# How Much Parallelism is There in Irregular Applications? *

Milind Kulkarni, Martin Burtscher,
Rajasekhar Inkulu, Keshav Pingali

Institute for Computational Engineering and Sciences
The University of Texas at Austin
{milind, burtscher, rinkulu}@ices.utexas.edu,
pingali@cs.utexas.edu

Călin Caşcaval

IBM Research
cascaval@us.ibm.com

## Abstract

Irregular programs are programs organized around pointer-based data structures such as trees and graphs. Recent investigations by the Galois project have shown that many irregular programs have a generalized form of data-parallelism called *amorphous data-parallelism*. However, in many programs, amorphous data-parallelism cannot be uncovered using static techniques, and its exploitation requires runtime strategies such as optimistic parallel execution. This raises a natural question: how much amorphous data-parallelism actually exists in irregular programs?

In this paper, we describe the design and implementation of a tool called ParaMeter that produces *parallelism profiles* for irregular programs. Parallelism profiles are an abstract measure of the amount of amorphous data-parallelism at different points in the execution of an algorithm, independent of implementation-dependent details such as the number of cores, cache sizes, load-balancing, etc. ParaMeter can also generate constrained parallelism profiles for a fixed number of cores. We show parallelism profiles for seven irregular applications, and explain how these profiles provide insight into the behavior of these applications.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

***General Terms*** Algorithms, Languages, Performance

***Keywords*** Optimistic Parallelism, Profiling, Parallelism Profiles

## 1. Introduction

> A theory is something nobody believes, except the person who invented it. An experiment is something everybody believes, except the person who did it.
>
> —*Albert Einstein*

Over the past twenty-five years, the parallel programming community has developed a deep understanding of the opportunities for exploiting parallelism in *regular* programs organized around dense vectors and matrices, such as matrix factorizations and stencil computations. The bulk of the parallelism in these programs is *data-parallelism*, which arises when an operation is performed on disjoint subsets of data elements of vectors and matrices. For example, in multiplying two $N \times N$ matrices using the standard algorithm, there are $N^3$ multiplications that can be executed concurrently as well as $N^2$ independent additions of $N$ numbers each. Data-parallelism in regular programs is independent of the data values in the computations, so it usually manifests itself in FORTRAN-style DO-loops in which iteration independence can be statically determined. Thanks to several decades of research, we now have powerful tools based on integer linear programming for finding and exploiting data-parallelism in regular programs [16].

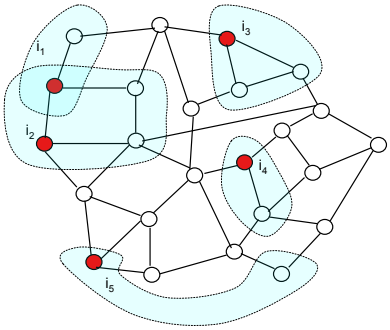



**Figure 1.** Active elements and neighborhoods

In contrast, we understand relatively little about the patterns of parallelism and locality in *irregular* programs, which are organized around pointer-based data structures such as trees and graphs. Recent case studies by the Galois project have shown that many irregular programs have a generalized form of data-parallelism called *amorphous data-parallelism* [20]. To understand this pattern of parallelism, it is useful to consider Figure 1, which is 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 keep the discussion simple, we assume from here on that active elements are nodes. Borrowing terminology from the literature on cellular automata, 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. Note that 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 [5] and preflow-push algorithm for maxflow computation [6], there is no *a priori* ordering on active nodes, and a sequential implementation

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



*PPoPP'09,* February 14–18, 2009, Raleigh, North Carolina, USA.

is free to choose any active node at each step of the computation. In other algorithms, such as event-driven simulation [22] and agglomerative clustering [25], the algorithm imposes an order on active elements, which must be respected by the sequential implementation. Both kinds of algorithms can be written using worklists to keep track of active nodes.

We illustrate these notions using Delaunay mesh refinement [5]. The input to this algorithm is a triangulation of a region in the plane such as the one in Figure 2(a). Some of the triangles in the mesh may be badly shaped according to certain shape criteria (these triangles are colored black in Figure 2). If so, an iterative refinement procedure, outlined in Figure 3, is used to eliminate them from the mesh. In each step, the refinement procedure (i) picks a bad triangle from the worklist, (ii) collects a number of triangles in the neighborhood (called *cavity*) of that bad triangle (lines 6-7 in Figure 4), shown as shaded regions in Figure 2, and (iii) re-triangulates the cavity (lines 8-9). If this re-triangulation creates new badly-shaped triangles, they are added to the worklist (line 10 in Figure 4). The shape of the final mesh depends on the order in which the bad triangles are processed, but it can be shown that every processing order terminates and produces a mesh without badly-shaped triangles. 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, active nodes are not ordered.

From this description, it is clear that bad triangles whose cavities do not overlap can be processed in parallel. However, the parallelism is very complex and cannot be exposed by compile-time program analyses such as points-to or shape analysis [13, 24] because the dependences between computations on different worklist items depend on the input mesh and on the modifications made to the mesh at runtime. To exploit this amorphous data-parallelism, it is necessary in general to use *speculative* or *optimistic* parallel execution [20]: worklist items are processed concurrently by different threads, but to ensure that the semantics of the sequential program are respected, the runtime system detects dependence violations between concurrent computations and rolls back conflicting computations as needed.

This paper addresses the following question: how much parallelism is there in applications that exhibit amorphous data-parallelism? This information is useful for understanding the nature of amorphous data-parallelism, for making algorithmic choices for a given problem, for choosing scheduling policies for processing worklist items, *etc*. Speedup numbers on real machines do not address this question directly since speedups depend on many factors including cache miss ratios, and communication and synchronization overheads. The amount of amorphous data-parallelism in irregular programs is usually dependent on the input data and varies in complex ways during the computation itself, as is the case for Delaunay mesh refinement, so analytical estimates are not possible. Therefore, we need a better approach to measure and express the parallelism of irregular programs.

This paper describes a tool called ParaMeter that uses instrumented execution to generate *parallelism profiles* of irregular programs. Intuitively, these profiles show how many (non-conflicting) worklist items can be executed concurrently at each step of the algorithm, assuming an idealized execution model in which (i) there are an unbounded number of processors and (ii) each worklist item takes roughly the same amount of time to execute (this is true in all our applications). ParaMeter also provides an estimate of the length of the critical path through the program under these assumptions. ParaMeter can also use information from the unbounded-processor parallelism profile to provide an estimate of the parallelism profile

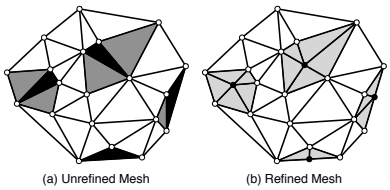



**Figure 2.** Mesh refinement.

```
1: Mesh mesh = /* 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:}
```

**Figure 3.** Pseudocode of the mesh refinement algorithm.

if some fixed number of processors are used to execute the program. Although ParaMeter uses the Galois infrastructure [20], the parallelism profiles it produces are independent of the infrastructure, and provide estimates of the amount of amorphous data-parallelism in the algorithm.

The remainder of this paper is organized as follows. Section 2 presents our notation for describing algorithms with amorphous data-parallelism. This notation was introduced in our earlier work on the Galois system [20]. Section 3 discusses metrics for estimating parallelism in programs. Section 4 describes how the ParaMeter tool is implemented. Section 5 analyzes the output of ParaMeter for seven irregular programs. Section 6 presents related work. We conclude in Section 7.

## 2. Program Notation and Execution Model for Amorphous Data-Parallelism

In this section, we briefly describe the programming notation and execution model for amorphous data-parallelism that we will use throughout this paper.

The programming model is a generic, sequential, object-oriented programming language such as Java augmented with two *Galois set iterators*:

- **Unordered-set iterator: for each e in Set S do B(e)**
  The loop body B(e) is executed for each element e of set S. Since set elements are not ordered, this construct asserts that in a serial execution of the loop, the iterations can be executed in any order. There may be dependences between the iterations, but any serial order of executing iterations is permitted. As an iteration executes, it may add elements to S.
- **Ordered-set iterator: for each e in Poset S do B(e)**
  This construct iterates over a partially-ordered set (Poset) S. It is similar to the Set iterator above, except that any execution order must respect the partial order imposed by the Poset S.

The use of Galois set iterators highlights opportunities in the program for exploiting amorphous data-parallelism. Figure 4 shows pseudocode for Delaunay mesh refinement, written using the unordered Galois set iterator. The unordered-set iterator implements "don't care non-determinism" [7] because it is free to iterate over the set in any order, and it allows the runtime system to exploit the fact that the Delaunay mesh refinement algorithm allows bad triangles to be processed in any order. This freedom is absent from the

```
1: Mesh mesh = /* read in initial mesh */
2: Worklist wl;
3: wl.add(mesh.badTriangles());
4: for each Triangle t in wl do {
5:   if (t no longer in mesh) continue;
6:   Cavity c = new Cavity(t);
7:   c.expand();
8:   c.retriangulate();
9:   mesh.update(c);
10:  wl.add(c.badTriangles());
11:}
```

**Figure 4.** Delaunay mesh refinement using a set iterator.

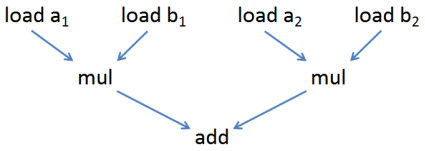



**Figure 5.** The computation DAG of $(a_1 * b_1) + (a_2 * b_2)$.

more conventional code in Figure 3, which constrains parallelism unnecessarily.

The execution model is the following. A master thread begins executing the program. When this thread encounters a Galois set iterator, it enlists the assistance of some number of worker threads to execute iterations concurrently with itself. The assignment of iterations to threads is under the control of a scheduling policy implemented by the runtime system. All threads are synchronized using barrier synchronization at the end of the iterator. Iterators may contain nested iterators, and the current Galois execution model "flattens" inner iterators, always executing them sequentially. The runtime system also performs conflict detection and resolution [20]. For ordered set iterators, the runtime system ensures that the iterations commit in the set order. We have used the Galois system to automatically parallelize a number of complex irregular applications including Delaunay mesh generation, mesh refinement, agglomerative clustering, image segmentation using graph cuts, *etc*. [19] for multicore processors. Although the speedup numbers from these experiments are useful, we note that they do not provide insight into how much parallelism there is to be exploited in irregular programs.

## 3. Parallelism in Programs

Parallelism in a program can be measured at different levels of granularity, ranging from the instruction level [1, 26] to the level of entire iterations or method invocations [2, 8]. For a given granularity level, parallelism is determined by generating a directed acyclic graph (DAG) in which nodes represent computations at that granularity level, and edges represent dependences between computations. To simplify the discussion, the nodes in the DAG are often assumed to take the same amount of time to execute.

### 3.1 Instruction-level parallelism

To introduce the key ideas, we use the relatively simple context of instruction-level parallelism in regular programs. Figure 5 shows an instruction-level DAG for the inner product of two vectors of length two. The maximum parallelism in this example is four. Moreover, because every path from a load to the addition contains three nodes, it will take three time units to perform this computation, assuming that each operation takes one time unit, even if there are an unbounded number of processors.

The longest path through the DAG is called the *critical path* (or the span) of the DAG. The length of the critical path is a lower bound on the execution time of the computation. If there are an unbounded number of processors and operations are scheduled as

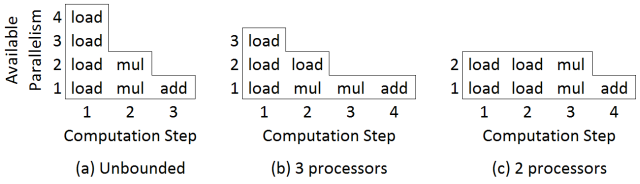



**Figure 6.** Parallelism profiles for the dot-product code.

early as possible, the width of the DAG at any level reflects the instantaneous parallelism, *i.e.*, the amount of available parallelism at a specific step, and the average width determines the average available parallelism.

The dataflow community has long used *parallelism profiles* [1] to visualize these kinds of execution metrics. The parallelism profile for a dataflow graph is a function $pp(t)$, which is the number of dataflow operators executed in each step $t$ on an idealized dataflow machine that has the following characteristics: (i) all dataflow operators take unit time, (ii) any number of operations can be performed in a single step, (iii) communication is instantaneous, and (iv) each operator executes as early as possible. The parallelism profile therefore represents a system-agnostic measure of how much parallelism an application has, independent of data locality, synchronization and communication overheads, *etc*. It provides important information such as whether it is profitable to run an application in parallel and, if so, how many processors we might allocate to its execution. For example, the parallelism profile of the dot-product computation in Figure 6(a) shows that it is useless to allocate more than four processors.

Parallelism profiles can also be generated for a fixed number of processors, provided a scheduling policy for selecting operations for execution at each step is specified [1]. We refer to this as the *constrained parallelism profile*. This information is useful to see how changing the number of processors changes parallelism behavior. Cilk, a language for writing task-parallel applications, uses a similar idea to estimate how much parallelism is available in a task DAG and how that parallelism can be exploited by a given number of processors [2, 8]. In the case of the dot product, the constrained parallelism profiles in Figure 6(b) and 6(c) reveal that using two or three CPUs results in the same runtime of four steps, but the utilization is better with two processors.

### 3.2 Amorphous data-parallelism

In this paper, we are interested in measuring the loop-level parallelism in the execution of Galois set iterators. At a high level, this can be accomplished by generating a computation DAG, similar to that of Figure 5, in which nodes represent iterations. However, it is important to realize that the behavior of irregular programs is vastly more complex than the behavior of the simple program shown in Figure 5 or even of dataflow programs. In these domains, the computation DAG for a program depends only on the input to the program and is independent of the execution schedule for those computations. Moreover, for many regular programs such as the BLAS kernels and most matrix factorization codes, the computation DAG usually depends on the size of the input but not the input values.

In contrast, the behavior of an amorphous data-parallel program is usually dependent not only on input values but also on the execution schedule of the computations, *even for an unbounded number of processors*. This is because an amorphous data-parallel program has a *computation graph* rather than a computation DAG: conflicting computations can be performed in any order, at least for unordered set iterators. Executing an amorphous data parallel program thus requires choosing an order to perform conflicting

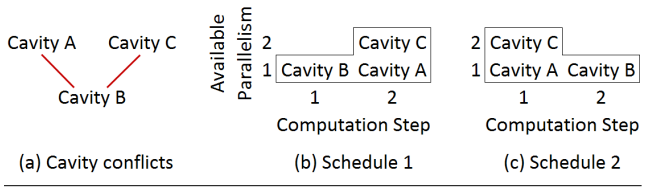



**Figure 7.** Parallelism graph with multiple resulting schedules.

computations. Scheduling conflicting tasks is equivalent to turning the computation graph into a computation DAG. However, different schedules of execution can produce different DAGs that have different parallelism profiles. Figure 7 shows an example in the context of Delaunay mesh refinement. If cavity B overlaps with cavities A and C, we cannot execute all three computations in one step, and the parallelism profile depends on scheduling choices even with an unbounded number of processors, as shown in Figures 7(b,c).

The previous example is relatively simple as the structure of the computation graph did not depend on scheduling decisions. In general, processing one piece of work may create new work or invalidate existing pieces of work. If a bad triangle B is in the cavity of another bad triangle A, processing A first will eliminate triangle B from the mesh. However, if triangle B is processed first, it may create other bad triangles, and depending on the schedule, these newly created bad triangles may be processed before triangle A is processed. The key issue is that the number of nodes and edges in the computation graph can change throughout the execution. Thus, there is no single "snapshot" of computation from which parallelism can be determined.

To sum up, amorphous data parallelism introduces two wrinkles to the standard notion of using DAGs to compute parallelism. First, there is the problem of *schedule dependence*: different execution orders produce different parallelism profiles. Second, there is the problem of *evolving computation*: the computation graph in an amorphous data parallel program changes dynamically.

### 3.3 Parallelism intensity

While parallelism profiles show the absolute amount of available parallelism, they do not reveal the full story. Consider two irregular programs, one with a worklist containing 1000 elements and the other with a worklist of 100 elements. Further, suppose that initially both programs have 100 elements that can be executed in parallel. From the perspective of available parallelism, both programs are identical. However, there is clearly a qualitative difference in the parallelism available in the two programs: the first program has little parallelism compared to the amount of work, while the second program is embarrassingly parallel.

What a parallelism profile does not capture is the amount of parallelism relative to the total amount of work that needs to be performed. We express this ratio using a metric called *parallelism intensity*. To compute this metric, we divide the amount of available parallelism by the overall size of the worklist at every point in the computation. Parallelism intensity is useful for choosing scheduling policies; for example, if the parallelism intensity is high, work can be picked at random from the worklist, but if the intensity is low, a random policy might result in frequent rollbacks because of conflicts [18].

## 4. The ParaMeter Tool

To handle the complexities inherent in profiling parallelism in irregular programs, we have developed a tool called ParaMeter. ParaMeter can profile irregular programs with both ordered and unordered amorphous data-parallel loops. By using instrumentation and an execution harness, ParaMeter is able to generate parallelism profiles for data parallel loops, measure parallelism intensity and model constrained parallelism.

As we outlined in the previous section, the highly dynamic and schedule-dependent nature of the computation DAGs of irregular, amorphous data-parallel programs makes it infeasible to generate a canonical parallelism profile for such applications. To tackle this problem, we first abandon the goal of an input-independent metric. Thus, the parallelism profiles ParaMeter generates for a program are input-dependent. In general, a profile may be valid for a class of inputs that behave similarly (for example, randomly generated input meshes for DMR), but will not be valid for all inputs.

Having accepted input dependence as a requirement for parallelism profiles, the second obstacle to generating parallelism profiles is the schedule dependence of irregular programs. Recall that in irregular programs, many edges in the computation graph may be undirected. Scheduling is the act of choosing directions for those edges, and different choices may lead to dramatically different DAGs. Furthermore, executing a node in a DAG may lead to the creation of additional DAG nodes. ParaMeter solves these problems by adopting two policies: *greedy scheduling* and *incremental execution*. Greedy scheduling means that at each step of execution, ParaMeter will try to execute as many elements as possible. This is equivalent to finding a maximally independent set of nodes in the computation graph. Thus, given the computation graph in Figure 7(a), ParaMeter would choose the schedule in Figure7(c) over the schedule shown in Figure 7(b).

To accommodate newly generated work, ParaMeter performs incremental execution. After performing greedy scheduling to determine which elements to execute in a computation step, ParaMeter fully executes them and adds any newly generated work to the computation graph. Thus, the next step of computation will be scheduled taking work generated in the previous step into account.

Conceptually, ParaMeter generates parallelism profiles by simulating the execution of a program on an infinite number of processors. Each computation step can be thought of as two phases: an *inspection phase*, which generates the computation graph from the current state of the worklist and identifies a maximal independent set of elements to execute, and an *execution phase*, which executes that independent set and sets up the worklist for the next step. Because generating an explicit conflict graph at each step is often computationally infeasible, the implementation of ParaMeter instead interleaves the inspection and execution phases while implicitly building the conflict graph. This procedure is explained in detail for unordered loops in Section 4.1. Section 4.2 discusses the changes required to profile ordered loops. We then discuss how we can use ParaMeter to determine parallelism profiles and other measures of parallel performance in Section 4.3.

### 4.1 Unordered loops

An intuitive approach to finding a maximal independent set of tasks to execute in a computation step is to examine the worklist, build a computation graph and find a maximal independent set in that computation graph. However, maintaining an explicit computation graph can be very expensive; in many scenarios, the worklist might contain tens of thousands of nodes, each of which may conflict with many other nodes. Representing such a graph can require large amounts of memory, and re-building the graph at each step can be computationally expensive[1].

ParaMeter uses a different approach. The algorithm shown in Figure 8. At each step, ParaMeter chooses a worklist element $e$ at random (line 6) and executes it speculatively (line 7). ParaMeter utilizes the machinery of the Galois system [20] to perform

[1] Nevertheless, some parallel implementations of Delaunay mesh refinement use this approach [15].

```
1  Set<Element> wl; //worklist of elements
2  wl = /* initialize worklist */
3  while (!wl.empty()) {
4    Set<Element> commitPool;
5    Set<Element> new_wl;
6    for (Element e : wl) { //random order
7      if (specProcess(e)) { //speculatively execute
8        new_wl.add(newWork(e)); //add new work
9        commitPool.add(e); //set e to commit
10     } else {
11       new_wl.add(e); //retry e in next round
12       abort(e);  //roll back e
13     }
14   }
15   for (Element e : commitPool) {
16     commit(e); //commit execution
17   }
18   wl = new_wl; //move to next round
19 }
```

**Figure 8.** ParaMeter pseudocode for unordered loops.

speculative execution using commutativity conditions (see Section 4.4 for further discussion of ParaMeter's speculative execution). If speculative execution succeeds, any new work generated by executing $e$ is added to a new worklist (line 8). However, the speculative execution of $e$ is not immediately committed. Rather, it is deferred until later (line 9). If, on the other hand, speculative execution does not succeed, then ParaMeter takes this as an indication that $e$ conflicts with some previously processed worklist element (as speculative execution of earlier elements has not yet committed). Thus, $e$ is assumed not to have executed in this computation step and is rolled back using the machinery of the Galois system (line 12). Its execution is deferred until the next step (line 11). Once the worklist is fully processed, all speculative execution is committed (lines 15–17), and the next computation step begins, using the elements on the new worklist (line 18).

The algorithm adopted by ParaMeter ultimately executes a maximally independent set of elements at each computation step, despite never explicitly computing a computation graph. To see why this is the case, consider the standard sequential algorithm for finding a maximal independent set in a graph:

1. Choose a node at random from the graph.
2. Remove the node and all its neighbors from the graph.
3. Repeat steps 1 and 2 until the graph is empty.

The nodes chosen in step 1 represent a maximal independent set. Note the following regarding conflicts during speculative execution: (i) if the speculative execution of two elements conflict, then the elements *would have been neighbors* in the computation graph; (ii) if the speculative execution of two elements do not conflict, then the elements *would not neighbor each other* in the conflict graph. Thus, by choosing elements at random from the worklist, executing them speculatively, and discarding those elements that conflict with previously executed elements, ParaMeter is emulating the algorithm for choosing a maximal independent set from a graph without explicitly building the computation graph.

Given this execution strategy, obtaining the parallelism profile is straightforward: the total number of elements committed at each computation step represent the amount of available parallelism in the profiled algorithm.

### 4.2 Ordered loops

In one sense, handling ordered loops is easier than profiling unordered loops; there are no scheduling effects to worry about. However, profiling ordered loops is more difficult than simply replacing the random iteration in Figure 8 with an iterator that respects the loop ordering—the parallel execution model for ordered loops requires that iterations appear to complete in order. Even if an element appears to be independent of all others in a given round, it cannot complete execution until it is the highest-priority element in the system. This leads to the following situation: an element can be executed in a particular round, but until it becomes the highest priority element in the system, it may be interfered with by higher priority work generated in a later round. Consider an ordered worklist that contains two elements, $a$ and $b$, in that order. Further, suppose that $a$ and $b$ are independent, but executing $a$ produces a new element $c$, which should execute before $b$ and conflicts with $b$. Even though $a$ and $b$ appear to be independent, $b$ will ultimately have to wait until $c$ executes before it can execute. However, if $b$ and $c$ *do not* interfere, then $b$ can safely execute in parallel with $a$ (even though its execution will not "commit" until later).

```
1  Set<Element> wl; //ordered worklist of elements
2  wl = /* initialize worklist */
3  Map<Element, integer> executionHistory;
4  int round = 0;
5  while (!wl.empty()) {
6    for (Element e : wl) { //in priority order
7      if (specProcess(e)) //speculatively execute
8        executionHistory.putIfAbsent(e, round);
9      else
10       executionHistory.remove(e);
11   }
12   for (Element e : wl) {
13     abort(e);
14   }
15   Set<Element> new_wl;
16   for (Element e : wl) { //in priority order
17     if (/*e.priority > elements in new_wl*/) {
18       execute(e); //execute non-speculatively
19       new_wl.add(newWork(e)); //add new work
20     } else {
21       new_wl.add(e); //element will be processed later
22     }
23   }
24   wl = new_wl; //move to next round
25   round++;
26 }
```

**Figure 9.** ParaMeter pseudocode for ordered loops.

The key difficulty in profiling an ordered loop is accounting for when an element executes. An element is not counted as executed when it first appears to be independent—as the previous example shows, the element's execution may conflict with work generated in later steps. Nor should an element be counted as executed when it eventually commits, as the element may have actually executed earlier and the cost of committing is minimal. Instead, ParaMeter counts an element as executed in the earliest round where it (i) is independent, and (ii) *remains independent* of newly generated work until it can eventually commit. The algorithm ParaMeter uses to determine when to count an element's execution is presented in Figure 9.

ParaMeter maintains a mapping from every element to the round it executed in, called `executionHistory`. At each computation step, ParaMeter speculatively executes every element in the worklist in order. If an element successfully executes, its execution time is recorded in `executionHistory` *if it does not already appear in the map* (line 8). Otherwise, the mapping in `executionHistory` is cleared (line 10). Next, all speculative execution is rolled back (line 13). Finally, ParaMeter "commits" the highest priority elements in the system by re-executing them (lines 16–23). Any newly created work will be executed in the next round (line 19). Note that ParaMeter does not add newly created work to the worklist until the element that generates it *commits*, regardless of when the element first became independent. Because lower priority work may conflict with this newly created work, it

cannot be executed in the current round and is deferred (line 21). This completes a single round of execution.

After all elements have been processed, ParaMeter assembles the parallelism profile by scanning `executionHistory` to see how many elements were counted for each round.

### 4.3 Metrics

There are several parallelism metrics that ParaMeter can measure. Other than the parallelism profile, it also measures *constrained parallelism* and the *parallelism intensity*.

***Parallelism profile*** The parallelism profile can be approximated by determining how many elements were executed in each round of execution. Since this data is maintained by ParaMeter, parallelism profiles can be generated simply by iterating through the rounds of execution and recording how many elements are executed in each round, as described in Section 4.

***Parallelism intensity*** ParaMeter computes the parallelism intensity by dividing the amount of work executed in each round by the total amount of work available for execution at that time, *i.e.*, the size of the worklist. As we discussed in Section 3.3, parallelism intensity provides some insight as to the effectiveness of random scheduling. Recall that the available parallelism is generated by executing a *maximally* independent set of elements in each round. However, in a parallel implementation of an irregular program, it may be infeasible to calculate an independent set of elements to execute. Instead, elements will be chosen based on some heuristic. Parallelism intensity correlates with the likelihood that random scheduling will actually choose independent iterations.

***Constrained parallelism*** Constrained parallelism estimates the critical path length of a program if it were to be run on a limited number of processors. This allows programmers to determine the incremental benefit of devoting additional processors to a problem. ParaMeter simulates the constrained parallelism by changing its behavior while executing elements: if the number of elements that can be executed in a round exceeds the number of simulated processors, all remaining elements are deferred to the next round.

Because it can be expensive to measure constrained parallelism, ParaMeter can also provide estimates of the constrained parallelism from the "unconstrained" parallelism profile. This estimate is based on the following simplifying assumption: if the execution of an element is deferred only because of processor constraints, it is assumed that it can still execute safely in subsequent rounds. This assumption drives ParaMeter's estimate of constrained parallelism. Consider estimating the critical path length of a program when run on $N$ processors. In each round, some number of elements are available to be executed in parallel. However, in a constrained setting, no more than $N$ can actually execute in a given round; all other elements are considered "excess" elements, and their execution must be deferred to a later round. ParaMeter examines each round of execution and determines the *total* number of excess elements, $p$. ParaMeter then assumes these $p$ elements can all be executed in parallel, taking $e = \lceil p/N \rceil$ rounds. The critical path length of the program when executed on $N$ processors is therefore estimated to be the number of rounds in the parallelism profile plus $e$.

### 4.4 Discussion

***Speculation*** Because ParaMeter's speculative execution is built on top of the Galois system, the system precisely captures which elements may be executed in parallel. ParaMeter uses commutativity checks to provide an algorithmic notion of conflict, based on the semantics of shared data structures, rather than the particular implementation of data structures. In fact, because ParaMeter is not tuned for parallel execution, it uses more precise checks than the commutativity checks of the Galois system, which are simplified for efficiency reasons.

Actual speculative systems using techniques such as thread level speculation [17, 23], Transactional Memory [11, 14], or partition locking [19] may generate false conflicts and hide parallelism that may be available. Thus, ParaMeter should be viewed as characterizing the parallelism in the *algorithm* rather than in a particular parallel implementation. However, the speculative execution framework used in ParaMeter can be replaced with other systems, allowing ParaMeter to simulate the execution of parallel algorithms when using, *e.g.*, transactional memory.

***Scheduling*** ParaMeter's greedy scheduling approach to profiling unordered loops is equivalent to simulating the execution of an algorithm on an infinite number of processors, with conflicts between concurrently executed tasks settled randomly. Note that this approach does not mean that ParaMeter necessarily finds the maximum amount of parallelism available in an algorithm; the scheduling heuristic finds a *maximal* set of independent iterations, not a *maximum* set. It is thus best to consider the amount of parallelism ParaMeter finds in an algorithm as an informative bound that approaches but does not necessarily reach the upper bound.

ParaMeter may produce a range of possible parallelism profiles when using different seeds in the random scheduler. For many algorithms, such as Delaunay mesh refinement, the variance in the parallelism profiles with different seeds is relatively small, as we see in Section 5.1. However, in some algorithms, different scheduling choices can lead to vastly different amounts of parallelism. An example of such an algorithm is agglomerative clustering, presented in Section 5.6.

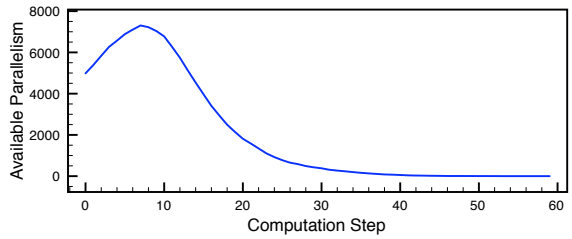
Interestingly, there are algorithms for which ParaMeter's random scheduling is *required* for completion. For example, some algorithms iterate over a set of active elements until some convergence criterion is reached. In these algorithms, active elements that do not help progress towards convergence are put back on the worklist, so their execution is effectively a no-op. If ParaMeter selected independent sets deterministically and every element in the independent set was a no-op, the algorithm would remain in the same state after processing those elements. ParaMeter would choose *exactly the same set of elements* in subsequent rounds and the algorithm would never terminate. By introducing some randomness to its choice of independent sets, ParaMeter avoids this issue when profiling algorithms such as survey propagation (Section 5.5) and agglomerative clustering (Section 5.6).

## 5. Experimental Results

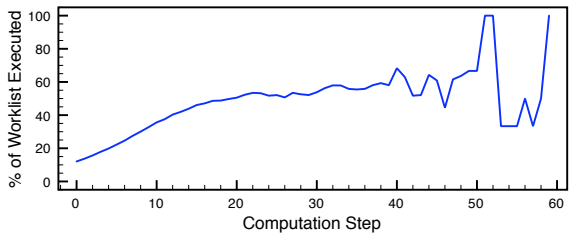
We evaluated ParaMeter on seven irregular applications exhibiting amorphous data-parallelism. Six of these applications use unordered set iterators: Delaunay mesh refinement, Delaunay triangulation, augmenting-paths maxflow, preflow-push maxflow, survey propagation and agglomerative clustering. We also evaluate a variant of agglomerative clustering that uses an ordered set iterator. For each application, we present the parallelism profile generated by ParaMeter and explain how the results correlate with the algorithm's characteristics. For three applications, Delaunay refinement, Delaunay triangulation and augmenting paths maxflow, we compare ParaMeter's estimate of constrained parallelism to an estimate computed using the parallelism profile, *cf.* Section 4.3.

### 5.1 Delaunay mesh refinement

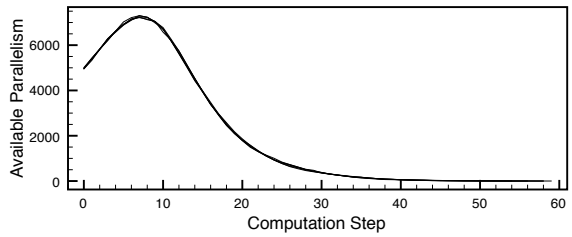
Delaunay mesh refinement (DMR) is the running example in this paper, and pseudocode for this algorithm was shown in Figure 3. The worklist is unordered and contains bad triangles from the mesh. As explained earlier, conflicts arise when the cavities of bad tri-




*(a) Parallelism profile*




*(b) Parallelism intensity*




*(c) Parallelism profiles from multiple ParaMeter runs*

**Figure 10.** Parallelism metrics for Delaunay mesh refinement.

angles overlap. For the experiments reported here, we used a randomly generated input mesh consisting of about 100,000 triangles, of which 47,000 are initially bad.

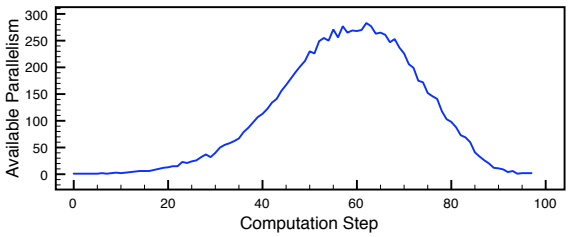
The parallelism profile for DMR is shown in Figure 10(a). Initially, there are roughly 5,000 bad triangles that can be processed in parallel. The amount of parallelism increases as the computation progresses, peaking at over 7,000 triangles at which point the available parallelism drops slowly. The parallelism intensity increases steadily as the computation progresses, as shown in Figure 10(b).

To explain the parallelism behavior of DMR, it is illustrative to consider how the application behaves in the abstract. The mesh can be viewed as a graph, with each triangle representing a node in the graph, and edges in the graph representing triangle adjacency. In this representation, processing a bad triangle is equivalent to removing some small connected subgraph from the overall mesh (the cavity) and replacing it with a new subgraph containing more nodes (the re-triangulated cavity). Therefore, as the application progresses, the graph becomes larger. However, the average size of a cavity remains the same, regardless of the overall size of the mesh. Hence, as the graph becomes refined, the likelihood that two cavities overlap decreases. Therefore, the available parallelism increases initially and then drops when the computation starts to run out of work, while the probability of conflicts decreases as the computation progresses.

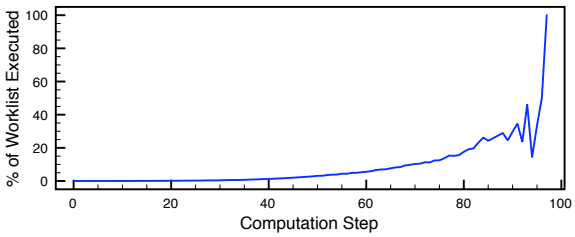
Figure 10(c) shows the result of profiling DMR with ParaMeter five times with different seeds, overlaid on a single plot. The amount of parallelism found in each step is roughly the same; the peak parallelism varies between 7,216 and 7,306 independent iterations, and the critical path varies between 54 and 59 steps. The behavior of many applications is similarly independent of the scheduling choices. Other applications are more sensitive to the scheduling, as we will see in Section 5.6.

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

**Figure 11.** Pseudocode for Delaunay triangulation.




*(a) Parallelism profile*




*(b) Parallelism intensity*

**Figure 12.** Parallelism metrics for Delaunay triangulation.

### 5.2 Delaunay triangulation

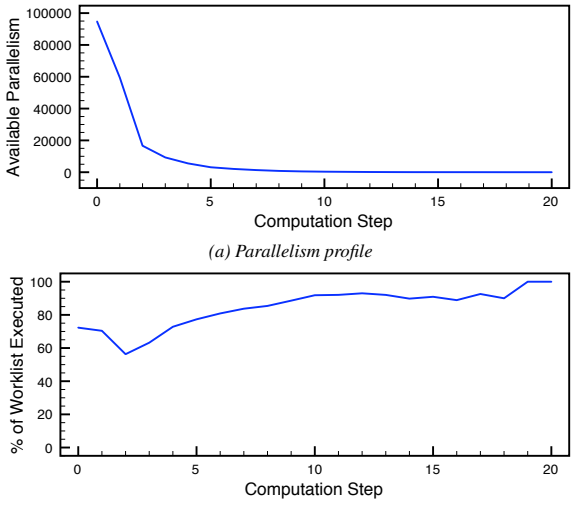
Delaunay triangulation (DT) is used to generate Delaunay meshes such as those used as input to DMR. We utilize an unordered, amorphous data-parallel algorithm by Guibas *et al.* [10]. Figure 11 shows the pseudocode. The input to DT is (i) a set of points in a plane and (ii) a large triangle in that plane that encompasses all those points. The output is the Delaunay triangulation of those points. The worklist consists of the input points. The mesh is built up incrementally. When a new point is inserted into the mesh, the triangle containing that point is split into three triangles, and the region around those triangles is re-triangulated to restore the Delaunay property. Thus, two points conflict with one another if their insertion would modify the same parts of the mesh.

At the beginning of execution, there is very little parallelism because all points attempt to split the same triangle and conflict with one another. This can be seen in the parallelism profile generated by ParaMeter for an input of 10,000 random points, shown in Figure 12(a). Initially, there is no parallelism (only one element can be executed per round), but as execution progresses, the amount of parallelism increases, and then decreases as the worklist is drained.

We notice that, in the abstract, DT, much like DMR, is a graph refinement code. It is thus not surprising that the parallelism profile of DT has the same characteristic bell shape as that of DMR. The parallelism intensity demonstrates the increase in parallelism that we expect of refinement codes, whereas the parallelism profile reflects a lack of work in later rounds.

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

**Figure 13.** Pseudocode for augmenting paths algorithm.




*(a) Parallelism profile*

*(b) Parallelism intensity*

**Figure 14.** Parallelism metrics for augmenting paths maxflow.

### 5.3 Augmenting paths maxflow

The Boykov-Kolmogorov algorithm for image segmentation [3] (AP) uses the idea of *graph-cuts*. This algorithm builds a graph in which there is a node for each pixel in the image and directed edges between two nodes if the corresponding pixels are adjacent in the image. There is also a source node and a sink node, each of which is connected to all the pixel nodes. In our experiments, the image is 512x512 pixels, so the bulk of the graph is a grid of that size. The algorithm assigns certain weights to nodes and edges. The main computation is a max-flow computation using the well-known principle of augmenting paths; pseudocode is provided in Figure 13. Search trees are built from both the source and the sink. When these trees connect, an augmenting path is found and the flow is augmented. Saturated edges are then dropped from the trees, and tree-building is resumed incrementally in the new graph.

The worklist consists of nodes on the frontiers of the search trees; initially, it contains only the source and sink nodes. Processing a node from the worklist involves a *search* operation and potentially an *augment* step. The search operation examines the neighbors of the node in the residual graph and attempts to grow the tree by finding a node that is not yet part of either the source tree or the sink tree. An augment operation traverses an augmenting path

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

**Figure 15.** Pseudocode for preflow-push.

and updates values on nodes and edges along this path. Notice that search and augment operations do not modify the structure of the graph. Conflicts arise between two iterations if they perform overlapping searches or augments.

Since there are a lot of short paths from the source to the sink, we would expect conflicts to be rare. This is borne out in Figure 14(b), which shows that the parallelism intensity is usually above 50%. Given the high parallelism intensity, the general shape of the parallelism profile, seen in Figure 14(a), is largely governed by the amount of work in the worklist. From the structure of the graph, it is clear that the size of the worklist becomes large very quickly (representing roughly half of the total amount of work done by the program). After that, the amount of work and the parallelism decrease steadily, as shown in Figure 14(a). It is likely that the behavior of AP will be very different for graphs from a different domain.

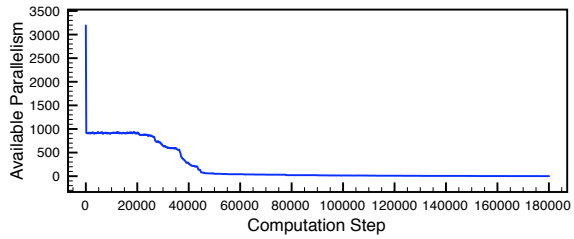
### 5.4 Preflow-push maxflow

Preflow-push (PP) is another algorithm for computing the maximum flow in a graph [9]. Pseudocode for preflow push is shown in Figure 15. Nodes maintain a *height*, which represents a lower bound on the node's distance to the sink. Nodes also maintain *excess flow*, which is the sum of all the flow entering that node. The worklist contains the nodes in the graph. The algorithm consists of pushing flow from a random node to a neighboring node of a lower height, and of raising the height of nodes so that they are higher than neighboring nodes. Both the push and relabel operations traverse a node and its neighbors, potentially updating their values.

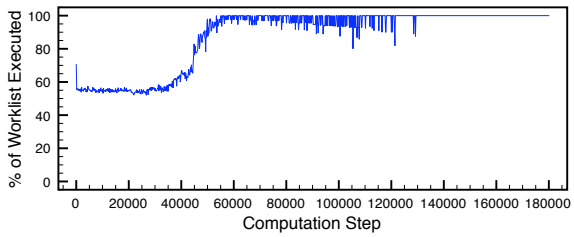
We use PP to solve the same types of image segmentation problems that AP is used for and hence utilize the same input graphs. Because operations in PP touch only few nodes, we expect it to behave similarly to AP. Figure 16(b) shows that the program is able to maintain an intensity over 50% throughout execution, just as AP does. Thus, the parallelism profile, in Figure 16(a), chiefly reflects the amount of work available to be processed. We see that there is significant work to do until there are only a few nodes with excess flow left in the graph.

### 5.5 Survey propagation

Survey propagation (SP) for SAT is a heuristic solver based on Bayesian inference [4]. Pseudocode is given in Figure 17. The underlying graph in SP is the *factor graph* for the Boolean formula, a bipartite graph in which one set of nodes represents the variables in the formula, and the other set of nodes represents the clauses. An edge links a variable to a clause if the variable or its negation is a literal in that clause. The worklist for SP consists of all nodes (both variables and clauses) in the graph. 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. The termination condition for SP is fairly complex – when the number of variables is small enough, or SP has




*(a) Parallelism profile*




*(b) Parallelism intensity*

**Figure 16.** Parallelism metrics for preflow-push maxflow.

```
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 17.** Pseudocode of survey propagation.

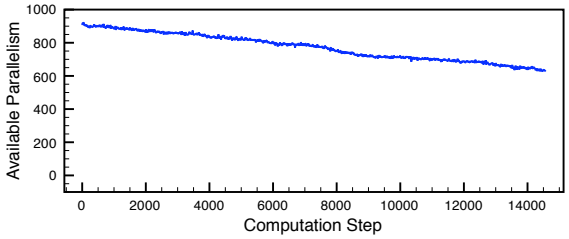
not made progress after some number of iterations, the SP iterations are terminated and the remaining problem is solved using a local heuristic like WalkSAT.

Abstractly, each operation in SP is a traversal and update of the nodes in a small neighborhood of the graph, with elements conflicting if the neighborhoods overlap. However, unlike in the maxflow problems, every node in the graph is in the worklist for SP. As a result, we would expect a higher amount of available parallelism (relative to the size of the input graph), but a lower parallelism intensity. Profiling SP using ParaMeter with a 3-CNF formula (a conjunction of 3-variable disjunctions) consisting of 1000 variables and 4200 clauses produces the results seen in Figure 18, which bear out this intuition.

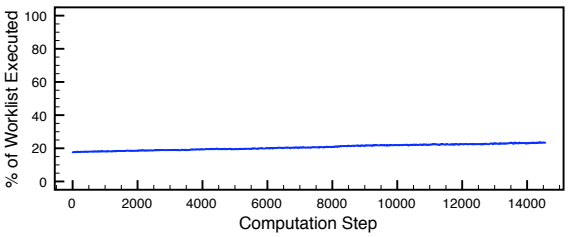
As we might expect from the above discussion, as variables are frozen and removed from the graph, the connectivity of the graph drops, and the worklist gets smaller. Thus, the amount of available parallelism drops (as there is less overall work to do), but the parallelism intensity increases (as the graph connectivity is lower, decreasing the chance of conflicts). It is important to note that, unlike the other applications we have examined, the termination condition for SP is *not* when the worklist empties. Thus, at the end of the execution, there are still several hundred elements in the worklist, and the parallelism intensity is still fairly low.

### 5.6 Agglomerative clustering (unordered)

The sixth application that we profiled is agglomerative clustering (AC). The goal of agglomerative clustering is to build a binary tree, called a *dendrogram*, representing a clustering of a set of points in




*(a) Parallelism profile*




*(b) Parallelism intensity*

**Figure 18.** Parallelism metrics for survey propagation.

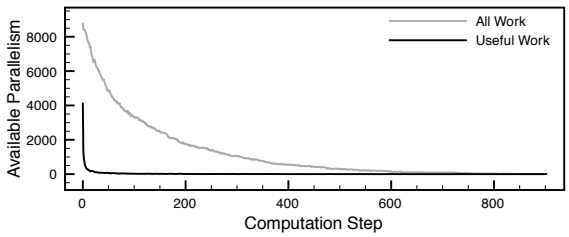
```
1: worklist = new Set(input_points);
2: for each Element a in worklist do {
3:   b = nearest(a); //closest point to a
4:   if (b == null) break; //stop if a is last element
5:   c = nearest(b); //closest point to b
6:   if (a == c) {
       //create new cluster e that contains a and b
7:     Element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

**Figure 19.** Pseudocode for unordered agglomerative clustering.

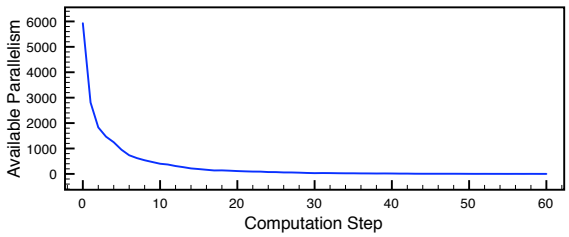
space. The algorithm proposed by Walter *et al*. uses an unordered set iterator [28]. Pseudocode is shown in Figure 19. The algorithm proceeds as follows. The worklist is initialized with all the points. Each point determines its nearest neighbor in the space. If two points agree that they are closest to each other, they are clustered, and a new point representing the cluster is inserted into the space. This is fundamentally an amorphous data-parallel algorithm, with worklist elements conflicting when the results of nearest neighbor computations are changed (due to insertions or deletions of points from the space).

Abstractly, we can think of AC as a bottom-up tree-building algorithm. If there were no conflicts between elements other than the dependences inherent in building a tree, we see that the structure of the tree limits the amount of parallelism available in the program: a bushy tree will have a lot of parallelism (as the lower levels of the tree have a lot of parallelism) while a skinny tree will have little parallelism (as each level is dependent on the level below).

We profiled AC with an input of 20,000 randomly generated points, and the results are shown in Figure 20. There appears to be a significant amount of parallelism in Figure 20(a). However, we note that significantly more work is performed than should be necessary to build the dendrogram. This is because processing a point in the worklist only to find that it cannot be clustered yet counts as a piece of work; there is a significant amount of work in the worklist that is "non-progressive," not contributing to the eventual solution. If we record only the worklist elements that contribute to the dendrogram, we obtain the black line in Figure 20(a) and significantly less *useful* available parallelism. Nonetheless, from the structure of the parallelism profile, we still see the expected




*(a) Parallelism profile (random schedule)*




*(b) Parallelism profile (oracle schedule)*

**Figure 20.** Parallelism metrics for unordered agglomerative clustering.

behavior with a lot of parallelism at the beginning (when the lower levels of the tree are being constructed) and far less at the end.

Note, however, that executing non-progressive work may prevent other, useful work from being executed in a given round (due to conflicts between worklist elements). We modified ParaMeter to use *oracle scheduling* rather than random scheduling. Thus, when choosing elements to execute in a given round, ParaMeter only considers elements that contribute to the solution. This approach generated the profile shown in Figure 20(b). We see that the oracle scheduler exhibits much higher parallelism than random scheduling. We also note that the critical path length when using the oracle scheduler is an order of magnitude shorter than when using the random scheduler. This is a prime example of algorithms where particular scheduling choices lead to dramatically different amounts of parallelism. Interestingly, with oracle scheduling, we find that AC is able to achieve the maximum possible parallelism available in a bottom-up tree building code.

### 5.7 Agglomerative clustering (ordered)

There is an alternative, greedy algorithm for agglomerative clustering (GAC) that uses an ordered set iterator [25]. Pseudocode is shown in Figure 21. The worklist is initialized as follows: for each point $p$, find the closest point $n$, and insert the *potential cluster* $(p, n)$ into the worklist. The worklist is ordered by the size of the potential clusters (*i.e.*, the potential cluster whose points are closest together is at the top of the list). The algorithm proceeds by greedily clustering the top potential cluster in the worklist. After clustering the two points, a new potential cluster is inserted containing the newly created cluster and the point nearest to it. Interestingly, this greedy algorithm produces the same dendrogram as the unordered algorithm presented in the previous section.

When we profile GAC with ParaMeter on the same input as for AC, we obtain the parallelism profile shown in Figure 22. Note that the first point in the profile is cut off and has a value of 5298. While there is a significant amount of parallelism in the first round, there is very little parallelism thereafter. This is due to the nature of the ordered execution strategy. In the first round, many of the potential clusters representing the lowest level of the dendrogram can be executed in parallel. However, recall that for ordered loops ParaMeter does not let a worklist element commit until it is the highest priority element in the system—even if its execution is accounted for in an earlier round—and new work is not added to

```
1:  pq = new PriorityQueue();
2:  foreach p in input_points {pq.add(<p, nearest(p)>)}
3:  foreach Pair <p,n> in pq {
4:    if (p.isAlreadyClustered()) continue;
5:    if (n.isAlreadyClustered()) {
6:      pq.add(<p, nearest(p)>);
7:      continue;
8:    }
9:    Cluster c = new Cluster(p,n);
10:   dendrogram.add(c);
11:   Point m = nearest(c);
12:   if (m != ptAtInfinity) pq.add(<c,m>);
13: }
```

**Figure 21.** Pseudocode for ordered agglomerative clustering.

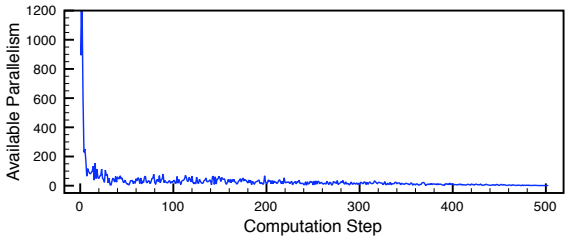



**Figure 22.** Parallelism profile for ordered agglomerative clustering

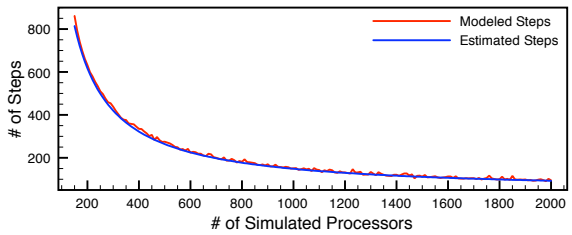
the worklist until the element that generates it commits. It takes several hundred steps for the elements recorded in the first round to actually commit, and the new work generated by the execution of those elements happens much later than the first round. In other words, work generated by those initial elements becomes available for execution across hundreds of rounds, rather than immediately. Thus, the critical path length of the application is inflated, resulting in lower available parallelism in later rounds.

### 5.8 Constrained Parallelism

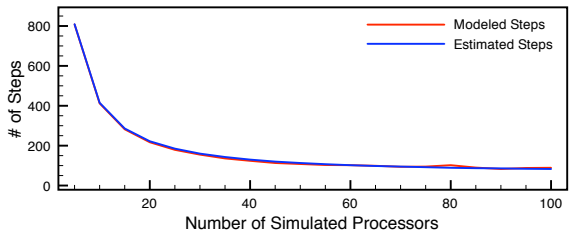
Beyond generating parallelism profiles and calculating parallelism intensity, ParaMeter can also calculate constrained parallelism, as described in Section 4.3. The goal of constrained parallelism is to estimate how the critical path of the execution changes if the number of processors available for execution is limited. Because it can be very expensive to calculate constrained parallelism for a large number of constraints, we can also use the parallelism profile (*i.e.*, the "unconstrained" parallelism) to estimate the length of the critical path when there are resource constraints.

We used ParaMeter to calculate the constrained parallelism for three of the applications we profiled: Delaunay mesh refinement, Delaunay triangulation, and augmenting paths maxflow. For each application, we found the critical path length for a range of processor counts. We also use the parallelism profiles to estimate the critical path length using the technique described in Section 4.3. The results are presented in Figure 23. In all three cases, our estimates of the critical path length closely track the critical path length measured by ParaMeter. These results demonstrate that it is feasible to use simple, easy-to-calculate models, as well as a single piece of profiling data (the parallelism profile) to estimate the parallelism in irregular programs for arbitrary numbers of processors.

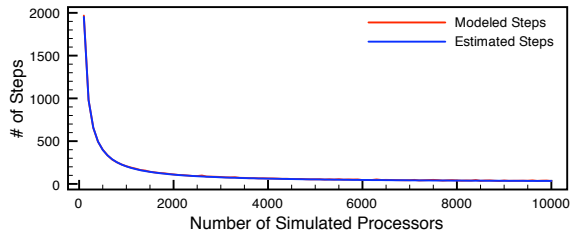
Constrained parallelism can inform the choice of platforms on which to run an application. We see that the constrained parallelism curves all follow the same general shape: a quick increase in performance when the number of processors is relatively low, and decreasing marginal benefits as the number of processors increases. The existence of a "knee" in the constrained parallelism curve is indicative of diminishing returns: beyond a certain point, there is little reason to increase the number of processors devoted to a task.




*(a) Delaunay mesh refinement*




*(b) Delaunay triangulation*




*(c) Augmenting paths maxflow*

**Figure 23.** Constrained parallelism: measured vs. estimated.

| Application | Unprofiled time | Profiled time |
|---|---|---|
| DMR | 18.0s | 57.4s |
| DT | 4.2s | 38.7s |
| AP | 0.36s | 19.4s |
| PP | 24.5s | 8m51.8s |
| SP | 3m54s | 25m30.2s |
| AC | 2.36s | 10m44.2s |
| AC(O) | N/A | 3m0.1s |
| GAC | 1.48s | 1m44.2s |

**Table 1.** Running times of unprofiled and profiled applications.

### 5.9 ParaMeter Performance

Table 1 shows the running time of each application when profiled with ParaMeter, as well as the running time of the non-profiled code (note that there is no sequential oracle scheduler). Our test platform is a dual-core AMD Opteron system with each core running at 1.8 GHz. ParaMeter and the test applications were written in Java and run using the Sun HotSpot JVM, version 1.6.

ParaMeter is designed as an off-line profiling tool, and hence performance is not the primary goal of its implementation. Nevertheless, we see that the tool has reasonable performance for most applications. The overhead is largely governed by two factors. The first is the cost of instrumentation. Thus, applications that require more instrumentation, such as augmenting paths and preflow push, suffer from higher overheads. Second, applications that have low parallelism intensity have higher overhead, as ParaMeter must continually inspect work that will ultimately not be executed. Unsurprisingly, AC has a very high overhead as the non-progressive work must be repeatedly executed by ParaMeter to no effect. Note that there is no equivalent sequential version of AC(O), as it requires an oracle scheduler.

## 6. Related Work

Much of the related work on profiling parallel execution is discussed in Section 3. Here, we highlight more recent work in the same area.

***Profiling for thread-level speculation*** In recent years, researchers have added profilers to compilers that target thread-level speculation systems [21, 29]. These profilers simulate the execution of a program on the target TLS system to determine how much parallelism can be exploited. While these approaches produce profile information akin to ParaMeter's parallelism profiles, there are two key differences.

The first difference recalls the distinction, discussed in Section 4.4, between profiling a particular parallelized program versus characterizing an algorithm. The TLS profiling tools are not used to measure parallelism in an algorithm; rather, they are used to measure the effectiveness of TLS approaches on certain regions of code. In contrast, ParaMeter's goal is to provide information about an *algorithm*, independent of the particular system used to parallelize that algorithm.

This difference of goals leads directly to the second distinction between ParaMeter and TLS-based profilers: because ParaMeter is independent of a parallelization system, it provides a more accurate view of the amount of parallelism in an algorithm. This is due to two effects. First, because TLS systems must adhere to the sequential ordering of the program; they are unable to find additional parallelism by executing worklist elements in other orders. Second, existing TLS profilers use memory-based conflict detection, which, as we discuss in Section 4.4, may obscure available parallelism.

***Profiling in functional languages*** For functional programs, Harris and Singh showed how profiling can be used to estimate available parallelism [12]. ParaMeter instead targets imperative programs that exhibit amorphous data-parallelism. This requires additional support for handling schedule dependence and conflicts in shared data structures.

***Dependence density*** Von Praun *et al.* recently studied "dependence density" in programs using optimistic concurrency [27]. While dependence density is a single number for a parallel section, it is essentially the inverse of the parallelism intensity. Thus, programs with a low dependence density can be easily executed by random schedulers, whereas those with higher dependence densities require more intelligent scheduling to exploit parallelism. However, von Praun *et al.* collect their data from an instrumented, sequential run of computation rather than staging computation as ParaMeter does. This leads to an inefficient handling of newly generated work: work generated at a later stage may conflict with work that could have been safely executed earlier. Further, their notion of dependence is tied to memory conflicts, which are more restrictive than ParaMeter's algorithmic conflicts, potentially resulting in an underreporting of the amount of parallelism.

## 7. Conclusions

This paper presents a tool called ParaMeter to determine how much parallelism is latent in irregular programs that exhibit amorphous data-parallelism. ParaMeter addresses some of the unique challenges presented by such programs: dynamically generated work and schedule dependent execution. While ParaMeter measures parallelism in amorphous data-parallel programs, its techniques can be extended to other models of parallelism in irregular programs. For example, thread-level speculation, which speculatively executes iterations of for-loops, can be profiled in ParaMeter by viewing loops as ordered worklists (with the order fixed by the for-loop).

The profiles generated by ParaMeter can be used to guide the development of parallel programs. Parallelism profiles produce a

"target" for parallelization: if an application's parallelism profile indicates little or no parallelism, there may not be a need for parallelization. Similarly, it may not be necessary to continue optimizing a parallel implementation that approaches the speedups predicted by the parallelism profile. Parallelism profiles can further be used to compare different implementations of the same algorithm. For example, we found that unordered agglomerative clustering exhibits more parallelism than the ordered counterpart.

Profiling parallelism intensity provides insight into the efficacy of random scheduling. High parallelism intensity makes it likely that randomly chosen elements can be executed in parallel, while low intensity makes scheduling difficult, even if there is high available parallelism. Finally, measuring constrained parallelism provides an estimate of how performance will be affected by changing the number of processors available to the program, which can be useful in choosing the right platforms for an application.

## References


[1] Arvind, David Culler, and Gino Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. *International Journal of High-performance Computing Applications*, 2(3), 1988.

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

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

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

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

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

[7] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

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

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

[11] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003.

[12] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 251–264, New York, NY, USA, 2007. ACM.

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

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, 1993.

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

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

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

[18] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA'08: Proceedings of the ACM Symposium on Parallel Architectures and Algorithms*, 2008.

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

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

[21] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, New York, NY, USA, 2006. ACM.

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

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

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

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

[26] K.B. Theobald, G.R. Gao, and L.J. Hendren. On the limits of program parallelism and its smoothability. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*, pages 10–19, Dec 1992.

[27] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 185–196, New York, NY, USA, 2008. ACM.

[28] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing (RT)*, 2008.

[29] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. *IEEE 14th International Symposium on High Performance Computer Architecture*, pages 290–301, Feb. 2008.