Scheduling Issues in Optimistic Parallelization

Milind Kulkarni and Keshav Pingali

University of Texas at Austin
Department of Computer Science
Austin, TX
{milind, pingali}@cs.utexas.edu

Abstract

Irregular applications, which rely on pointer-based data structures, are often difficult to parallelize. The input-dependent nature of their execution means that traditional parallelization techniques are unable to exploit any latent parallelism in these algorithms. Instead, we turn to optimistic parallelism, where regions of code are speculatively run in parallel while runtime mechanisms ensure proper execution. The performance of such optimistically parallelized algorithms is often dependent on the schedule for parallel execution; improper choices can prevent successful parallel execution.

We demonstrate this through the motivating example of Delaunay mesh refinement, an irregular algorithm, which we have parallelized optimistically using the Galois system. We apply several scheduling policies to this algorithm and investigate their performance, showing that careful consideration of scheduling is necessary to maximize parallel performance.

[1]

1 Introduction

When parallelizing a program, the first step is to determine what can be done in parallel safely. Traditionally, this has required determining regions of code which are completely independent. Much of the existing work on parallel programming has focused on parallelizing "regular" applications which deal with matrices and arrays. These applications are amenable to analysis, meaning that finding independent regions of code is relatively straightforward. However, a significant class of algorithms are "irregular," using pointer-based data structures such as trees and graphs. An exemplar of this class of algorithms is Delaunay mesh refinement, a widely used computational geometry algorithm which makes heavy use of sets and graphs.

Existing analysis techniques are often insufficient to accurately detect dependences in these types of algorithms. This is due not only to the complexity of pointer-based structures but also as a result of the input-dependent nature of the algorithms. Dependences between regions of code only become apparent at run-time. Thus, traditional techniques must conservatively assume that the dependences exist, and hence cannot exploit any parallelism in these applications.

One promising approach to parallelizing such applications is *optimistic parallelization*. These techniques assume that regions of code are independent, and allow them to execute concurrently. Because no *a priori* analysis is performed, run-time checks are used to detect whether the concurrent execution is, in fact, correct. If it is, then execution continues. If it turns out that the concurrent regions were not independent, a recovery mechanism is employed to ensure proper execution. In this way latent parallelism, which may only become apparent at run-time, can be exposed and exploited. We have developed a system for optimistic parallelization, called Galois, which can be easily applied to problems such as Delaunay mesh refinement.

The efficacy of optimistically parallelized algorithms is dependent on how often the speculative parallel execution is actually safe. If conflicts between concurrently executing regions of code are rare, then significant speedups may be achieved. If conflicts are frequent, then optimistic parallelization provides no benefit, as execution will essentially serialize. Thus, it is important to employ a suitable *scheduling policy* to effectively decide what code should be exe-

[1]This work was supported in part by the National Science Foundation under grant number 0615240. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

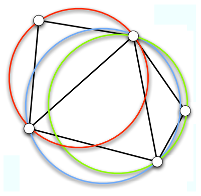

Figure 1. A Delaunay mesh. Note that the circumcircle for each of the triangles does not contain other points in the mesh.

cuted in parallel. The choice of scheduling policy can have a significant effect on performance.

The remainder of the paper is organized as follows. Section 2 discusses the example problem of Delaunay mesh refinement and how it may be parallelized. Section 3 surveys existing parallelization techniques and why they are unsuitable for problems such as Delaunay mesh refinement. Section 4 briefly describes the Galois system for optimistic parallelization. Section 5 explains the impact that scheduling policy can have on performance, Section 6 provides an evaluation of several policies on a Galois implementation of Delaunay mesh refinement and Section 7 concludes.

2 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 [2]. The Delaunay triangulation for a set of points in the plane is the triangulation such that no point is inside the circumcircle of any triangle (this property is called the *empty circle property*). An example of such a mesh is shown in Figure 1.

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose quality constraints governing the shape and size of the triangles. For a given Delaunay mesh, this is accomplished by *iterative mesh refinement*, which successively fixes "bad" triangles (triangles that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating. Figure 2 illustrates this process. To fix the bad triangle in Figure 2(a), a new point is added at the circumcenter of this triangle. Adding this point may invalidate the empty circle property for some neighboring triangles, so all affected triangles are determined (this region is called the *cavity* of the bad triangle), and the cavity is re-triangulated, as shown in Figure 2(c). Re-triangulating a cavity may generate new bad triangles but it can be shown

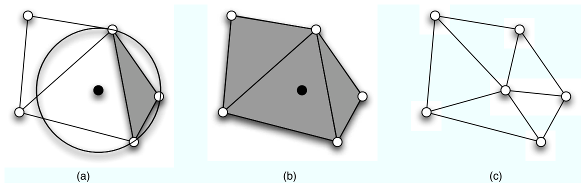

Figure 2. Fixing a bad element.

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

Figure 3. Pseudocode of the mesh refinement algorithm

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 [2].

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

Opportunities for Exploiting Parallelism. The natural unit of work for parallel execution is the processing of a bad triangle. Our measurements show that on the average, each unit of work takes about a million instructions of which about 10,000 are floating-point operations. Because a cavity is typically a small neighborhood of a bad triangle, two bad triangles that are far apart on the mesh may have cavities that do not overlap. Furthermore, the entire refinement process (expansion, retriangulation and graph updating) for the two triangles is completely independent; thus, the two triangles can be processed in parallel. This approach obviously extends to more than two triangles (see Figure 4). If however the cavities of two triangles overlap, the triangles can be processed in either order but only one of them can be processed at a time. Whether or not two bad triangles have overlapping cavities depends entirely on the structure of the mesh, which changes throughout the execution of the algorithm. Full details of how this algorithm can be parallelized are available in [5]

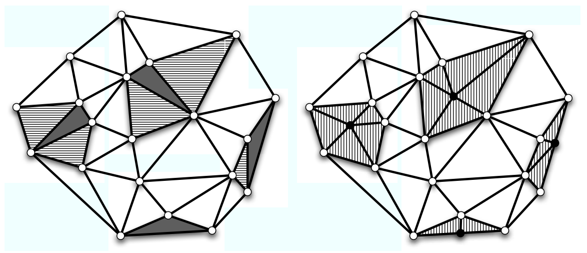

Figure 4. An example of processing several elements in parallel. The left mesh is the original mesh, while the right mesh represents the refinement. In the left mesh, the *dark grey* **triangles represent the "bad" elements, while the** *horizontally shaded* **are the other elements in the cavity. In the right mesh, the the** *black* **points are the newly added points and** *vertically shaded* **triangles are the newly created elements.**

How much parallelism is there in Delaunay mesh generation? The answer obviously depends on the mesh and on the order in which bad triangles are processed, and may be different at different points during the execution of the algorithm. One study by Antonopoulos *et al.* [1] on a mesh of one million triangles found that there were more than 256 cavities that could be expanded in parallel until almost the end of execution.

3 Existing Techniques

3.1 Pessimistic Parallelization

The standard approach to parallelizing a sequential code requires that regions of code which are independent of one another be identified. Determining the independence of two regions can either be done manually or through the use of dependence analyses. Because problems such as Delaunay mesh refinement make heavy use of pointer based data structures, various alias analyses [4] and shape analyses must be used [9].

Once independent regions of code are identified, they can then be executed in parallel. Thus, if all the iterations of a loop are independent (as in a Fortran-style *do-all* loop), the loop can be executed in parallel.

While this is a natural method of exploiting parallelism in many loop-based codes, a brief inspection shows that it is not immediately applicable to Delaunay mesh refinement. First, the worklist creates a loop-carried dependence throughout the loop; every iteration depends on the new state of the worklist created by the previous iteration. Even disregarding this dependence (which can be partially mitigated through the use of work-sharing constructs such as OpenMP's parallel-for [3]), we cannot guarantee that executing the iterations concurrently will produce the desired result.

Recall that triangles can be processed in parallel *only if their cavities do not overlap*, a property which cannot be determined *a priori*, as this is entirely dependent on the input data. Because dependence analyses are conservative, they will determine that any two iterations may depend on one another. Traditional parallelization techniques thus demand that we execute the loop sequentially; the potential dependence between iterations proscribes any parallelization.

3.2 Inspector-Executor Parallelization

One approach to taking input data into account in generating the parallel schedule is the *inspector-executor* approach [6]. This approach splits the computation 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. The inspector can sometimes be generated automatically by a compiler from the source program.

Because the inspector phase determines the schedule at runtime, it can use the input data in computing the schedule. This approach is not useful for our applications since key data structures such as the mesh structure in Delaunay mesh generation change at each iteration as the codes execute, so the inspector must do all the work of the executor.

3.3 Optimistic Parallelization

We instead turn to *optimistic parallelization*. We note that while there *may* be a dependence between any two iterations, with high probability, there won't be. Optimistic techniques speculatively execute regions of code concurrently in multiple threads while relying on some run-time mechanism to determine if dependences exist between the two regions. If no dependence exists, then the parallel execution was successful and execution continues. If a dependence is detected, then the two regions of code cannot be safely executed in parallel and one of the threads of execution is rolled back. Thus, it is possible to extract parallelism from regions of code which are often, but not always, independent.

Optimistic parallelization of loops was proposed by Rauchwerger and Padua in [8], and hardware implementations of the run-time conflict detection and rollback systems were discussed in several papers on Thread Level Speculation [10, 7]. However, these approaches do not apply easily

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

Figure 5. Delaunay mesh refinement using set iterator

to programs such as Delaunay mesh refinement which are based on dynamic worklists.

4 The Galois System

We have developed an object-based optimistic parallelization system called *Galois* which allows complex irregular applications to be parallelized relatively easily. For lack of space, we give only the high level details.

The Galois memory model is a concurrent, shared-memory execution model. The shared memory is organized as a collection of objects whose methods are invoked by threads to manipulate the internal state of these objects. There are three main aspects to Galois: (i) syntactic constructs for packaging optimistic parallelization, (ii) assertions about methods in class libraries, and (iii) a runtime system for detecting and recovering from unsafe accesses made by an optimistic computation to shared memory.

4.1 Concurrency constructs and execution model

The Galois system exposes parallelism through the use of simple *loop iterators*. These constructs, which take the form `foreach e in set S do B(e)`, indicate several things to the Galois system: (i) the loop should be parallelized, (ii) the iterations of the loop can be executed in any order (as a `set` has no ordering constraints) and (iii) there may be dependences between the various iterations. This allows us to rewrite the Delaunay code in a very natural way, as seen in Figure 5.

When a program begins execution, there is a single master thread. When the thread encounters an iterator, it spawns some number of worker threads to aid in the concurrent execution of the iterator. During this concurrent execution, it is important that the system preserve the sequential semantics of the iterator. Thus, the semantics of the set iterator require that the final execution behave as if the iterations were executed in some serial order.

4.2 Writing Galois Classes

Implementing the semantics of iterators in a parallel execution is a non-trivial problem because each iteration may manipulate a number of shared objects, such as the work list and kd-tree in our applications, and method invocations from different concurrent iterations get interleaved by the objects. Therefore, it is important to determine when two iterations can safely invoke methods on the same object since disallowing this concurrent access to shared objects severely restricts the amount of concurrency available.

Semantic Commutativity To permit multiple iterations to access shared objects concurrently without violating transactional semantics, the Galois system exploits *commutativity* of method invocations: if two concurrent iterations invoke the methods of a shared object, these invocations can be interleaved without violating transactional semantics if the invocations commute.

Because the internal state of objects is never visible, 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, *add(x)* commutes with *remove(y)*, not with *remove(x)*. Commutativity information must be provided by the class implementor.

Inverse Methods Because iterations are executed in parallel, it is possible for commutativity conflicts to prevent an iteration from completing. Once a conflict is detected, some recovery mechanism must be invoked to allow execution of the program to continue despite the conflict. To permit this, every method of a shared object that may modify the state of that object must have an associated *inverse* method that undoes the side-effects of that method invocation. For example, for a set, the inverse of *add(x)* is *remove(x)*, and the inverse of *remove(x)* is *add(x)*. This information too must be provided by the class implementor.

4.3 Runtime System

The Galois runtime system is responsible for maintaining the sequential semantics of the parallel iterator. It does so by monitoring the concurrent execution of iterations and detecting when iterations conflict. Each shared object keeps track of which methods have been invoked by currently executing iterations (called "outstanding invocations"). When an iteration attempts to invoke a method, it is checked for conflicts against all outstanding invocations. If no conflicts are found, the conflict log is updated and execution continues. Otherwise, one of the conflicting iterations is rolled back and restarted. In this manner, the sequential semantics are preserved.

5 Scheduling

Optimistic parallelism is a viable approach to concurrent execution as long as the optimism is warranted. If there are infrequent conflicts and most concurrent iterations complete successfully, then forward progress is rapidly made and significant speedup can be achieved when executing in parallel. However, if iterations conflict often (*i.e.* there are few independent iterations), there is essentially no benefit to optimistic parallelization; most iterations will simply be rolled back, and only one iteration will complete at a time. Effectively, the computation will serialize.

Even if a problem has a significant amount of potential parallelism, this is not enough to guarantee that the optimistic execution will successfully exploit it. If the ordering of iterations is unspecified in the algorithm, we can think of different *scheduling policies*. These policies determine which iterations should be executed concurrently. A good scheduling policy will minimize the number of concurrent iterations which conflict with one another and lead to efficient parallel execution, while a bad scheduling policy will lead to a large number of conflicts and lead to highly inefficient parallel execution. Note that even though the majority of iterations in Delaunay mesh refinement will not conflict (as they will operate on distant portions of the mesh), it is still possible to schedule iterations poorly such that execution will serialize.

5.1 Abort Ratios

To allow us to compare various scheduling policies, we introduce the concept of *abort ratio*. The abort ratio is the ratio of the number of iterations which are rolled back to the total number of iterations executed (whether completed successfully or rolled back). Thus, the abort ratio gives us an estimate of the efficacy of a given scheduling policy. An ideal schedule, which results in no conflicts between concurrent iterations, would have an abort ratio of zero. A poor schedule, on the other hand, means that only one iteration at a time will successfully execute while the rest roll back. This will result in a abort ratio which approaches 100% (note that because the total number of executed iterations includes successful ones, the ratio will never reach 100%).

While there is a correlation between abort ratio and parallel efficiency, it may not always be significant; the cost of the rollback is important as well. The major loss of efficiency due to rollbacks can be attributed to lost work. A rolled back iteration means that whatever processing time was spent executing the iteration was wasted. However, if rollbacks occur at the beginning of an iteration, not much work is lost due to the rollback. Thus, it is possible to have a high abort ratio without impacting parallel efficiency much.

5.2 Policy Choices and Implementation

In the Galois system, different scheduling policies can be applied by varying the implementation of the worklist being iterated over. The worklist supports a getAny method which is used by the runtime system to select elements to process and an add method by which new work is placed on the worklist. Thus, the implementation of the two methods controls the schedule of execution. We can consider several different scheduling policies for Delaunay mesh refinement:

- **Queue**: The simplest implementation of the worklist is a queue. The getAny method removes elements from the front of the worklist, while the add method places new elements at the back. This is the default implementation in the sequential Delaunay mesh refinement code.
- **Randomized**: The randomized scheduling policy requires that getAny choose elements at random from the worklist.
- **Partitioned**: Another policy which aims to reduce the abort ratio when running on multiple processors is partition-based scheduling. In this policy, the mesh is logically partitioned, and each processor is given elements from different partitions.

6 Evaluation

We evaluated the performance of these scheduling policies when applied to a Galois implementation Delaunay mesh refinement. The test system was a 4-processor Itanium 2 machine, with each processor running at 1.5 GHz, running Red Hat Linux. We first determined the abort ratio of the three policies when running on 4 processors, as seen in Table 1. We see that the partitioned schedule had the smallest abort ratio, closely followed by the randomized schedule, both of which eliminated almost all rollbacks. The queue schedule performs the worst. To understand these results, we instrumented the codes to determine the origins of these conflicts. It turned out that when a bad triangle is fixed, it might produce a bunch of smaller bad triangles in its cavity. In the queue implementation of the worklist, these bad triangles are next to each other, so with high probability, they are scheduled for execution on different processors at roughly the same time. Their cavities will conflict with high probability, and this leads to high abort ratios. The randomized data structure reduces the probability

Scheduling Policy	Abort Ratio (%)
Queue	56.14
Randomized	0.8432
Partitioned	0.1601

Table 1. Abort ratios for different scheduling policies when running the Galois implementation of Delaunay mesh refinement on 4 processors

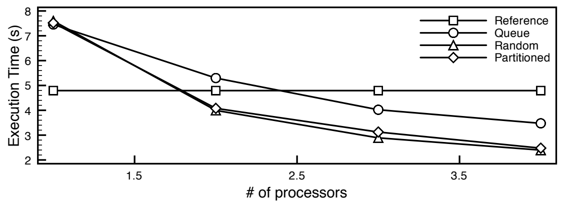

Figure 6. Execution time vs. Number of processors

of conflicts substantially, as does the partitioned approach, and this improves performance.

We also examined the execution time and speedups (execution time divided by single-processor execution time) for the three different scheduling policies. We also compared them against a reference, sequential implementation of mesh refinement. Results for execution time are in Figure 6, and those for speedup are in Figure 7.

As expected from the abort ratios, randomized scheduling and partitioned scheduling performed the best, each achieving speedups of over 3 on four processors. Interestingly, despite the lower abort ratio for partitioned scheduling, its speedup is slightly less than randomized scheduling. This is likely due to small amounts of load imbalance, which manifest during the last stages of execution. As reflected in its abort ratio, the default queue scheduling policy performs quite poorly in comparison to more informed policies.

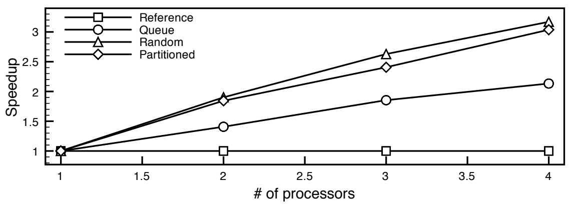

Figure 7. Speedup vs. Number of processors

7 Conclusions

It is clear that optimistic parallelization is the only feasible approach to parallelizing many types of irregular applications, such as Delaunay mesh refinement. As a result, a more careful consideration of the issues surrounding optimistic parallelism is clearly necessary. Successfully exploiting speculation demands that such speculation have a high success rates. Thus it is very important to consider scheduling in the context of optimistic techniques.

We have presented a number of different potential scheduling policies, and evaluated them in the context of an optimistic implementation of Delaunay mesh refinement. We have found that naive scheduling (*e.g.* relying on the default iteration ordering from the sequential implementation) can lead to a substantial performance penalty. More considered approaches to scheduling are clearly necessary, as demonstrated by the higher performance of the randomized and partitioned policies. We feel that a careful study of the behavior of various algorithms can lead to highly efficient scheduling policies which fully exploit any available parallelism.

References

[1] 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*, pages 367–376, New York, NY, USA, 2005. ACM Press.

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

[3] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[4] 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.

[5] Milind Kulkarni, L. Paul Chew, and Keshav Pingali. Using transactions in delaunay mesh generation. In *Workshops on Transactional Memory Workloads*, 2006.

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

[7] 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.

[8] Lawrence Rauchwerger and David Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 218–232, New York, NY, USA, 1995. ACM Press.

[9] 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.

[10] 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.