On the Scalability of an Automatically Parallelized Irregular Application

Martin Burtscher, Milind Kulkarni, Dimitrios Prountzos, and Keshav Pingali

Center for Grid and Distributed Computing
Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, TX 78712
{burtscher, milind}@ices.utexas.edu, {dprountz, pingali}@cs.utexas.edu

Abstract. Irregular applications, i.e., programs that manipulate pointer-based data structures such as graphs and trees, constitute a challenging target for parallelization because the amount of parallelism is input dependent and changes dynamically. Traditional dependence analysis techniques are too conservative to expose this parallelism. Even manual parallelization is difficult, time consuming, and error prone. The Galois system parallelizes such applications using an optimistic approach that exploits higher-level semantics of abstract data types.

In this paper, we study the performance and scalability of a Galoised, i.e., automatically parallelized, version of Delaunay mesh refinement (DR) on a shared-memory system with 128 CPUs. DR is an important irregular application that is used, e.g., in graphics and finite-element codes. The parallelized program scales to 64 threads, where it reaches a speedup of 25.8. For large numbers of threads, the performance is hampered by the load imbalance and the nonuniform memory latency, both of which grow as the number of threads increases. While these two issues will have to be addressed in future work, we believe our results already show the Galois approach to be very promising.

1. Introduction

Over the last three decades, the problem of automatic parallelization, i.e., the mechanical transformation of sequential code into parallel code by identifying program regions that can execute concurrently, has been studied extensively. As a result, modern compilers are able to achieve very good parallel performance in certain application domains virtually without programmer guidance.

In particular, for applications that process arrays and matrices, which we refer to as "regular" applications, a multitude of techniques have been developed to prove independence between array accesses and to uncover, package, and schedule parallelism at various levels [6]. In this class of programs, data parallelism mainly manifests itself as FOR-ALL loops over integer intervals for which the iterations can be statically proven to be independent.

However, there exist a significant number of important "irregular" applications that manipulate pointer-based data structures such as trees and graphs, which are much harder to parallelize. A characteristic example of this class of programs is Delaunay

mesh refinement, a widely used computational geometry algorithm. For these applications, static parallelization techniques based on pointer [4] and shape analysis [5], [11], [16] are often insufficient because they must be correct for all possible inputs. However, the amount of parallelism in irregular applications is almost always data dependent. For example, in Delaunay refinement, it is highly dependent on the shape of the input mesh. Thus, statically produced parallel schedules tend to be overly conservative for most inputs and unnecessarily serialize program execution.

In semi-static approaches, the computation is split into an inspector phase, which determines the dependences between units of work, and an executor phase, which uses this schedule to perform the computations concurrently [14]. Since the inspector is also executed at runtime, the input of the program is taken into account when producing the schedule. For Delaunay refinement, the usefulness of this approach is, however, limited because the mesh changes as the algorithm progresses. Hence, the inspector would have to be executed repeatedly, which is expensive because it involves expanding the cavities (i.e., a substantial part of the work of a single iteration).

The most promising way to automatically parallelize “irregular” programs is employing dynamic approaches that speculatively parallelize the code at runtime. In this approach, portions of the application are executed in parallel assuming that dependences are not violated. The runtime system is responsible for detecting any such violations and for restoring the program to the correct state by aborting one of the conflicting computations and executing it later. If no dependence violation is detected, the speculative state is committed, thus becoming visible to the rest of the program.

In previous work, we introduced the Galois system [10], which we discuss in more detail below, to automatically and speculatively parallelize irregular code. While we believe our system to be practical, our previous studies [8], [9], [10] have not investigated the scalability beyond small multicore systems. The goal of this paper is to study the performance of an application that has been automatically parallelized by the Galois system on a large-scale shared-memory multiprocessor. This study not only provides insight into the effectiveness of our approach but also brings out important issues pertaining to the parallelization of irregular applications in a real-world setting.

The rest of the paper is organized as follows. Section 2 discusses the Delaunay mesh refinement algorithm in detail. Section 3 illustrates the Galois system. Section 4 presents the experimental methodology. Section 5 shows the results. Section 6 summarizes related work. Section 7 concludes the paper with a summary and future work.

2. Delaunay Mesh Refinement

Mesh generation is a vital component of many applications in graphics and the numerical solution of partial differential equations. The goal of mesh generation is to represent a surface or a volume as a tessellation composed of simple shapes like triangles or tetrahedra.

Although many types of meshes are used in practice, Delaunay meshes are particularly important since they have a number of desirable mathematical properties [3]. The Delaunay triangulation of 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 given in Fig. 1.

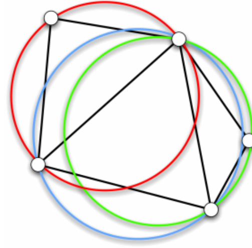

Fig. 1. Delaunay mesh (the circumcircles of the triangles contain no mesh points).

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 (i.e., triangles that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating it. Fig. 2 illustrates this process.

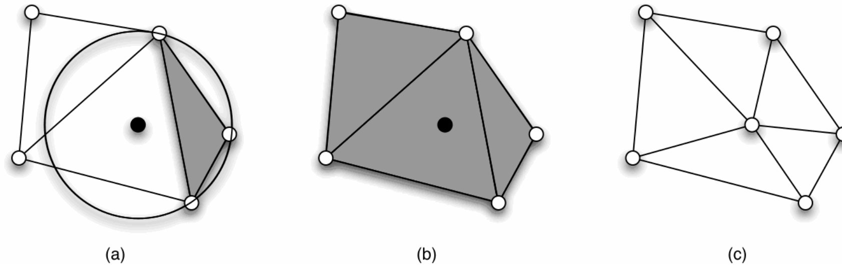

Fig. 2. Delaunay mesh refinement steps.

The shaded triangle in Fig. 2(a) is assumed to be bad. To fix it, a new point is added at the center of this triangle's circumcircle. Adding this point may invalidate the empty circle property of some neighboring triangles. Hence, all affected triangles need to be determined. This region is called the *cavity* of the bad triangle and is shaded in Fig. 2(b). In this example, all triangles belong to the cavity, but in larger meshes, a cavity usually only covers a small fraction of the mesh. In the final step, the cavity is re-triangulated as shown in Fig. 2(c). Re-triangulating a cavity may generate new bad triangles, but it can be proven that this iterative refinement process will ultimately terminate and produce a guaranteed-quality mesh [3]. Different orders of processing bad triangles may lead to different meshes, but all such meshes satisfy the quality constraints.

Fig. 3 provides pseudocode for mesh refinement. The input 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 algo-

rithm. One is a worklist containing the bad triangles in the mesh. The other is a graph representing the mesh structure where the nodes correspond to the triangles and the edges denote triangle adjacencies. The two-dimensional algorithm works as follows.

1. Find all the bad triangles in the mesh and put them into the worklist [line 3]. Then repeat the following steps until the list of bad triangles is empty [line 4].
2. Pick a triangle from the list [line 5]. The processing of other bad triangles may have removed this triangle from the mesh. If so, there is nothing to do [line 6].
3. Compute the cavity of the bad triangle as follows. Find the circumcenter of the triangle, add this new point to the mesh and determine the triangles that no longer satisfy the empty circle property because of this new point [lines 7 and 8].
4. Re-triangulate the cavity [line 9].
5. Replace the triangles in the cavity with the new triangles (i.e., remove the old triangles from the mesh and add in the newly calculated triangles) [line 10].
6. Because the newly created triangles are not guaranteed to meet the quality constraints, any newly created bad triangles must be added to the worklist [line 11].

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

Fig. 3. Pseudocode of the 2D mesh refinement algorithm.

2.1. Opportunities for Exploiting Parallelism

The natural unit of work for parallel execution in Delaunay mesh refinement is the processing of a bad triangle. Because a cavity is typically a small neighborhood of a bad triangle, the cavities of two bad triangles that are far apart in the mesh often do not overlap and can therefore be processed concurrently.

An example of processing several triangles in parallel is given in Fig. 4. The left mesh is the original mesh, and the right mesh represents its refinement. In the left mesh, the *black* triangles are the bad triangles while the *dark grey* triangles are the other triangles in the cavities. In the right mesh, the *black* points mark the newly added points and the *light grey* triangles denote the newly created triangles. Clearly, all the cavities in Fig. 4 can be refined in parallel without conflicts. Thus, Delaunay mesh refinement is an example of a worklist algorithm where the units of work may be independent.

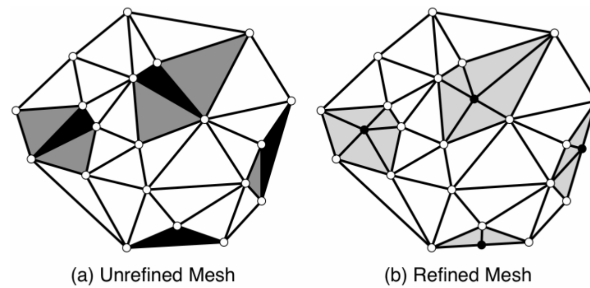

Fig. 4. Processing triangles in parallel.

3. The Galois Model

The Galois programming model [10] is a concurrent, object-based, shared-memory model that is designed to be implemented as an extension to an object-oriented language. An application parallelized under this model consists of two components: the client code, which is written by the user of the system and has easily-understood sequential semantics, and the library and runtime code, which encapsulates all the complexity of parallel execution.

3.1. Client Code

The programming model provides two language constructs, called optimistic set iterators, that allow the user to implicitly express parallelism. The well-defined sequential semantics of these set iterators makes it easier to understand, write, and debug client code.

- ***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 the elements of a set are not ordered, this construct denotes that, in a serial execution of the loop, the iterations can be executed in any order. There may be dependences between the iterations, as is the case with Delaunay mesh refinement, but any serial order of executing iterations is permitted. Iterations may dynamically add elements to S.
- ***Ordered-set iterator***: `for each e in OrderedSet S do B(e)`
  This construct denotes a partially-ordered iterator over S. Contrary to the Set iterator, the execution order must respect the partial order imposed by the OrderedSet S.

3.2. The Galois Runtime and Class Libraries

The runtime system speculatively executes iterations of set iterators in parallel, thereby taking advantage of potential data-parallelism in the application. To guarantee that the parallel execution preserves the sequential semantics of the iterators, the system

must ensure that concurrent accesses of and method invocations on shared objects are properly coordinated.

One way to detect conflicts in the Galois system is through *commutativity checks*. Intuitively, two iterations that concurrently access a shared object do not conflict if they call commuting methods on the object. Note that commutativity conditions are a property of the abstract data type of the object and are therefore only dependent on the public interface of the type and not on the concrete implementation of that interface. Thus, the implementation can be changed without affecting the commutativity conditions. These conditions are specified as annotations in the class definition [10].

Alternatively, conflict detection can be performed on *partitioned* data structures [9]. For example, a Delaunay mesh can easily be partitioned. Whenever two iterations touch the same partition of a shared data structure, a conflict is raised. Even though this scheme is less precise than commutativity checking, it is simpler and has a lower overhead, which is why we use it in this study.

The Galois system also supports overdecomposition. The basic idea of overdecomposition is to partition the data into more partitions than there are cores in the machine so that multiple partitions are mapped to each core. When a thread accesses a partition of a data structure, it owns all elements in that partition, and the other threads are not allowed to access them. Assigning multiple partitions to a core increases the probability that a thread can continue to perform useful work even if other threads have temporarily locked some of its partitions [9].

Whenever a conflict is detected, one or more iterations must be rolled back, i.e., a series of *undo actions* are executed by the runtime system. For each method of a shared object type, the class implementor must provide another method that performs a "semantic undo". For example, the undo of *add(x)* in a set is *remove(x)*. As each iteration executes, the system records the undo actions corresponding to the methods that get called and uses them to perform a rollback if a conflict occurs.

4. Methodology

To study the scalability and other performance aspects of our Galoised version of Delaunay mesh refinement, we performed experiments on a Sun E25K server running SunOS 5.9. The system contains sixteen CPU boards with four dual-core 1.05 GHz UltraSPARC IV processors. The 128 CPUs share 512 GB of main memory. Each core has a 64 kB four-way set-associative L1 data cache and a unified 8 MB L2 cache.

We use Sun's Java compiler version 1.6.0_02 and the HotSpot 64-bit server virtual machine version 1.6.0-b105. Because HotSpot dynamically compiles frequently executed bytecode into native machine code, we repeat each experiment nine times in the same VM and report results for the median as well as the fastest run. To prevent other jobs from interfering with our measurements, we always reserve all 128 CPUs regardless of how many threads we create. Furthermore, to minimize the interference by the garbage collector, we use a 400 GB heap and force a garbage collection by calling System.gc() five times before executing the measured code section.

All measurements are obtained through source code instrumentation; that is, we read the timer and the CPU performance counters before and after the measured code

section, compute the difference, and write the result to the standard output. We use the Java Native Interface and C code we wrote to access the performance counters.

We evaluate Delaunay mesh refinement on three random inputs. The small input contains 100,770 triangles of which 47,768 are initially bad. The middle input has 219,998 triangles of which 104,229 are initially bad. The large input consists of 549,998 triangles of which 261,100 are initially bad.

All measurements in this study refer to the refinement algorithm only. In particular, reading the input, building the initial graph, and partitioning the initial graph are excluded as we have not yet Galoised (i.e., parallelized) these components of the code.

5. Results

5.1. Speedup

Fig. 5 shows the speedup of the parallel code on the three inputs for various thread counts relative to the fastest run of our sequential implementation. The solid lines display the best and the dashed lines the median speedups. The overdecomposition factor is 32. There was no garbage collection during the execution of the timed code. The sequential refinement code takes 31.96, 76.16, and 195.1 seconds, respectively, for the small, medium, and large input.

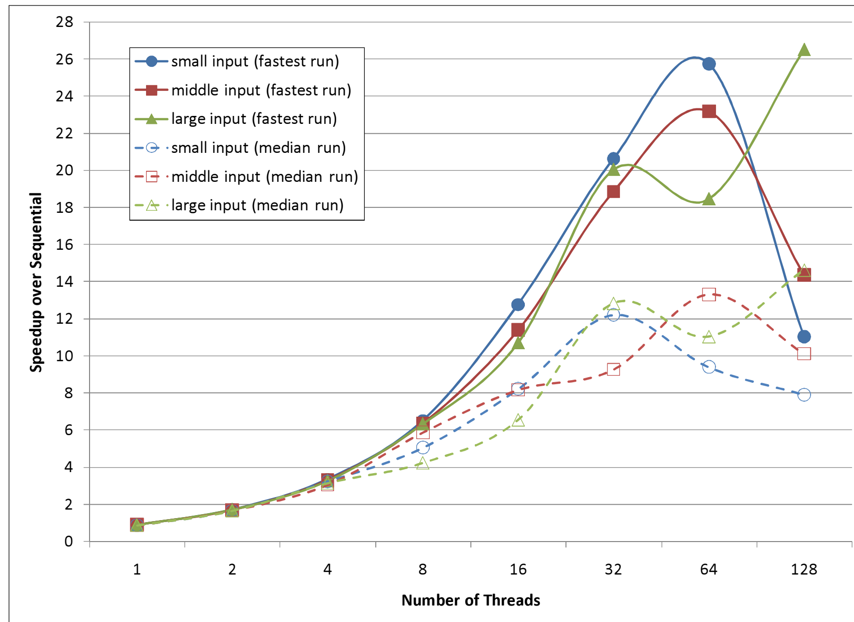

Fig. 5. Speedup over the fastest sequential run.

The automatically parallelized code scales to 64 or 128 threads, depending on the input. Scaling is good (over 50% efficiency) up to 32 threads, where the speedup is roughly 20 for all three inputs. The large input scales to 128 threads, where it reaches a speedup of a 26.5, the highest we observed. The performance drop with 64 threads is due to a high CPI (cf. Section 5.2), which we believe is caused by unfortunate partitioning that results in a large amount of communication.

The median runtimes start to diverge from the fastest runtimes at eight threads because of slow communication between CPU boards. Recall that our machine comprises sixteen boards with eight processors each. Thus, experiments with eight or fewer threads may incur only intra-board communication whereas experiments with sixteen or more threads necessarily incur inter-board communication, which is slow. Hence, as long as the operating system executes all threads on CPUs of the same board, the runtimes of the nine experiments vary only slightly and the median is very close to the fastest runtime. However, as soon as multiple boards are involved, which may already happen with eight worker threads because of other JVM threads, it greatly matters whether threads that exchange data are assigned to the same board or not. Because some assignments result in better locality than others, the random allocation of threads to cores by the operating system yields different runtimes for each experiment, as is reflected by the discrepancy of up to a factor of 2.7 between the median and the best speedups. Due to this high variability, we believe the results reported in this paper for large numbers of threads show the correct trends but the absolute values might be unreliable.

The parallel code with one thread is 13% slower than our sequential implementation. This result reflects the overhead introduced by Galois, which includes the time it takes to start and terminate worker threads, the cost of checking for runtime conflicts (even though no conflicts can occur with just one thread), and the expense of recording the undo information.

In summary, the scaling is surprisingly good for an automatically parallelized irregular application. The efficiency is better than 66.6% up to 16 threads for the fastest runs. Above 16 threads, it drops quickly due to load imbalance and memory latency.

5.2. Memory Access Latency

To confirm the negative impact of the inter-board communication, we measured the average number of cycles it takes to execute an instruction. Fig. 6 shows the results. Because the same code is executed and because most instructions that do not access the memory have a fixed latency, we attribute any increase in the number of cycles per instruction (CPI) to slower memory accesses. Direct measurements of the L2 cache stall cycles corroborate our results but only capture part of the memory latency.

The CPI (and therefore the memory latency) starts to greatly increase above 16 threads, exposing the nonuniformity in the memory access time. The CPI with large numbers of threads is up to 2.44 times as high as the CPI with a single thread. Since instructions that touch the memory represent only a fraction of the executed instructions, the average slowdown per memory access is, of course, even higher. Thus, the slow inter-board communication speed is one of the primary performance hurdles in the Galoised refinement code for large numbers of threads on our system.

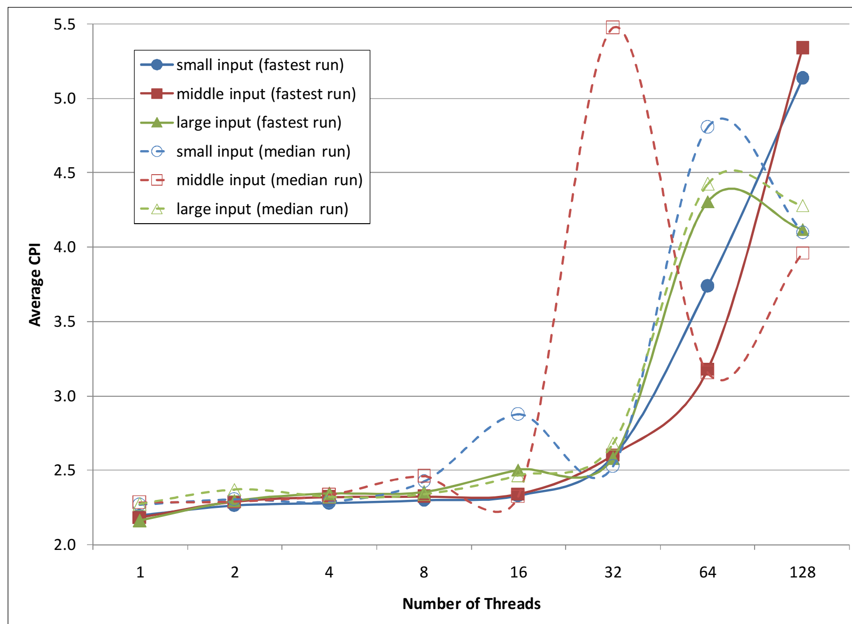

Fig. 6. Average cycles per instruction (CPI).

5.3. Load Balance

Fig. 7 illustrates the fraction of time that the concurrent threads have to wait, on average, for the slowest thread to finish. For example, in the 128-thread run with the small input, the threads idle 75.5% of the time, on average. Note that there is no task stealing and new work generated by a thread is always handled by that thread.

We observe very little load imbalance (under six percent idle time) up to eight threads. Above eight threads, the imbalance starts to become significant and for 128 threads, on average over half of the time the threads are idling for all three inputs. The load imbalance grows with the number of threads because larger thread counts result in less work per thread, which increases the likelihood of imbalance problems. Thus, load imbalance is the second main reason preventing the automatically parallelized code from scaling well to 128 CPUs. Note that, on the one hand, our random inputs are quite homogenous and thus probably more balanced than real inputs, meaning that load imbalance may be an even bigger problem in practice. On the other hand, we use a somewhat naive work partitioner based on recursive subdivision, and employing a more sophisticated partitioner may improve the load balance.

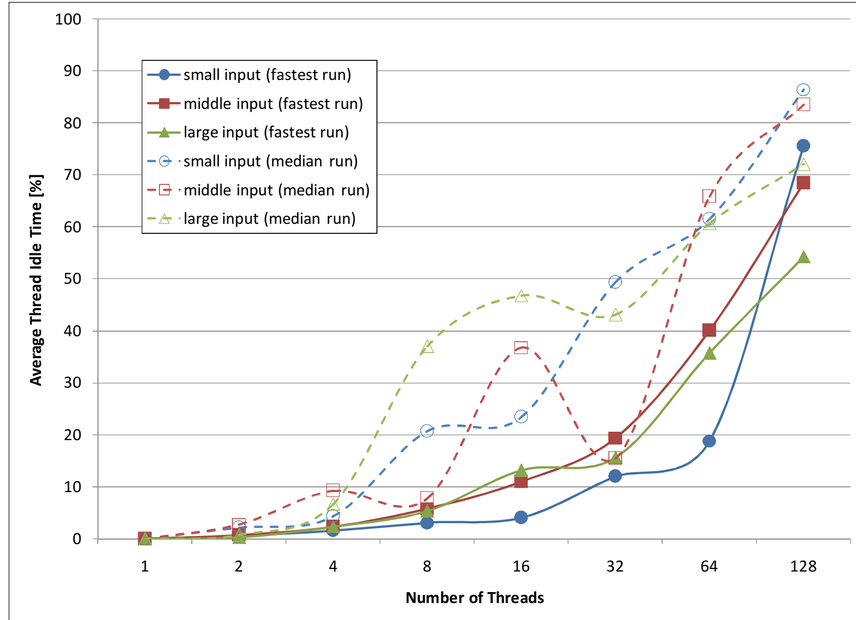

Fig. 7. Average thread waiting time as a percentage of the total runtime.

5.4. Aborted Speculations

Some of the speculative refinements fail because their cavity extends to a partition of the graph that is locked by another thread. Fig. 8 depicts the fraction of the attempted refinements that had to be aborted (and retried later). The overdecomposition factor is 32. In the median run with the large input, 18.8% of the refinements were aborted.

There are relatively few aborts, as one might expect with a large overdecomposition factor. With the fastest runs, we see almost no aborts up to four threads. Larger numbers of threads cause more aborts for two reasons. First, the higher latency of the aforementioned inter-board communications slows down some of the refinements, meaning that they take longer and are therefore more likely to conflict with other concurrent refinements. This is probably also the reason why the (slower) median runs sometimes have much higher abort ratios than the fastest runs. Second, increasing the thread count increases the number of partitions but makes them smaller. Hence, the chance of a cavity overlapping multiple partitions increases.

Nevertheless, the observed abort ratios are too low to severely impact the scalability. In fact, aborts are detected quite early, namely during the cavity expansion and before the actual refinement work. As a result, they only have a small effect on the runtime (cf. Section 5.6).

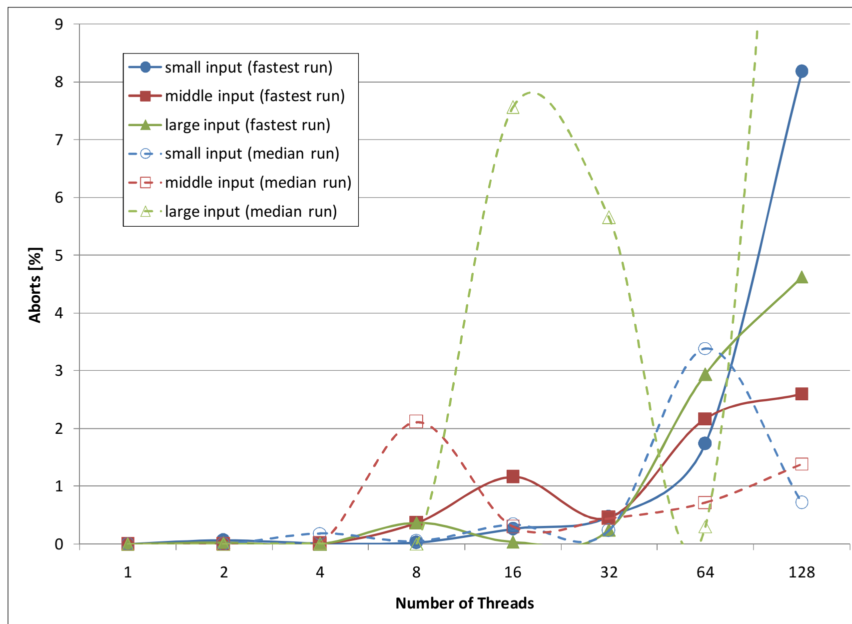

Fig. 8. Percentage of attempted refinements that were aborted due to speculation conflicts.

5.5. Overdecomposition

Fig. 9 illustrates the impact of the overdecomposition factor on the speedup over the sequential code. For improved readability, we only show results for the fastest runs with the middle input.

As one might expect, one partition per thread results in poor performance. Two partitions per thread are necessary and sufficient for good performance up to eight threads. For larger numbers of threads, higher overdecomposition factors tend to help because they lower the misspeculation rate of the optimistic execution. However, beyond a certain level of overdecomposition, there is little benefit in using smaller partitions. In fact, the locking overhead increases as the partitions become smaller because the cavities are more likely to span multiple partitions, which necessitates the acquisition of multiple locks. The odd behavior with one partition per thread and 128 threads may be an artifact of the aforementioned high variability in our measurements with large thread counts.

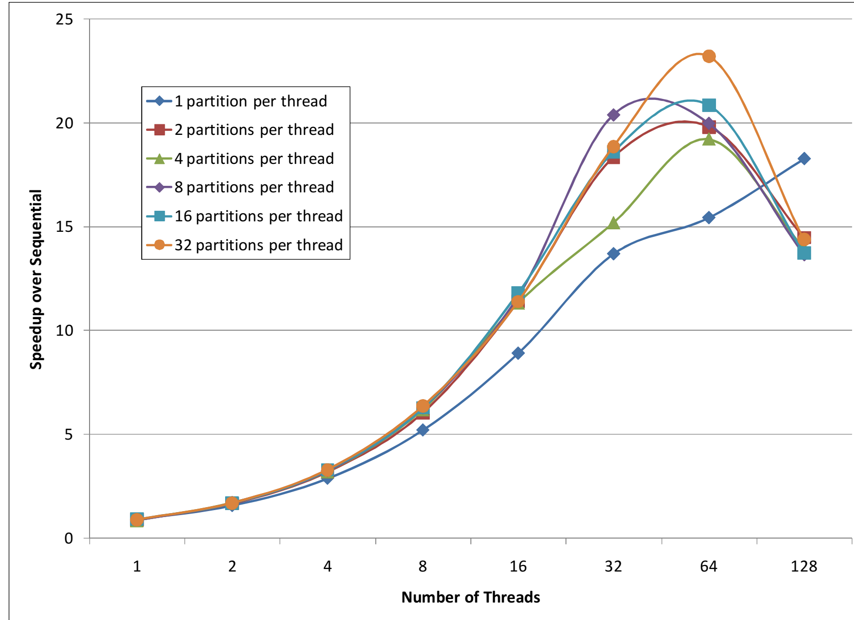

Fig. 9. Speedup of the fastest runs and middle input for different overdecomposition factors.

5.6. Result Summary

Fig. 10 summarizes the results from the previous subsections by accumulating the runtime of all threads. The figure shows numbers for the fastest run with the middle input. The results for the other two inputs are qualitatively similar. The runtimes are relative to the sequential runtime. Each bar is broken down into five categories. They are, from bottom to top, 1) the runtime of the sequential code, 2) the single-thread overhead, i.e., the runtime of the parallel code with just one thread, 3) the aborted work due to misspeculations, 4) the memory latency as computed in Section 5.2, and 5) the time the threads idle while waiting for the slowest thread to finish.

Because Fig. 10 sums up the runtime across all threads, the total runtime would be the same regardless of the number of threads if the parallel implementation scaled perfectly. However, as we noted in the previous subsections, there are factors that hamper the scalability of Delaunay mesh refinement. Up to four threads, the overheads of the Galois system remain low, and hence the total runtime stays roughly constant. However, beyond four threads we see that the load imbalance and the memory latency begin to increase rapidly. Both of these overheads are the result of processors being unable to perform useful work (either because of waiting for memory or for slower threads). For large numbers of threads, the load imbalance represents the biggest performance bottleneck followed by the memory latency. The aborted work and the parallelization (single-thread) overhead are minor in comparison.

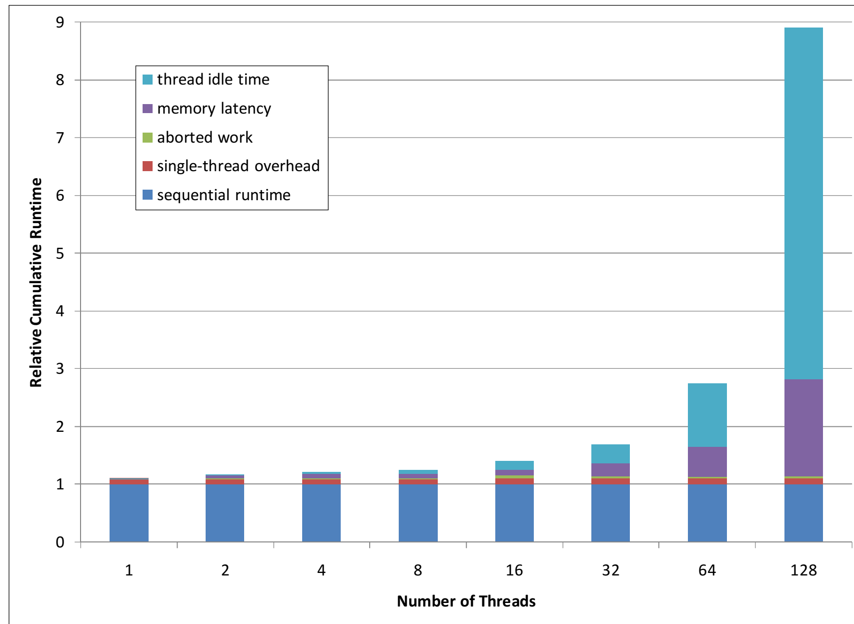

Fig. 10. Accumulative runtime breakdown for the fastest run with the middle input.

6. Related Work

Hand-written parallel implementations of Delaunay mesh refinement exist for 2D [1] and 3D [2] meshes. While both implementations eschew optimistic parallelization, they are not amenable to automatic parallelization (i.e., an automatic approach could not generate these particular parallel implementations from the sequential algorithm) for several reasons. In the 2D code, the mesh is partitioned among multiple processors, and each partition is processed relatively independently. However, the standard Delaunay algorithm is augmented with special handling for the boundaries between partitions. Thus, this approach is effectively a new algorithm for parallel Delaunay refinement rather than a straightforward parallelization of the sequential algorithm. In the 3D code, the mathematical properties of the Delaunay algorithm were examined and the authors developed a distance metric establishing the greatest possible size of a cavity in the mesh. Thus, regions sufficiently far apart can be processed in parallel. This approach to parallelization requires specific algorithmic knowledge and, again, is therefore not a straightforward parallelization of the sequential algorithm.

Other approaches to automatic parallelization of irregular programs include Thread Level Speculation (TLS) [7], [15]. This technique automatically parallelizes FOR loops in sequential programs using optimistic parallelization. However, because TLS focuses on parallelizing standard sequential programs, it cannot leverage key algo-

rithmic semantics in the parallelization. Thus, the generated parallel programs must exactly match the sequential program, preventing TLS from, e.g., reordering parallel computation to better exploit locality.

Another approach that has been studied extensively is Transactional Memory [12] (TM). One key distinction between the Galois approach and TM is that the latter is mainly concerned with *optimistic synchronization* as opposed to *optimistic parallelization*. In other words, the input program for a TM system has already been parallelized and the goal is to find an efficient and less error-prone way to synchronize the parallel tasks. In contrast, the main concern of the Galois model is to present the user with the right abstractions to express the parallelism in irregular codes as well as to provide an efficient implementation of those abstractions.

TLS and TM both detect speculative conflicts based on memory-level consistency. As we discuss elsewhere [10], tracking conflicts at such a low level may trigger false conflicts and thus disallow parallel execution that is actually safe. One way to overcome this issue in the case of Transactional Memory is to use open nested transactions [13]. This approach, however, complicates the semantics of the program and, as a result, increases the effort required by the programmer.

7. Conclusions and Future Work

This paper studies the scalability of an important irregular application, Delaunay mesh refinement, which has been automatically parallelized using the Galois system. Our measurements on a 128-CPU shared-memory computer identified the load imbalance and the long nonuniform memory latency (due to inter-board communication) to be the primary bottlenecks to scaling for large numbers of threads. Speculation aborts and the overdecomposition factor have a relatively minor impact on the performance. Overall, the automatically parallelized code scales to 64 or 128 threads, depending on the input, and achieves a maximum speedup of 26.5.

While this work only investigates a single application, it raises several issues that are known to be problematic in parallelization. Thus, we believe Delaunay mesh refinement to be a representative program worth studying and our findings to be more generally applicable. For instance, future multicore systems will likely also have nonuniform memory latency.

Addressing this issue is our primary target for future work. To minimize the interboard communication, i.e., the slowest memory accesses, we will hierarchically partition the work and pin it to CPUs such that the memory hierarchy (including the interconnection network) matches the hierarchy of the work partitions. To address the load imbalance, we will modify the work scheduler and add support for work or partition stealing. Other future work includes Galoising the sequential mesh partitioner, which currently takes longer to run than the parallel refinement code.

Acknowledgments

This work is supported in part by NSF grants 0833162, 0719966, 0702353, 0615240, 0541193, 0509324, 0509307, 0426787 and 0406380, as well as grants from IBM and Intel Corporation. Milind Kulkarni is supported by a DOE HPCS Fellowship.

References

1. Andrey Chernikov and Nikos Chrisochoides. "Parallel 2D Constrained Delaunay Mesh Generation." *ACM Transactions on Mathematical Software*, Vol. 34(1). 2008.
2. Andrey Chernikov and Nikos Chrisochoides. "Three-dimensional Delaunay Refinement for Multi-core Processors." Proceedings of the *22nd International Conference on Supercomputing*, 214-224. 2008.
3. L. Paul Chew. "Guaranteed-quality Mesh Generation for Curved Surfaces." Proceedings of the *Ninth Annual Symposium on Computational Geometry*. 1993.
4. Rakesh Ghiya and Laurie J. Hendren. "Putting pointer analysis to work." Proceedings of the *25th Symposium on Principles of Programming Languages*, 121-133. 1998.
5. Laurie J. Hendren and Alexandru Nicolau. "Parallelizing Programs with Recursive Data Structures." *IEEE Transactions on Parallel and Distributed Systems,* Vol. 1(1), 35-47. 1990.
6. Randy J. Allen and Ken Kennedy. "Optimizing Compilers for Modern Architectures: a Dependence-based Approach." *Morgan Kaufmann Publishers Inc*. 2002.
7. Venkata Krishnan and Josep Torrellas. "A Chip-multiprocessor Architecture with Speculative Multithreading." *IEEE Transactions on Computers*, Vol 48(9). 1999.
8. 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." Proceedings of the *Symposium on Parallelism in Algorithms and Architectures*, 217-228. 2008.
9. Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. "Optimistic Parallelism Benefits from Data Partitioning." Proceedings of the *International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 36(1), 233-243. 2008.
10. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. "Optimistic Parallelism Requires Abstractions." Proceedings of the *Conference on Programming Language Design and Implementation*, Vol. 42(6), 211-222. 2007.
11. James R. Larus and Paul N. Hilfinger. "Detecting Conflicts between Structure Accesses." Proceedings of the *Conference on Programming Language Design and Implementation*. 1988.
12. James Larus and Ravi Rajwar. "Transactional Memory (Synthesis Lectures on Computer Architecture)." *Morgan & Claypool Publishers*. 2007.
13. Yang Ni, Vijay S. Menon, Ali Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Brantin Saha, and Tatiana Shpeisman. "Open Nesting in Software Transactional Memory." Proceedings of the *12th Symposium on Principles and Practice of Parallel Programming*, 68-78. 2007.
14. Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. "Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse." Proceedings of the *Conference on Supercomputing*, 361-370. 1993.
15. Lawrence Rauchwerger and David Padua. "The LRPD Test: Speculative Runtime Parallelization of Loops with Privatization and Reduction Parallelization." *IEEE Transactions on Parallel Distributed Systems*, Vol. 10(2), 160-180. 1999.
16. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-valued Logic." Proceedings of the *26th Symposium on Principles of Programming Languages*, 105-118. 1999.