. State is immutable, so synchronization is not a concern. Furthermore, the purely functional style often makes parallelism naturally explicit: operations such as map and reduce are naturally data-parallel. This has led to the development of explicitly data-parallel languages such as NESL [13], as well as large-scale parallel architectures such as Google's MapReduce [31].

An closely related approach is that of *dataflow languages* [69], which express programs in terms of operations which are linked together based on dataflow. Conceptually, each operation is a box with one or more inputs and one or more outputs. Programs are created by linking together the inputs and outputs of these boxes. When the inputs to the operation are available, the box can "fire," producing its output, which in turn makes inputs available to other operations. Much like functional programs, dataflow programs make explicit the available parallelism (as any operations with available input can be fired, regardless of where in the program they appear). However, both dataflow languages as well as functional languages suffer because of the lack of mutable shared state; this makes them unsuitable for programs exhibiting amorphous data-parallelism, which focus on updates to shared data structures.

6.2 Summary of contributions

Through a series of application studies, we identified a common paradigm of parallelism that appears in irregular programs, *amorphous data-parallelism*. This type of parallelism manifests itself as algorithms over worklist of various kinds. Unlike in traditional data-parallelism, we allow iterations of the parallel loop to have dependences between one another. This broader definition of data-parallelism appears in numerous applications, and exposes a large class of al-

gorithms to potential parallelization.

As a result of these studies, we developed the Galois approach for parallelizing algorithms that exhibit amorphous data-parallelism. This approach imagines parallel programs as being composed of three parts: (i) user code with largely sequential semantics, with set iterators to express amorphous data-parallelism; (ii) class libraries annotated with semantic information such as commutativity and locality properties; and (iii) a run-time system which speculatively executes the user code in parallel while leveraging the semantics of the class libraries to perform accurate dependence detection and intelligent scheduling of parallel work.

Recall that in Chapter 1, we laid out three features that a system for parallelizing irregular applications should possess: a reasonable sequential programming model, a dynamic approach to parallelization, and a higher level of abstraction. The design of the Galois system exhibits each of these qualities:

Intuitive programming model: The user code of the Galois system is purely sequential in nature, as discussed in Section 3.2.1. Programmers do not have to deal with locks, threads or any notion of parallelism when writing Galois programs. The only additional features of the Galois programming model are the ordered and unordered set iterators. Each of these iterators have well understood sequential semantics and as such programmers do not need to understand how their code may execute in parallel. The Galois class libraries and run-time ensure that the user code precisely matches its sequential semantics when run in parallel. This feature makes it straightforward to develop code for the Galois system: once it works sequentially, it will work in parallel.

Each of the extensions that we presented to the baseline Galois system (partitioning in Chapter 4 and scheduling in Chapter 5) do little to reduce the ease

of writing Galois programs. Partitioning requires only changes to the Galois class libraries; the object oriented nature of the Galois approach means that user code needs to change very little to leverage any locality semantics that objects might possess. Scheduling changes can be made purely in the run-time; complex scheduling decisions can be hidden from the programmer can can be specified by a single line in the user code.

This work presented the first easy-to-use, flexible programming model for writing programs with amorphous data-parallelism.

Optimistic parallelization: The Galois run-time performs optimistic parallelization of programs containing Galois iterators. By speculatively executing code in parallel, the Galois approach can successfully parallelize programs exhibiting amorphous data-parallelism that are not amenable to parallelization by other techniques.

The Galois approach provides two significant benefits over existing dynamic approaches. First, the programmer determines where parallelization may be profitable, as indicated by the optimistic iterators; this prevents the Galois run-time from overspeculating, which would otherwise lead to a waste of hardware resources. Second, the usercode provides crucial semantic information regarding ordering constraints to the run-time, allowing the run-time to make intelligent scheduling decisions which can have a significant impact on performance, as we saw in Chapter 5.

We developed the first optimistic parallelization system that can exploit key semantic properties of algorithms and data structures to obtain significant parallel performance.

Exploiting abstractions: The fundamental approach in all aspects of the design of the Galois system has been to raise the level of abstraction that program-

mers use when writing parallel programs. As we have already discussed, the user code makes use of two abstractions to express amorphous data-parallelism, which naturally capture the potential parallelism inherent in a program as well as provide key semantic information to the run-time.

The Galois system also provides a higher-level shared memory abstraction. Rather than focusing on individual reads and writes, as in most parallelization schemes, Galois focuses on *shared objects* and the methods that manipulate them. By expressing all accesses to shared memory in terms of method invocations on objects, we are able to leverage the semantics of shared objects in several ways: semantic commutativity allows us to precisely capture when concurrently executing iterations are in fact dependent; semantic information about object effects allow us to provide semantic rollback; partitioning information allows us to capture semantic locality and use it to perform locality-enhancing scheduling as well as reduced-overhead conflict detection.

We have shown how programmers can easily capture many key semantic properties of their algorithms and data structures, and clearly demonstrated the utility of these abstractions in successfully exploiting parallelism in irregular programs.

6.3 Future work

While this thesis presents a viable foundation for a study of parallelism in irregular programs, the Galois system as presented is only a first step to fully exploiting the parallelism in irregular programs. There are several promising avenues for future work.

Compiler analysis: To this point, we have only considered run-time approaches to parallelizing irregular applications; we use no compiler analyses.

However, there are several interesting areas where compiler analysis can be useful. For example, analysis can be useful in determining which objects are truly shared in a system; the run-time system does not need to track objects that are only accessed by one iteration at a time, reducing overheads.

Compiler analysis can also be useful in determining a "point of no return" for iterations. For example, in Delaunay mesh refinement, once the extent of a cavity has been determined, it is no longer necessary for an iteration to roll back; this is the point of no return. At this point, no conflict checks need to be performed, as there is no way for an iteration to come into conflict with another. This can further reduce overheads.

Verification: The Galois system makes use of a multitude of object semantics when making parallelization and scheduling decisions. In the current implementation, we assume that the semantics as specified by the programmer are correct. However, it may be possible to bring to bear formal verification tools to ensure that, for example, the commutativity conditions specified by a programmer do indeed match the semantics of an object.

Scalability: We have only studied the performance of the Galois system on small scale multicore systems. The performance of this particular implementation, and indeed of the approach in general, is still untested on larger scale systems. Simple commutativity checks may not be feasible on large-scale systems, as they require centralized data structures. By performing studies on larger systems, we can also investigate performance optimizations such as architecture-aware abstract domain mapping (as discussed in Section 4.2.2).

Expanding the scope of parallel execution: To this point, we have only investigated programs with single iterators. It may be beneficial to extend the Galois system to exploit the parallelism in nested iterators (as discussed in Section 3.2.4), or "pipelined" iterators, where one loop produces work that is consumed by a second loop. Expanding the scope of parallelism in this manner opens up interesting areas of study for scheduling and locality.

An pessimist sees the difficulty in every opportunity;
An optimist sees the opportunity in every difficulty.

— Sir Winston Churchill

BIBLIOGRAPHY

[1] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[2] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

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

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[7] Richard J. Anderson and João C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, New York, NY, USA, 1992. ACM Press.

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

[9] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N. Scherer III, Chen Ding, and Michael L. Scott. A key-based adaptive transactional memory executor. In *Proceedings of the NSF Next Generation Software Program Workshop*. Mar 2007. Invited paper. Also available as TR 909, Department of Computer Science, University of Rochester, December 2006.

[10] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 2008. ACM.

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

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

[13] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[15] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM.

[16] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. *SIGPLAN Not.*, 38(1):213–223, 2003.

[17] 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.

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[19] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The ATOMOS

transactional programming language. In *PLDI '06: Proceedings of the Conference on Programming Language Design and Implementation*, 2006.

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

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

[22] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.

[23] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM.

[24] 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.

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

[26] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1993. ACM.

[27] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24, Vancouver, Canada, June 2000.

[28] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, 1998.

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

[30] Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Not.*, 35(5):35–46, 2000.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[32] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM.

[33] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM.

[34] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov 1976.

[35] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[36] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, 1998.

[37] Java Collections Framework. http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html.

[38] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM*

SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[40] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996. ACM.

[41] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in c programs with recursive data structures. In *Computational Complexity*, pages 159–173, 1998.

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

[43] 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.

[44] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[45] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.

[46] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 241–255, New York, NY, USA, 2001. ACM.

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

[48] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. *SIGOPS Oper. Syst. Rev. (ASPLOS 2008)*, 42(2):297–307, 2008.

[49] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32th Annual ACM Symposium on the Principles of Programming Languages*, Long Beach, CA, January 2005.

[50] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 2000.

[51] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, 1998.

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

[53] 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. ACM Press.

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

[55] Laurie J. Hendren, Joseph Hummell, and Alexandru Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 249–260, New York, NY, USA, 1992. ACM.

[56] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[57] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[58] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

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

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

[61] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[62] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[63] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[64] John Hogg. Islands: aliasing protection in object-oriented languages. *SIGPLAN Not.*, 26(11):271–285, 1991.

[65] S. Horwitz, 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.

[66] 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.

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

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

[69] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[70] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 244–256, New York, NY, USA, 1979. ACM.

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

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

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

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

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

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

[77] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[78] Milind Kulkarni, L. Paul Chew, and Keshav Pingali. Using transactions

in delaunay mesh generation. In *Workshops on Transactional Memory Workloads*, 2006.

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

[80] 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.

[81] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.

[82] 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.

[83] Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for java. *SIGPLAN Not.*, 42(10):423–440, 2007.

[84] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharaya, David Eisenstat, William N. Scherer, III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical report, Computer Science Department, University of Rochester, 2006.

[85] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[86] 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.

[87] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. *SIGPLAN Not.*, 41(11):359–370, 2006.

[88] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang,

and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[89] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *4th Workshop on Memory Performance Issues (WMPI)*, 2006.

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

[91] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 327–338, New York, NY, USA, 2007. ACM.

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

[93] 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.

[94] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 158–185, London, UK, 1998. Springer-Verlag.

[95] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM.

[96] OpenMP. http://www.openmp.org.

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

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

[99] Fred J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)(abstract only). In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[100] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[101] C. D. Polychronopoulos, D. J. Kuck, and D.A. Padua. Execution of parallel loops on parallel processor systems. In *Proc. 1986 Int. Conf. Parallel Processing*, 1986.

[102] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993.

[103] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[104] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. *SIGARCH Comput. Archit. News*, 33(2):494–505, 2005.

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

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

[107] Lawrence Rauchwerger and David A. Padua. Parallelizing while loops for multiprocessor systems. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, 1995.

[108] 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.

[109] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–67, 1996.

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

[111] R. Ronen, A. Mendelson, K. Lai, S-L. Lu, F. Pollack, and J. P. Shen. Coming challenges in microarchitecture and architecture. *Proc. IEEE*, 89(3):325–340, March 2001.

[112] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), May 2002.

[113] 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.

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

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

[116] 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.

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

[118] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[119] 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.

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

[121] Standard Template Library. http://www.sgi.com/tech/stl/.

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

[123] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

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

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

[126] Xinan Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, page 14, Washington, DC, USA, 1997. IEEE Computer Society.

[127] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th*

International Conference on Compiler Construction, pages 179–196, London, UK, 2002. Springer-Verlag.

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

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

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

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

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

[133] 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.

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

[135] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. *SIGPLAN Not.*, 30(6):1–12, 1995.

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

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