Lonestar: A Suite of Parallel Irregular Programs *

Milind Kulkarni[a], Martin Burtscher[a], Călin Caşcaval[b], and Keshav Pingali[a]
[a]The University of Texas at Austin
[b]IBM T.J. Watson Research Center

Abstract

Until recently, parallel programming has largely focused on the exploitation of data-parallelism in dense matrix programs. However, many important application domains, including meshing, clustering, simulation, and machine learning, have very different algorithmic foundations: they require building, computing with, and modifying large sparse graphs. In the parallel programming literature, these types of applications are usually classified as *irregular applications*, and relatively little attention has been paid to them. To study and understand the patterns of parallelism and locality in sparse graph computations better, we are in the process of building the Lonestar benchmark suite. In this paper, we characterize the first five programs from this suite, which target domains like data mining, survey propagation, and design automation. We show that even such irregular applications often expose large amounts of parallelism in the form of *amorphous data-parallelism*. Our speedup numbers demonstrate that this new type of parallelism can successfully be exploited on modern multi-core machines.

1 Introduction

With the increasing importance of parallel programming, there is a need to broaden the scope of parallelization research. While significant research effort has been expended over the past few decades investigating parallelism in domains such as dense linear algebra and stencil codes, less effort has been spent in finding and exploiting parallelism in irregular algorithms, *i.e.*, those that manipulate pointer-based data structures such as trees and lists.

Many irregular programs in important application domains, such as data mining, machine learning, computational geometry and SAT solving, implement iterative, worklist-based algorithms that manipulate large, sparse graphs. Recent case studies by the Galois project have shown that many such programs have a generalized form of data-parallelism called *amorphous data-parallelism* [13]. To understand this pattern of parallelism, it helps to consider Figure 1, which shows an abstract representation of an irregular algorithm. Typically, these algorithms are organized around a graph that has some number of nodes and edges; in some applications, the edges are undirected while in others they are directed. At each point during the execution of an irregular algorithm, there are certain nodes or edges in the graph where computation might be performed. Performing a computation may require reading or writing other nodes and edges in the graph. The node or edge on which a computation is centered is called an *active element*. To simplify the discussion, we assume henceforth that active elements are nodes. Borrowing terminology from the cellular automata literature, we refer to the set of nodes and edges that are read or written in performing the computation at an active node as the *neighborhood* of that active node. Figure 1 shows an undirected graph in which the filled nodes represent active nodes, and shaded regions represent the neighborhoods of those active nodes. In general, the neighborhood of an active node is distinct from the set of its neighbors in the graph. In some algorithms, such as Delaunay mesh refinement [8] and the preflow-push algorithm for maxflow computation [10], there is no *a priori* ordering on the active nodes, and a sequential implementation is free to choose any active node at each step of the computation. In other algorithms, such as event-driven simulation [17] and agglomerative clustering [19], the algorithm imposes an order on active elements that must be respected by the sequential implementation. Both kinds of algorithms can be written using worklists to keep track of active nodes.

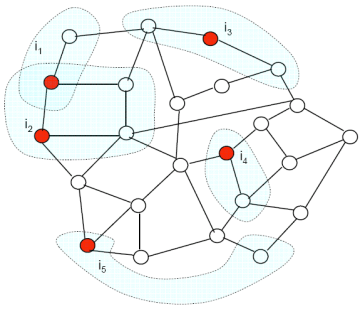


Figure 1: Active elements and neighborhoods

Let us illustrate these notions on the example of Delaunay Mesh Refinement [8]. The input to this algorithm is a triangulation of a set of points in a plane (Figure 2(a)), with some triangles designated as "bad" according to some quality criterion (colored in black in Figure 2(a)). The bad triangles are placed on a worklist, and, for each bad triangle, the algorithm collects a number of triangles around the bad triangle, called a *cavity* (col-

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

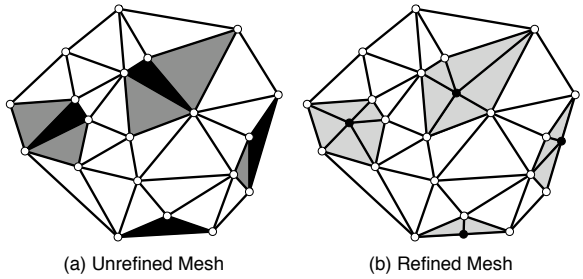


Figure 2: Delaunay Mesh Refinement

```
1: Mesh m = /* read input mesh */
2: Worklist wl = new Worklist(m.getBad());
3: foreach Triangle t in wl {
4:   Cavity c = new Cavity(t);
5:   c.expand();
6:   c.retriangulate();
7:   m.updateMesh(c);
8:   wl.add(c.getBad());
9: }
```

Figure 3: Pseudocode for Delaunay Mesh Refinement

ored in grey in Figure 2(a)), removes the cavity from the graph, and retriangulates the region (colored in grey in Figure 2(b)). If the retriangulation creates new bad triangles, these are placed on the worklist and processed in turn. To relate this algorithm to Figure 1, we note that the mesh is usually represented by a graph in which nodes represent triangles and edges represent triangle adjacencies. At any stage in the computation, the active nodes are the nodes representing badly shaped triangles and the neighborhoods are the cavities of these triangles. In this problem, the active nodes are not ordered.

It should be apparent that this algorithm can be parallelized by processing bad triangles in parallel as long as their cavities do not overlap. In general, the parallelism in an algorithm with amorphous data-parallelism is exploited by processing multiple elements from the worklist concurrently. Because the dependences between elements on the worklist are very complex (*e.g.*, whether or not two cavities overlap is dependent on the size and shape of the mesh), it appears as though there may be little parallelism in these applications. However, we have found that these applications do, indeed, exhibit significant parallelism [12], which we expand on in Section 4.1.

Kulkarni *et al.* [13] propose set iterators as an abstraction to express the parallelism in amorphous data-parallel programs. Two types of iterators are used, un-ordered iterators that allow loop iterations to be executed in any order (as in Delaunay Mesh Refinement), and ordered iterators that impose a partial order on the loop iterations. These iterators take the form of `foreach` statements, where additional work can be added to the worklist being iterated over. Delaunay Mesh Refinement can be easily expressed using these constructs, as seen in Figure 3. The execution model is straightforward: an application executes sequentially until encountering a set iterator, at which point multiple threads begin to execute iterations from the worklist in parallel. A runtime system manages the execution of the threads to ensure that sequential semantics are preserved.

Given this execution model, we implemented five amorphous data-parallel algorithms that span data mining, meshing, simulation, and SAT solving in the Lonestar benchmark suite. We characterize these applications with respect to:

- the amount of available parallelism in the algorithm; we use two metrics, the parallelism profile and the parallelism intensity, that provide insight into the dynamic availability of parallel work in these algorithms for different inputs;
- parallel execution (scaling) on three hardware platforms, demonstrating that the parallelism can, indeed, be exploited; and
- program characteristics, such as cache behavior, memory accesses, and working set sizes to provide an overview of how the algorithmic implementation maps to architectural features.

This paper makes the following contributions.

- It introduces the Lonestar benchmark suite, whose focus is on pointer-based (irregular) data structures and amorphous data-parallelism. The suite covers important emerging and well-known applications, all of which process large datasets and are long running, making it important to parallelize them.
- It reveals that these algorithms have a lot of parallelism, and that this parallelism grows with the input size.
- It demonstrate that this algorithmic parallelism can, indeed, be exploited in parallelized implementations thereof.
- It presents important workload characteristics of these programs. One key finding is that the transactions can be quite large, making them less suitable for hardware-based approaches.

The C++ and Java source code of the benchmark programs, sample inputs, and some documentation are available on-line at http://iss.ices.utexas.edu/lonestar/.

The rest of this paper is organized as follows. Section 2 explains the key algorithms, the data structures, and the parallelism strategy in each of the five benchmarks. Section 3 presents our evaluation methodology. Section 4 shows the workloads' theoretical and delivered parallelism as well as other characteristics. Section 5 discusses related work and Section 6 summarizes the paper.

2 Applications

The following is a detailed description of our benchmarks, their data structures, algorithms and parallel decomposition.

2.1 Agglomerative Clustering

This benchmark is an implementation of a well-known data-mining algorithm, *Agglomerative Clustering* [19], as used in

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

Figure 4: Psuedocode for Agglomerative Clustering

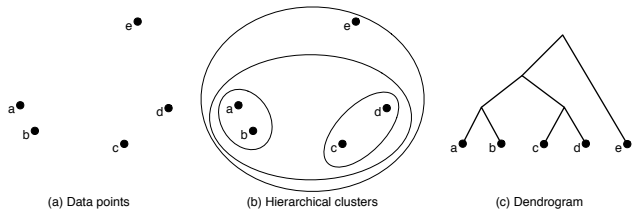


Figure 5: Example of Agglomerative Clustering

the graphics application Lightcuts [21]. The input to the clustering algorithm is (1) a data-set consisting of points in an N-dimensional space and (2) a measure of the similarity between items in the data-set. We refer to this measure of similarity as a *distance metric*; the more dissimilar items are, the farther apart they are according to the distance metric. The output of the algorithm is a binary tree (called a *dendrogram*) representing a hierarchical, pair-wise clustering of the items in the data set. Figure 5(a) shows a data-set containing points in the plane, with a distance metric corresponding to Euclidean distance. The clustering for the data-set is in Figure 5(b), and the resulting dendrogram is in Figure 5(c).

We implement an Agglomerative Clustering algorithm first described by Walter *et al.* [20], which is based on the following observation: if at any time two points agree that they are one another's nearest neighbor, they will be clustered together in the final dendrogram[1]. Pseudocode for the algorithm is in Figure 4. Initially, all data points are placed onto a worklist (line 1). We then build a *kd-tree*, an acceleration structure that allows the quick determination of a point's nearest neighbor (line 2). The algorithm proceeds as follows. For each point p in the worklist, find its nearest neighbor q (line 5) and determine if q's nearest neighbor is p (lines 6-8). If so, cluster p and q and insert a new point into the worklist representing the cluster (lines 9-13). Otherwise, place p back onto the worklist (line 15). The algorithm terminates when there is only one point left in the worklist.

There are two key data structures in Agglomerative Clustering:

- **Unordered Set**: The points left to be clustered can be processed in any order, so the worklist holding those points is represented as an unordered set.
- **KD-Tree**: The kd-tree represents a hierarchical decomposition of the point space in a manner similar to an octree. It is built prior to performing clustering as a means of accelerating nearest-neighbor queries. Rather than requiring $O(n)$ time to find a point's nearest neighbor, the kd-tree allows the search to occur in $O(\log n)$ time. It is kept up-to-date throughout execution by removing clustered points and adding newly created clusters. The kd-tree interface is essentially that of a set (supporting addition and removal) augmented with a nearest-neighbor method.

Parallelism The active nodes in agglomerative clustering are the points to be clustered; they can be processed in any order. Intuitively, two points can be processed simultaneously if their clustering decisions are independent. This means that the following conditions must hold: (i) the formed clusters must involve different points (otherwise, both iterations would attempt to remove the same point from the kd-tree) and (ii) the newly added points must not interfere with the nearest neighbor computations performed by other iterations.

2.2 Barnes-Hut

This benchmark simulates the gravitational forces acting on a galactic cluster using the Barnes-Hut n-body algorithm [3]. The positions and velocities of the n galaxies are initialized according to the empirical Plummer model. The program calculates the motion of each galaxy through space for a number of time steps.

The Barnes-Hut force-calculation algorithm employs a hierarchical data Structure, called an octree, to approximately compute the force that the n bodies in the system induce upon each other. With n bodies, $O(n^2)$ interactions need to be considered, *i.e.*, the precise calculation is quadratic in the number of bodies. The Barnes-Hut algorithm hierarchically partitions the volume around the n bodies into successively smaller cells. Each cell forms an internal node of the octree and summarizes information about the bodies it contains, in particular their combined mass and center of gravity. The leaves of the octree are the individual bodies. This hierarchy reduces the time to calculate the force on the n bodies to $O(nlogn)$ because, for cells that are sufficiently far away, it suffices to perform only one force calculation with the cell instead of performing one calculation with every body inside the cell.

Step by step, the algorithm works as follows (pseudocode is in Figure 6). First, the list of bodies is initialized with the starting location and velocity of each body (line 1). Then the code iterates over the time steps (line 2). In each iteration, a new octree (*i.e.*, spatial hierarchy) is generated by inserting all bodies (lines 3 - 6). Then the cumulative mass and center of mass of each cell is recursively computed (line 7). Next, the force acting on each body is computed (lines 8 - 10) by traversing the octree. The traversal along any path is terminated as soon as a leaf node (*i.e.*,

[1]Subject to certain conditions on the distance metric

```
 1: List bodylist = ...
 2: foreach timestep do {
 3:   Octree octree;
 4:   foreach Body b in bodylist {
 5:     octree.Insert(b);
 6:   }
 7:   octree.SummarizeSubtrees();
 8:   foreach Body b in bodylist {
 9:     b.ComputeForce(octree);
10:   }
11:   foreach Body b in bodylist {
12:     b.Advance();
13:   }
14: }
```

Figure 6: Pseudocode for Barnes-Hut

a body) or an internal node (i.e., a cell) that is far enough away is encountered. Finally, each body's position and velocity are updated based on the computed force (lines 11 - 13).

There are two key data structures in Barnes-Hut:

- **Unordered list**: The bodies are stored in an unordered list, as they can be processed in any order.
- **Octree**: The spatial hierarchy is represented by an octree (the 3-dimensional equivalent of a binary tree), where each node has up to eight children. The leaves of the octree correspond to individual bodies whereas the internal nodes represent cells that contain multiple spatially nearby bodies. This data structure supports the insertion of bodies and recursive traversal.

Parallelism While many phases of Barnes-Hut can be parallelized (including building the octree and calculating the summary information), we focus on parallelizing the force-computation step, which consumes the vast majority of the runtime. The active nodes in this step are the bodies. Calculating the force on each body requires reading some portion of the octree, so the accessed nodes and edges form the body's neighborhood. However, because these accesses are read-only, the bodies can be processed in any order, and in parallel, provided that a body does not update its position and velocity until all force computations are complete.

2.3 Delaunay Mesh Refinement

This benchmark is an implementation of the algorithm described by Kulkarni *et al.* [13]. It is the algorithm discussed in Section 1. The application produces a guaranteed quality Delaunay mesh, which is a Delaunay triangulation with the additional quality constraint that no angle in the mesh be less than 30 degrees. The benchmark takes as input an unrefined Delaunay triangulation and produces a new mesh that satisfies the quality constraint.

The algorithm is initialized with a worklist of all the triangles in the input mesh that do not meet the quality constraints, called "bad" triangles. In each step, the refinement procedure (i) picks a bad triangle from the worklist, (ii) collects the affected triangles in the neighborhood of that bad triangle (called the *cavity*, shown in grey in Figure 2(a)), and (iii) re-triangulates the cavity (creating the new grey triangles in Figure 2(b)). If this re-triangulation creates new badly-shaped triangles in the cavity, these are processed as well until all bad triangles have been eliminated from the mesh. The order in which the bad triangles are processed is irrelevant—all orders lead to a valid refined mesh.

In more detail, the algorithm proceeds as follows (pseudocode is provided in Figure 3). After reading in the input mesh (line 1), a worklist is initialized with the bad triangles in the mesh (line 2). For each bad triangle, a cavity is created (line 4) and expanded to encompass the neighboring triangles (line 5). The algorithm then determines the new triangles that should be created (line 6) and updates the original mesh by removing the old triangles and adding the new triangles (line 7). Recall that the order of processing is irrelevant in this algorithm, so the *foreach* in line 3 iterates over an unordered set.

There are two key data structures used in Delaunay Mesh Refinement:

- **Unordered Set**: The worklist used to hold the bad triangles is represented as an unordered set as the triangles can be processed in any order.
- **Graph**: The mesh is represented as a graph. Triangles in the mesh are represented as nodes in the graph, and triangle adjacency is represented by edges between nodes. The data structure supports the addition and removal of nodes and edges, membership tests for nodes and edges, and a method that returns the neighbors of a given node (this is used during cavity expansion).

Parallelism As discussed in Section 1, the active nodes in Delaunay Mesh Refinement are the bad triangles; the algorithm can be parallelized by processing multiple bad triangles simultaneously. Because the neighborhood of a bad triangle is its cavity, this may result in significant parallelism if the triangles are far enough apart so that their cavities do not overlap (as in Figure 2, where all of the bad triangles can be processed in parallel).

2.4 Delaunay Triangulation

This benchmark produces a Delaunay Triangulation given a set of points. It implements the algorithm proposed by Guibas *et al.* [11]. The algorithm produces a 2D Delaunay mesh given a set of points in a plane. It can be used to generate inputs for Delaunay mesh refinement.

The algorithm proceeds as follows. A worklist is initialized with the points to be inserted, and the mesh is initialized with a single, large triangle encompassing all the points. In each step, the triangulation procedure (i) picks a new point from the worklist; (ii) determines which triangle contains the point; (iii) splits this triangle into three new triangles that share the point; and (iv) re-triangulates the neighborhood. The order in which the points are processed is irrelevant—all orders lead to the same valid Delaunay mesh.

In more detail, the algorithm proceeds as follows (pseudocode is provided in Figure 7). After initializing the mesh with one sur-

```
 1: Mesh m = /* initialize with one
            surrounding triangle */
 2: Set<Point> points = /* read
                points to insert */
 3: Worklist wl;
 4: wl.add(points);
 5: foreach 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 7: Pseudocode for Delaunay Triangulation

rounding triangle (line 1), a worklist is initialized with the set of input points (lines 2-4). For each point p in the worklist (line 5), the triangle t that contains p is retrieved (line 6). Then t is split into three new triangles such that they share the point p (line 7). Because these new triangles may not satisfy the Delaunay property, a procedure called *edge flipping* is applied to restore the Delaunay property (lines 9-15); If any edge of the newly created triangles is non-Delaunay, 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). Thus, at the end of each iteration of the outer loop, the resulting mesh is once again a Delaunay mesh. Recall that the order of processing is irrelevant in this algorithm, so the *foreach* in line 5 iterates over an unordered set.

There are three key data structures used in Delaunay Triangulation:

- **Unordered Set**: The worklist used to hold the 2D points is represented as an unordered set, as the points can be inserted in any order.
- **Graph**: The mesh is represented as a graph, as in Delaunay refinement.
- **History DAG**: To efficiently locate the triangle containing a given point, a data structure called *history DAG* is used, which behaves like a ternary search tree. After each step of the algorithm, the leaves represent the triangles in the current mesh. Splitting a triangle (line 9) adds three children to the data structure corresponding to the three newly created triangles. 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. With this structure finding the triangle containing a point is equivalent to traversing the history DAG from the root to the corresponding leaf. The data structure supports the addition and removal of nodes and membership tests for nodes.

Parallelism The active nodes in Delaunay Triangulation are the points to be inserted into the mesh, which can be processed in any order. Processing a point requires splitting a triangle and then flipping some set of edges; the affected triangles are in the point's neighborhood. Because these neighborhoods are typically small, connected regions of the mesh, significant parallelism can be achieved by inserting multiple points in parallel, provided the points affect triangles that are far apart in the mesh.

```
1:  FactorGraph f = /* read initial formula */
2:  wl.put(f.clausesAndVariables());
3:  foreach Node n in wl {
4:  if (/*time out or number of variables is small*/) {
5:    break;
6:  }
7:  if (n.isVariable()) {
8:    n.updateVariable();
9:    if (/* n is frozen */) {
10:     /* remove n from graph */
11:           continue;
12:         } else {
13:           n.updateClause();
14:         }
15:    wl.add(n);
16:  }
```

Figure 8: Pseudocode for Survey Propagation

2.5 Survey Propagation

Survey Propagation is a heuristic SAT-solver based on Bayesian inference [6]. The algorithm represents the Boolean formula as a *factor graph*, a bipartite graph with variables on one side and clauses on the other. An edge connects a variable to a clause if the variable participates in the clause. The edge is given a value of -1 if the literal in the clause is negated, and +1 otherwise. The general strategy of SP is to iteratively update each variable with the likelihood that it should be assigned a truth value of **true** or **false**.

Pseudocode is given in Figure 8. The worklist for Survey Propagation consists of all nodes (both variables and clauses) in the graph. At each step, Survey Propagation chooses a node at random and processes it. To process a node, the algorithm updates the value of the node based on the values of its neighbors. After a number of updates, the value for a variable may become "frozen" (*i.e.*, set to **true** or **false**). At that point, the variable is removed from the graph. If a node is not frozen, it is returned to the worklist to be processed again. As the algorithm progresses and variables become frozen, the graph begins to shrink. Note that although the algorithm chooses variables to update at random, the algorithm is nonetheless highly order dependent: different orders of processing will lead to variables becoming frozen at different times.

The termination condition for Survey Propagation is fairly complex: when the number of variables is small enough, the Survey Propagation iterations are terminated, and the remaining problem is solved using a local heuristic such as WalkSAT. Alternatively, if there is no progress after some number of iterations, the algorithm may just give up.

There are two key data structures in Survey Propagation:

- **Unordered Set**: Because the algorithm is based on iteratively updating the values of variables chosen at random, the worklist can be represented as an unordered set. There are no ordering constraints on the processing the elements of

the worklist (although, as discussed above, different orders of processing may lead to different algorithmic behavior).
- **Factor Graph**: The bipartite graph representing the boolean formula.

Parallelism The active nodes in Survey Propagation are the clauses and variables of the factor graph; in other words, every node in the graph is an active node, and they can be processed in any order. Because processing a node requires reading its neighbors, two nodes can be processed in parallel as long as they are not neighbors. If a variable node needs to be removed from the graph because it is frozen, this restriction becomes a little tighter. Because removing a variable from the factor graph requires updating the edge information at the neighboring clause nodes, an iteration that removes a variable cannot occur concurrently with an iteration that reads *or* writes one of the neighboring clauses.

3 Evaluation Methodology

First, we studied each algorithm to determine how much *potential* for parallelism it has. To do this, we used a profiling tool called ParaMeter [12], which estimates the amount of *available parallelism* in an algorithm with amorphous data-parallelism. Details on ParaMeter and the results from profiling our applications are presented in Section 4.1.

To determine whether amorphous data-parallelism can be exploited to speed up program execution, we used the Galois system [13] to produce a parallel version of each application. We measured the performance with various numbers of threads on the following three systems.

- The **UltraSPARC IV** system is a Sun E25K server running SunOS 5.9. It 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.
- The **Xeon X7350** system is a SuperMicro SuperServer-8045C-3R running Linux 2.6.22. It contains four CPU modules with four-core 2.93 GHz Intel Xeon X7350 x86_64 processors. The 16 CPUs share 48 GB of main memory. Each core has a 32 kB L1 data cache and shares a unified 4 MB L2 cache with one other core.
- The **UltraSPARC T1** system is a Sun-Fire-T200 server running SunOS 5.10. It contains an eight-core 1.2 GHz UltraSPARC T1 "Niagara" processor. The cores are four-way multithreaded. The 32 virtual CPUs share 16 GB of main memory. Each core has an 8 kB four-way set-associative L1 data cache and all cores share a unified 3 MB L2 cache.

We compiled the parallel code with Sun's Java compiler version 1.6.0 and ran it on the HotSpot 64-bit server virtual machine on each platform. 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 fastest run. We use a 400 GB heap on the UltraSPARC IV, a 40 GB heap on the Xeon X7350, and a 15 GB heap on the UltraSPARC T1. To minimize the interference by the garbage collector, we force a full-heap garbage collection before executing the measured code section.

All measurements, other than memory footprints, are obtained through source-code instrumentation; that is, we read the timer (and the CPU performance counters where applicable) 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 on the UltraSPARC IV system. Note that the performance counters only capture events in user mode.

To approximate the memory footprints of our applications, we determined the minimum heap size for which the program would complete when run using the HotSpot 64-bit virtual machine. Because each program uses its maximum amount of memory during the parallelized section, this approach suffices to find the memory footprint of the parallel code.

We ran each benchmark with three random inputs. Note that we only measure the core of each application and omit, for example, initialization (reading in or generating the input) and finalization code. In all cases, the omitted code represents no more than a few percent of the total sequential runtime.

- **Agglomerative Clustering** The small input contains 500,000, the middle input 1,000,000, and the large input 2,000,000 numbers. We measure the clustering code but exclude building the kd-tree.
- **Barnes-Hut** The small input contains 65,000, the middle input 110,000, and the large input 220,000 bodies. The bodies' positions and velocities are initialized according to the empirical Plummer model. We only measure the force calculation code. Building the octree is excluded from our measurements.
- **Delaunay Mesh Refinement** 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, and the large input consists of 549,998 triangles of which 261,100 are initially bad. We only measure the refinement algorithm. Building the initial graph and partitioning it are excluded from our measurements.
- **Delaunay Triangulation** The small input contains 20,000, the middle input 40,000, and the large input 80,000 points.
- **Survey Propagation** The small input is a Boolean formula in conjunctive normal form with three literals per clause and a total of 250 variables and 1050 clauses, the medium input contains 350 variables and 1470 clauses, and the large input contains 500 variables and 2100 clauses.

4 Workload Characterization

4.1 Available Parallelism

A commonality among several of the worklist algorithms we have described is that, while the potential for parallelism exists, the pattern of dependences between individual iterations of the

algorithm are quite complex. For example, in Delaunay Triangulation, it is apparent that it should be possible to insert multiple points into the mesh simultaneously, provided they affect different regions of the mesh. However, determining whether two points are, indeed, independent is non-trivial and dependent on the current state of the computation as well as the two points in question.

Due to the nature of these complex dependencies, it is not always clear that there is, in fact, any parallelism to be exploited in irregular applications such as the ones we have presented. To determine how much parallelism actually exists in our applications, we applied a profiling tool called ParaMeter [12] to our benchmark applications.

The goal of ParaMeter is to estimate an upper bound on the amount of exploitable parallelism in algorithms exhibiting amorphous data-parallelism. It does so by simulating execution of the application on a system with an infinite number of processors. The tool divides the execution of the algorithm into a series of discrete computation steps. In each step, the worklist is examined and a maximally independent set of iterations is executed. That is, in each step, a set of iterations that can safely be executed in parallel is chosen and executed. Iterations are deemed independent if they do not conflict *algorithmically*, regardless of any dependences due to specific data structure implementations or conflicts due to parallelization run-time systems such as the Galois system [13] or Transactional Memory [14]. In other words, the parallelism found by ParaMeter is an intrinsic property of the algorithm.

ParaMeter generates *parallelism profiles* and *parallelism intensity* plots. A parallelism profile shows the amount of parallel work available in each computation step of an algorithm; it makes clear how the amount of available parallelism changes throughout the execution of an application. Parallelism profiles show the absolute amount of parallelism available at any point in time. However, while low available parallelism may be a result of too many conflicts between iterations, it may also be a product of too little work to do. Parallelism intensity addresses this by expressing the amount of parallel work as a percentage of the total amount of work available to be executed. Thus, a low parallel intensity means that most of the work in the worklist cannot safely be executed in parallel, while a high intensity means that most of the work in the worklist is independent.

We applied ParaMeter to our suite of applications, examining both the available parallelism and the parallelism intensity across the three input sizes for each application. The results of these profiling runs can be seen in Figures 9(a) through 9(d).

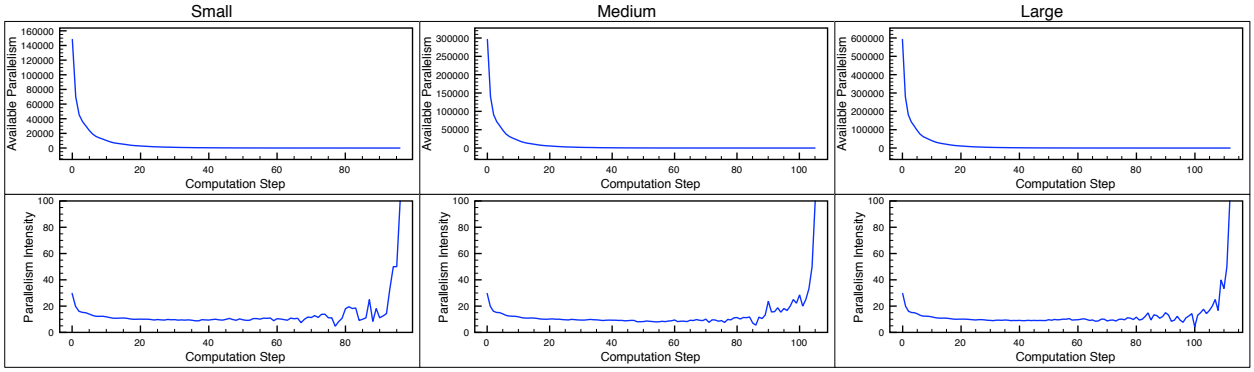
Agglomerative Clustering Agglomerative Clustering is, at its heart, a bottom-up tree-building algorithm. We note that the tree can be built level-by-level in parallel: all the leaves can be clustered simultaneously, then the second-to-last level, and so forth. The amount of parallelism is thus proportional to the number of nodes at each level of the tree. Intuitively, building a bushy tree will allow more parallelism than building a skinny tree.

Because the points in our Agglomerative Clustering inputs are randomly generated, we expect relatively bushy trees, and hence a pattern of parallelism that begins high and rapidly decreases as computation progresses. This is precisely the behavior we see in the parallelism profiles of Figure 9(a). Further, as the computation progresses, we see that the parallelism intensity (shown in Figure 9(b)) decreases as well, until the end of the computation, when there is little work in the worklist. If we were building a complete, balanced binary tree, we would expect the intensity to remain constant: at each step, all of the points in the worklist could be clustered. This would result in an intensity of 50%, as for each pair of points, only one could complete successfully. Our results are lower than that as the inputs do not lead to such a tree. Finally, we note that, as we increase the input size, the amount of parallelism scales roughly linearly.

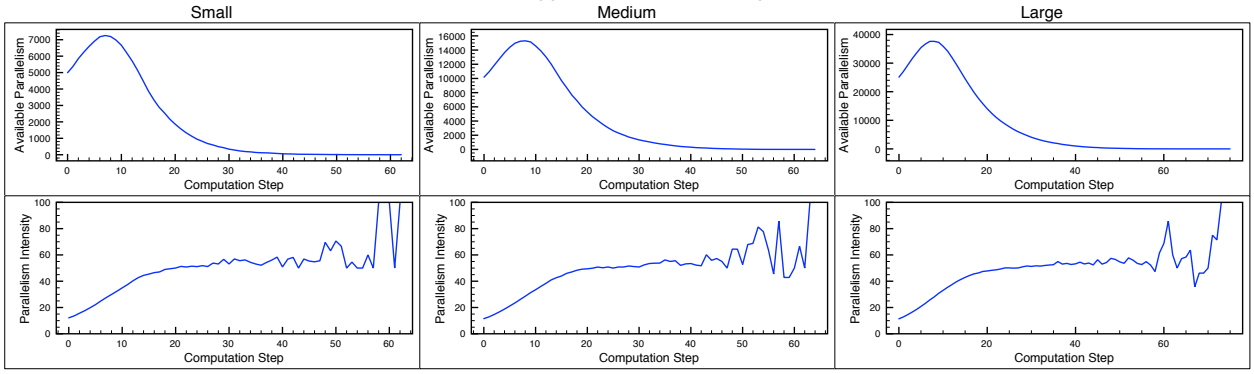
Barnes-Hut The parallelized portion of Barnes-Hut, the force calculation code, is embarrassingly parallel; there are no conflicts between any of the iterations in the worklist. As a result, the parallelism profile and parallelism intensity plot for the application are uninteresting. All the iterations can be safely executed in parallel in a single computation step, and the parallelism intensity is 100% for that step. This is true for all input sizes.

Delaunay Mesh Refinement In the abstract, Delaunay Mesh Refinement is a graph refinement code. The current mesh is represented as a graph, with nodes of the graph representing triangles in the mesh and edges between nodes representing triangle adjacency. When processing a bad triangle, the cavity formed is a small, connected set of triangles, which is a subgraph of the overall graph. The retriangulated cavity is another subgraph, containing more nodes than the subgraph it is replacing, and hence the graph becomes larger. As the graph becomes bigger, the likelihood of two cavities overlapping decreases, so we would expect the available parallelism and parallelism intensity to increase.

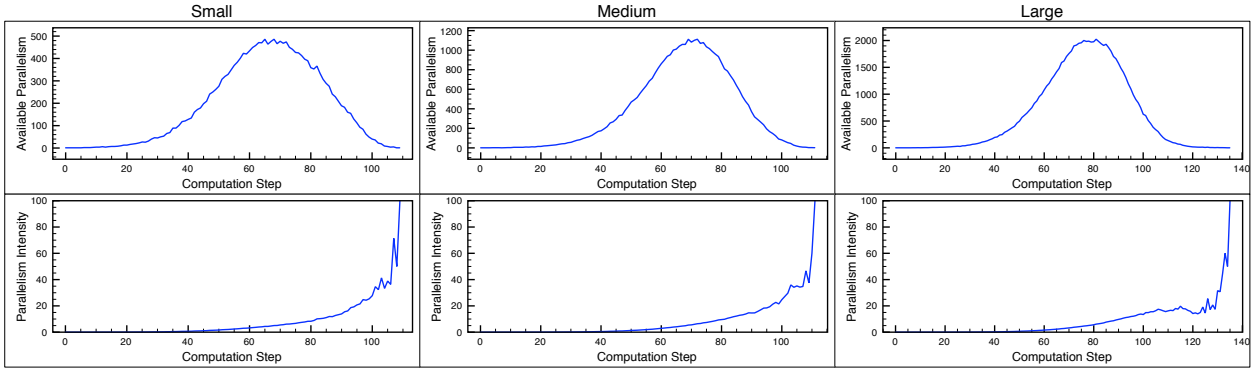
We see that the parallelism profiles for Delaunay Mesh Refinement (the top graphs in Figure 9(b)) behave largely as expected: we start out with a significant amount of parallelism; as the graph becomes larger, the amount of parallelism increases, until it begins to drop as the algorithm runs out of work to do. The parallelism intensity plots (the bottom graphs in Figure 9(b)) show the expected increase in intensity. However, even though the graph continues to get bigger throughout execution, by the middle portion of the computation, the intensity stops increasing. This runs contrary to our expectation of increasing parallelism as the graph grows. This is because parallelism intensity increases as the graph grows larger *only if the work is uniformly distributed through the graph*. Initially, bad triangles are uniformly distributed through the mesh, so the intensity increases as the mesh becomes larger and more computations become independent. Eventually, however, the majority of the work in the worklist is from bad triangles created by retriangulation—in other words, newly created work. This work, rather than being uniformly distributed, is created at the site of retriangulations. Suppose retriangulating a cavity produces, on average, two new bad triangles. No matter how large the graph is, those two bad



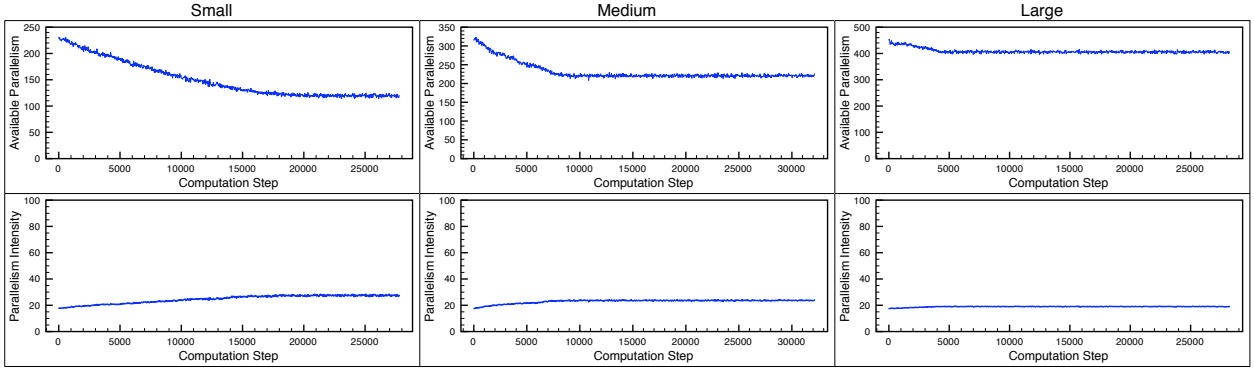
(a) Agglomerative Clustering



(b) Delaunay Mesh Refinement



(c) Delaunay Triangulation



(d) Survey Propagation

Figure 9: Parallelism profiles and parallelism intensity for three input sizes

triangles are necessarily near each other, and are therefore likely to conflict with one another. Thus, in the steady state, where all work is newly created work, we would expect a parallelism intensity of roughly 50%, which happens to be close to what we observe. Finally, we see that as the input size increases, the pattern of parallelism is identical, but the amount of parallelism scales with the input.

Delaunay Triangulation Delaunay Triangulation is also a graph refinement code. Point insertion removes a subgraph from the graph (representing the triangle that will be split, as well as any triangles affected by edge-flipping actions) and replaces it with a larger subgraph (the three triangles formed as the result of the splitting, as well as the triangles created after edge-flipping). We thus expect a behavior similar to that of Delaunay Mesh Refinement. This is borne out by the parallelism profile seen in the top set of graphs in Figure 9(c), which exhibits the same bell shape as the profiles of Delaunay Mesh Refinement. Unlike in mesh refinement, at the beginning of the computation, there is effectively no parallelism. This is because the initial mesh is very small, consisting of a single triangle. The parallelism intensity plots also look similar to those of mesh refinement. However, because the work for triangulation is uniformly distributed, we see that the parallelism intensity increases throughout execution. As in the case of mesh refinement, the amount of parallelism increases as the input size increases.

Survey Propagation Each iteration of Survey Propagation touches a single node in the factor graph, and a small neighborhood around that node; iterations conflict with one another if those neighborhoods overlap. The structure of the graph is largely constant, except for occasionally removing a node. Thus, we would expect the available parallelism to reflect the connectivity of the graph, and remain roughly constant, dropping as nodes are removed from the graph. This is what we observe in the parallelism profiles shown in Figure 9(d). Note that unlike the other applications we have examined, Survey Propagation terminates before the worklist is empty.

Parallelism intensity, interestingly, *increases* slightly as nodes are removed from the graph, as seen in Figure 9(d). This is because removing nodes from the factor graph serves to reduce the connectivity of the graph, making it less likely that two iterations will conflict. Unsurprisingly, the amount of parallelism is correlated with the input size, increasing roughly linearly.

4.2 Exploiting Amorphous Data-Parallelism

To ascertain that the amorphous data-parallelism identified in the previous subsection can, in fact, be exploited, we wrote Galois versions of our applications and measured their performance on three platforms. Figure 10 shows the results. The five rows of panels correspond to the five applications and the three columns correspond to the three platforms. Each panel contains three graphs representing the three inputs. The x-axis in each panel lists the number of threads and the y-axis shows the speedup of the Galoised code over our sequential implementation of the same algorithm. The sequential codes do not include any locking, threading, conflict detection, or undo-information-recording overhead. The absolute runtimes of the sequential programs are presented in the next subsection.

We observe that all five applications achieve at least a 15x speedup on one of their inputs. Survey Propagation scales the least but also has the smallest amount of available parallelism. It is the only application that does not achieve a speedup over sequential with two threads. The remaining four applications scale well and achieve over 50% parallel efficiency up to 16 threads on the UltraSPARC IV system.

Almost all program and input combinations result in worse performance with 128 threads than with 64 threads. We performed a detailed investigation of the causes for this behavior on Delaunay Refinement [7] and identified two main culprits. The first is inter-board communication, which is high for large numbers of threads. Recall that this machine comprises sixteen boards with eight processors each. Hence, threads that execute on different boards have to communicate via much slower channels than threads that are collocated on the same CPU board. The second culprit is load imbalance. Above eight threads, the imbalance starts to become significant, and for 128 threads on average over half of the time the threads are idling with all three inputs. The load imbalance grows with the number of threads because more threads result in less work per thread, increasing the likelihood of imbalance problems. Clearly, the Galois system should be augmented with a load balancer and should attempt to allocate worker threads that communicate to nearby CPUs.

In most instances, especially Barnes-Hut and Delaunay Triangulation but also Agglomerative Clustering and Survey Propagation, larger inputs result in higher speedups. This observation is in line with the results from the previous subsection and probably means that even larger inputs would result in even higher parallel speedups. Only Delaunay Refinement does not follow this trend. We surmise that this is because better load balancing and increased parallelism are the "low-hanging fruit" that allow larger problem sizes to exhibit improved scalability. The former factor aids Barnes-Hut, while the latter factor aids Delaunay Triangulation. In contrast, Delaunay Refinement already has significant parallelism with the small input, and load balancing may not improve as the input size increases.

Comparing the speedup with 16 threads on the different systems, we find that Agglomerative Clustering scales much better on the UltraSPARC IV than on the other platforms. Barnes Hut scales similarly well on each platform. It is trivially parallelizable and reaches a speedup of about 14x with 16 threads. Delaunay Refinement does much better on the two UltraSPARC machines than on the Xeon system. Similarly, Delaunay Triangulation scales best on the UltraSPARC T1 and worst on the Xeon. We believe the reason for this behavior is the large number of stores and data-cache write misses of the two Delaunay codes (see next section), which cause a lot of coherence traffic that affects the Xeon system more than the other two systems. Nevertheless, in absolute terms, the Xeon system is typically the

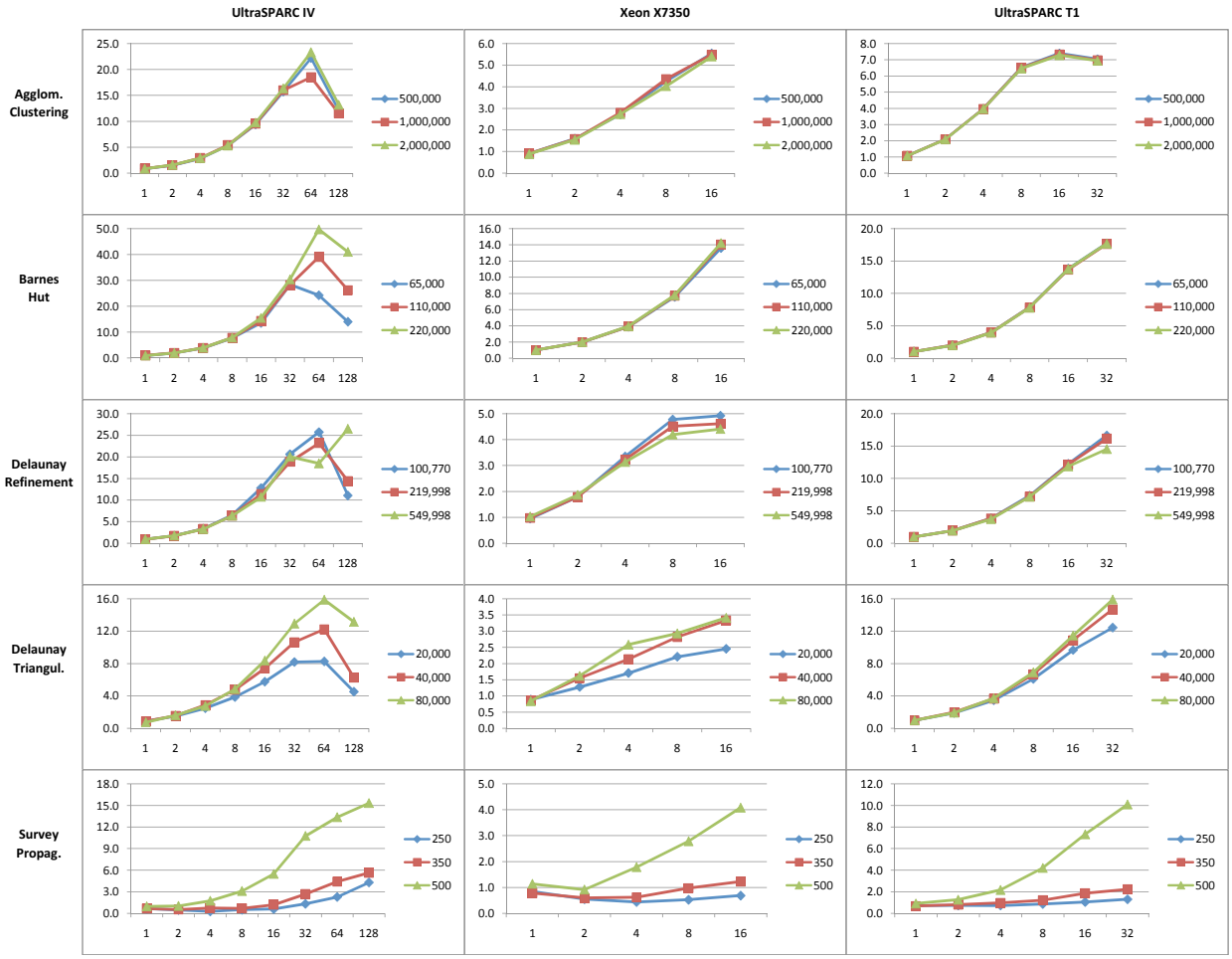


Figure 10: Speedup over sequential code (y-axis) of the 5 benchmarks on the 3 architectures for different numbers of threads (x-axis)

	iterations	sequential runtime [s] on			min. heap	CPI	instructions	mem. acc.	loads per	stores per	L1d miss	load miss	store miss	% of instructions that are		
		SPARC IV	Xeon	SPARC T1	size [MB]		per iteration	per iteration	iteration	iteration	rate (%)	rate (%)	rate (%)	mem. acc.	loads	stores
Agglom. Clustering	499,999	54.6	8.3	420.6	261	1.82	63,825.8	13,028.2	11,958.3	1,069.9	7.0	1.5	68.7	20.4	18.7	1.7
	999,999	114.8	17.2	857.2	519	1.81	66,248.2	13,447.2	12,350.5	1,096.7	7.0	1.5	69.2	20.3	18.6	1.7
	1,999,999	235.4	33.9	1,748.1	1,039	1.89	67,920.4	13,831.2	12,708.2	1,123.0	7.1	1.5	69.6	20.4	18.7	1.7
Barnes Hut	65,000	22.5	7.1	2,810.0	13	2.15	167,852.7	41,884.3	34,528.6	7,355.7	13.3	10.3	27.3	25.0	20.6	4.4
	110,000	41.8	12.9	5,157.7	21	2.20	182,194.8	45,370.1	37,376.4	7,993.7	14.4	11.7	27.2	24.9	20.5	4.4
	220,000	92.0	28.3	11,330.6	41	2.19	199,166.7	49,789.0	40,982.9	8,806.1	14.1	11.3	27.1	25.0	20.6	4.4
Delaunay Refinement	206,738	32.0	6.1	231.7	455	2.16	75,281.4	23,464.3	14,408.9	9,055.4	31.8	6.3	72.5	31.2	19.1	12.0
	503,920	76.2	15.4	545.6	1,022	2.18	72,505.0	22,729.4	13,985.0	8,744.4	31.8	6.3	72.5	31.3	19.3	12.1
	1,297,380	195.1	43.1	1,390.8	2,545	2.24	72,747.1	22,684.0	13,958.8	8,725.2	31.7	6.3	72.4	31.2	19.2	12.0
Delaunay Triangul.	20,000	13.2	3.3	63.4	238	2.73	240,953.1	80,663.2	45,689.0	34,974.2	40.0	11.8	76.7	33.5	19.0	14.5
	40,000	29.2	7.2	130.6	483	2.82	267,690.5	87,009.5	48,571.4	38,438.1	41.2	12.2	77.8	32.5	18.1	14.4
	80,000	60.0	15.5	263.9	927	2.99	262,952.4	91,547.1	51,771.8	39,775.4	41.1	12.5	78.3	34.8	19.7	15.1
Survey Propag.	966,204	57.7	13.4	103.8	8	1.21	51,770.6	14,263.3	12,925.7	1,337.6	4.6	2.2	27.8	27.6	25.0	2.6
	1,875,670	139.5	25.6	245.5	8	1.24	62,783.1	10,048.7	8,741.2	1,307.5	7.2	3.7	30.2	16.0	13.9	2.1
	4,492,403	673.4	131.0	1,364.6	8	0.87	177,885.1	42,016.7	40,610.8	1,405.8	2.1	1.1	29.3	23.6	22.8	0.8

Table 1: Information about the five benchmarks for the three inputs

fastest.

The performance anomaly in Delaunay Refinement with 64 threads is due to a very high CPI, which we believe is caused by unfortunate partitioning that results in an unusually large amount of communication.

4.3 Program Characteristics

Table 1 provides information about the sequential versions of our benchmarks as it pertains to the UltraSPARC IV system (unless otherwise noted). For each program, it lists results for the small, middle, and large inputs from top to bottom. From left to right, the table presents information on the number of iterations in the loop that is the parallelization target, the runtimes on the three evaluation platforms, the smallest heap size necessary to run the applications, the average cycles per executed instruction (CPI), the average number of executed instructions, memory accesses (mem. acc.), loads, and stores per iteration, the L1 data-cache (L1d) miss rate, the load and store miss rates, and the fraction of executed instructions that access, read from, and write to memory.

The main observations are that, in all five applications, the average iteration executes several tens of thousands of instructions, including thousands of memory accesses. A substantial fraction of the memory accesses misses in the 64 kB four-way associative cache. These results indicate that L1-cache-based concurrency mechanisms such as TM may not be ideal for these programs.

The instruction mix contains between 20% and 35% memory accesses. Each application executes, on average, close to 20% loads. However, only the two Delaunay codes execute many stores (over 12% compared to under 4.5% for the other applications). For all programs except Barnes-Hut, most of the stores miss in the L1 cache. Part of the reason for the poor L1 d-cache performance on stores is the UltraSPARC IV's no-write-allocate policy. The load miss rates are much lower but still high for Barnes-Hut, Delaunay Triangulation, and, to a lesser degree, Delaunay Refinement. These three programs also suffer from high overall L1 miss rates, especially the two Delaunay codes. This is expected as they jump back and forth between processing different parts of the mesh. Because of the high cache miss rates, these three programs also have the highest CPIs. Nevertheless, the CPIs are not low for the remaining programs, either. In other words, the superscalar CPU is only able to execute one instruction per two to three cycles on average.

The total data size of these programs varies between 8 MB and 2.5 GB. Except for Survey Propagation, the heap sizes exceed common last-level cache sizes. Of the three systems, the Xeon is the fastest (probably because of its much higher clock frequency) and the UltraSPARC T1 the slowest (presumably because of its small caches).

5 Related Work

A number of recent papers have explored the characterization of parallel program behavior for emerging workloads. PARSEC [5] is a suite of multithreaded applications targeted to shared memory multiprocessors. The suite focuses on a set of diverse emerging workloads that spans a number of domains (financial, data mining, data processing, computer vision, *etc.*) and parallelism models (data-parallel, pipeline parallel, and unstructured). The suite contains one unstructured parallel benchmark, *canneal*, that performs a simulated annealing search used in circuit placement. All presented results are simulated using a PIN [15] plugin that provides a cache model. A subset of these benchmarks is ported to the Thread Building Blocks library [9] and characterized on real hardware up to four cores and simulated up to 32. This characterization addresses mainly TBB overheads for dynamic management of parallelism.

The STAMP benchmark suite [16] is a set of codes intended to characterize transactional memory behavior. The codes in the suite are also intended to capture the parallel behavior of new emerging algorithms in domains such as data mining, search and classification. They use a variety of dynamic data structures such as lists, trees and graphs. The published characterization of these benchmarks is mainly focused on simulated transactional memory behavior and does not address the parallelization potential. In contrast, our study is a theoretical and practical characterization of irregular applications, quantifying the available parallelism, the memory behavior, and the scalability on different architectures.

Several other studies have characterized older parallel benchmark suites, including Perfect Club [4], SPLASH-2 [22], NAS Parallel Benchmarks [2], and SPEComp [1]. The codes in these benchmarks are almost exclusively characteristic of the scientific, high-performance computing domain of regular dense linear algebra. Many of these codes are no longer considered representative of the behavior of workloads on current machines. The Olden benchmarks [18] are small, hand-parallelized kernels that operate over irregular data structures. They are intended for program analysis research for finding structural invariants rather than for studying parallelism.

6 Conclusions

We have identified a type of parallelism, called amorphous data-parallelism, that arises naturally in iterative algorithms that operate over sparse-graph data structures. To study this style of parallelism better, we have begun to collect the Lonestar suite of real-world applications that exhibit amorphous data-parallelism, the first five of which are Agglomerative Clustering, Barnes-Hut, Delaunay Triangulation, Delaunay Refinement, and Survey Propagation. These applications span a range of domains, from simulation to data-mining to graphics.

We have studied these algorithms and determined that, despite the complex nature of their execution, there is significant parallelism available to be exploited, and this parallelism scales with the problem size. We have further shown that this potential parallelism can, indeed, be realized by a software parallelization system, achieving high speedups over sequential execution on a range of architectures. Finally, we examined the low-level char-

acteristics of these applications and found that the granularity of the parallelism is quite high, presenting interesting challenges to the development of run-time systems to exploit this parallelism.

Because these applications represent an important class of algorithms that appear in numerous domains and have significant potential for parallelism, we feel they are worthwhile targets for parallelization research.

References

[1] V. Aslot, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *In Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[3] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–559, December 1986.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, and R. Goodrum. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3:5–40, 1989.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27:201–226, 2005.

[7] M. Burtscher, M. Kulkarni, D. Prountzos, and K. Pingali. On the scalability of an automatically parallelized irregular application. In *Languages and Compilers for Parallel Computing (LCPC)*, 2008.

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

[9] G. Contreras and M. Martonosi. Characterizing and improving the performance of the Intel Threading Building Blocks. In *IISWC 2008: IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[11] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1):381–413, December 1992.

[12] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. How much parallelism is there in irregular applications? In *Principles and Practices of Parallel Programming (PPoPP)*, pages 3–14, 2009.

[13] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.

[14] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[16] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC 2008: IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[17] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[18] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, 1995.

[19] P.-N. Tan, M. Steinbach, and V. Kumar, editors. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.

[20] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing (RT)*, 2008.

[21] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. Greenberg. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (SIGGRAPH)*, 24(3):1098–1107, July 2005.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.