The Tao of Parallelism in Algorithms *

Keshav Pingali[1,3], Donald Nguyen[1], Milind Kulkarni[5],
Martin Burtscher[4], M. Amber Hassaan[2], Rashid Kaleem[1], Tsung-Hsien Lee[2], Andrew Lenharth[3],
Roman Manevich[3], Mario Méndez-Lojo[3], Dimitrios Prountzos[1], Xin Sui[1]

[1]Department of Computer Science, [2]Electrical and Computer Engineering and [3]Institute for Computational Engineering and Sciences
The University of Texas at Austin
[4]Department of Computer Science, Texas State University–San Marcos
[5]School of Electrical and Computer Engineering, Purdue University

Abstract

For more than thirty years, the parallel programming community has used the *dependence graph* as the main abstraction for reasoning about and exploiting parallelism in "regular" algorithms that use dense arrays, such as finite-differences and FFTs. In this paper, we argue that the dependence graph is not a suitable abstraction for algorithms in new application areas like machine learning and network analysis in which the key data structures are "irregular" data structures like graphs, trees, and sets.

To address the need for better abstractions, we introduce a data-centric formulation of algorithms called the *operator formulation* in which an algorithm is expressed in terms of its action on data structures. This formulation is the basis for a structural analysis of algorithms that we call *tao-analysis*. Tao-analysis can be viewed as an abstraction of algorithms that distills out algorithmic properties important for parallelization. It reveals that a generalized form of data-parallelism called *amorphous data-parallelism* is ubiquitous in algorithms, and that, depending on the tao-structure of the algorithm, this parallelism may be exploited by compile-time, inspector-executor or optimistic parallelization, thereby unifying these seemingly unrelated parallelization techniques. Regular algorithms emerge as a special case of irregular algorithms, and many application-specific optimization techniques can be generalized to a broader context.

These results suggest that the operator formulation and tao-analysis of algorithms can be the foundation of a systematic approach to parallel programming.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms: Algorithms, Languages, Performance

Keywords: amorphous data-parallelism, Galois system, irregular programs, operator formulation, tao-analysis.

* This work is supported in part by NSF grants 0923907, 0833162, 0719966, and 0702353 and by grants from IBM, NEC and Intel.

1. Introduction

道 n. /tau/, /dau/: **1. a.** the source and guiding principle of all reality as conceived by Taoists; **b.** process which is to be followed for a life of harmony. Origin: Chinese *dào*, literally, way. (adapted from Merriam-Webster)

Dense matrix computations arise naturally in high-performance implementations of important computational science methods like finite-differences and multigrid. Therefore, over the years, the parallel programming community has acquired a deep understanding of the patterns of parallelism and locality in these kinds of "regular" algorithms. In contrast, "irregular" data structures such as sparse graphs, trees and sets are the norm in most emerging problem domains such as the following.

- In social network analysis, the key data structures are extremely sparse graphs in which nodes represent people and edges represent relationships. Algorithms for betweenness-centrality, maxflow, etc. are used to extract network properties [10].
- Machine-learning algorithms like belief propagation and survey propagation are based on message-passing in a factor graph, a sparse bipartite graph [44].
- Data-mining algorithms like k-means and agglomerative clustering operate on sets and multisets [59].
- Simulations of electrical circuits and battlefields often use event-driven (discrete-event) simulation [49] over networks of nodes.
- Optimizing compilers perform iterative and elimination-based dataflow analysis on structures like inter-procedural control-flow graphs [1].
- Even in computational science, n-body methods use spatial decomposition trees [6], and finite-element methods use 2D and 3D meshes produced using algorithms like Delaunay mesh generation and refinement [11].

Unfortunately, we currently have few insights into the structure of parallelism and locality in irregular algorithms, and this has stunted the development of techniques and tools that make it easier to produce parallel implementations. Domain specialists have written parallel programs for some of the algorithms discussed above (see [5, 20, 31, 33] among others). There are also parallel graph libraries such as Boost [22] and STAPL [2]. However, it is difficult to extract broadly applicable abstractions, principles, and mechanisms from these implementations. Another approach is to use points-to and shape analysis [19, 27, 30] to find data structure invariants that might be used to prove independence of computations. This approach has been successful in parallelizing n-body methods like Barnes-Hut that are organized around trees [17], but most of the applications discussed above use sparse graphs with no particular structure, so shape analysis techniques fail to find any parallelism. These difficulties have seemed insurmountable, so irregular algorithms remain the Cinderellas of parallel programming in spite of their central role in emerging applications.

In this paper, we argue that these problems can be solved only by changing the abstractions that we currently use to reason about parallelism in algorithms. The most widely used abstraction is the *dependence graph* in which nodes represent computations, and edges represent dependences between computations; for programs with loops, dependence edges are often labeled with additional information such as dependence distances and directions. Computations that are not ordered by the transitive closure of the dependence relation can be executed in parallel. Dependence graphs produced either by studying the algorithm or by compiler analysis are referred to as *static* dependence graphs. Although static dependence graphs are used extensively for implementing regular algorithms, they are not adequate for modeling parallelism in irregular algorithms. As we explain in Section 2, the most important reason for this is that dependences between computations in irregular algorithms are functions of runtime data values, so they cannot be represented usefully by a static dependence graph.

To address the need for better abstractions, Section 3 introduces a *data-centric* formulation of algorithms, called the *operator formulation* of algorithms. In the spirit of Niklaus Wirth's aphorism "Program = Algorithm + Data Structure" [63], we express algorithms in terms of operations on abstract data types (ADTs), independently of the concrete data structures used to implement the abstract data types. This formulation is the basis of a structural analysis of algorithms that we call *tao-analysis* after the three key dimensions of the analysis. In the literature on parallel programming, there are many abstractions of parallel machines, such as a variety of PRAM models [32]. Tao-analysis can be viewed as an abstraction of algorithms that distills out properties important for parallelization, hiding unnecessary detail.

Tao-analysis reveals that a generalized data-parallelism called *amorphous data-parallelism* is ubiquitous in algorithms, as we discuss in Section 4. We also show that depending on the tao-structure of the algorithm, this parallelism may be exploited by compile-time, inspector-executor or optimistic parallelization, thereby unifying these seemingly unrelated techniques by consideration of the binding time of scheduling decisions. In addition, regular algorithms emerge as special cases of irregular algorithms.

In Section 5, we show how these concepts can be applied to the parallelization of many important algorithms from the literature. Some of the techniques discussed in this paper have been incorporated into the Galois system[1]. In Section 6, we give experimental results for three full applications that demonstrate the practical utility of these ideas. Extensions to the amorphous data-parallelism model are discussed in Section 7. We summarize the main contributions in Section 8.

2. Inadequacy of static dependence graphs

The need for new algorithmic abstractions can be appreciated by considering Delaunay Mesh Refinement (DMR) [11], an irregular algorithm used extensively in finite-element meshing and graphics. The Delaunay triangulation for a set of points in a plane is the triangulation in which each triangle satisfies a certain geometric constraint called the Delaunay condition [14]. In many applications, triangles are required to satisfy additional quality constraints, and this is accomplished by a process of iterative refinement that re-

[1] Available from `http://iss.ices.utexas.edu/galois`

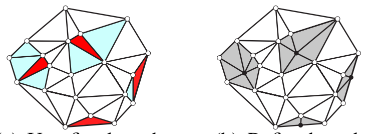

(a) Unrefined mesh (b) Refined mesh

Figure 1. Fixing bad triangles

```
Mesh mesh = // read in initial mesh
Worklist<Triangle> wl;
wl.add(mesh.badTriangles());
while (wl.size() != 0) {
  Triangle t = wl.poll(); // get bad triangle
  if (t no longer in mesh) continue;
  Cavity c = new Cavity(t);
  c.expand();
  c.retriangulate();
  mesh.update(c);
  wl.add(c.badTriangles());
}
```

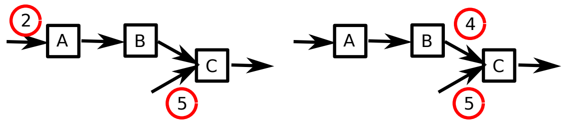
Figure 2. Pseudocode of the mesh refinement algorithm

peatedly fixes "bad" triangles (those that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating. Refining a bad triangle by itself may violate the Delaunay property of triangles around it, so it is necessary to compute a region of the mesh called the *cavity* of the bad triangle and replace all the triangles in the cavity with new triangles. Figure 1 illustrates this process; the darker-shaded triangles are "bad" triangles, and the cavities are the lighter shaded regions around the bad triangles. Re-triangulating a cavity may generate new bad triangles, but it can be shown that, at least in 2D, this iterative refinement process will ultimately terminate and produce a guaranteed-quality mesh. Different orders of processing bad triangles lead to different meshes, although all such meshes satisfy the quality constraints and are acceptable outcomes of the refinement process [11]. This is an example of Dijkstra's *don't-care non-determinism* (also known as *committed-choice non-determinism*) [16]. Figure 2 shows the pseudocode for mesh refinement. Each iteration of the while-loop refines one bad triangle; we call this computation an *activity*.

DMR can be performed in parallel since bad triangles whose cavities do not overlap can be refined in parallel. Note that two triangles whose cavities overlap can be refined in either order but not concurrently. Unfortunately, static dependence graphs are inadequate for exposing this parallelism. In Delaunay mesh refinement, as in most irregular algorithms, dependences between loop iterations (activities) are functions of runtime values: whether or not two iterations of the while-loop in Figure 2 can be executed concurrently depends on whether the cavities of the relevant triangles overlap, and this depends on the input mesh and how it has been modified by previous refinements. Since this information is known only during program execution, a static dependence graph for Delaunay mesh refinement must conservatively assume that every loop iteration might interfere with every prior iteration, serializing the execution.

A second problem with dependence graphs is that they do not model don't-care non-determinism. In dependence graphs, computations can be executed in parallel if they are not ordered by the dependence relation. In irregular algorithms like Delaunay mesh refinement, it is often the case that two activities can be done in either order but cannot be done concurrently, so they are not ordered by dependence and yet cannot be executed concurrently. Exploiting don't-care non-determinism can lead to more efficient parallel implementations, as we discuss in Section 4.

Event-driven simulation [49], which is used in circuit simulations and system modeling, illustrates a different kind of complex-



(a) Initial state (b) State after A and B have fired

Figure 3. Conflicts in event-driven simulation

ity exhibited by some irregular algorithms. This application simulates a network of processing stations that communicate by sending messages along FIFO links. When a station processes a message, its internal state may be updated, and it may produce zero or more messages on its outgoing links. Sequential event-driven simulation is implemented by maintaining a global priority queue of events called the event list and processing the earliest event at each step. In this irregular algorithm, activities correspond to the processing of events.

In principle, event-driven simulation can be performed in parallel by processing multiple events from the event list simultaneously, rather than one event at a time. However, unlike in Delaunay mesh refinement, in which it was legal to process bad triangles concurrently provided they were well-separated in the mesh, it may not be legal to process two events in parallel even if their stations are far apart in the network. Figure 3 demonstrates this. Suppose that in a sequential implementation, node A fires and produces a message time-stamped 3, and then node B fires and produces a message time-stamped 4. Notice that node C must consume this message *before* it consumes the message time-stamped 5. Therefore, in Figure 3(a), the events at A and C cannot be processed in parallel even though the stations are far apart in the network. However, notice that if the message from B to C had a time-stamp greater than 5, it would have been legal to process in parallel the events at nodes A and C. To determine if two activities can be performed in parallel at a given point in the execution of this algorithm, we need a crystal ball to look into the future!

In short, static dependence graphs are inadequate abstractions for irregular algorithms for the following reasons.

1. Dependences between activities in irregular algorithms are usually complex functions of runtime data values, so they cannot be usefully captured by a static dependence graph.
2. Many irregular algorithms exhibit don't-care non-determinism, but this is hard to model with dependence graphs.
3. Whether or not it is safe to execute two activities in parallel at a given point in the computation may depend on activities created later in the computation. It is not clear how one models this with dependence graphs.

3. Operator formulation of algorithms

These problems can be addressed by a data-centric formulation of algorithms, called the *operator formulation*, in which an algorithm is viewed in terms of its action on data structures. The operator formulation can be defined for any abstract data type, and we will use the *graph* ADT to illustrate the key ideas. As is standard, a graph is (i) a set of nodes V, and (ii) a set of edges $E \subseteq V \times V$ between these nodes. Graphs can be directed or undirected, and nodes and edges may be labeled with values.

We will not discuss a particular *concrete representation* for the graph ADT. The implementation is free to choose the concrete representation that is best suited for a particular algorithm and machine: for example, cliques may be represented using dense arrays, while sparse graphs may be represented using adjacency lists. This is similar to the approach taken in relational databases: SQL programmers use the relation ADT in writing programs, and the underlying DBMS system is free to implement the ADT using B-trees, hash tables, and other concrete data structures.

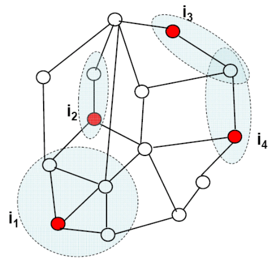


Figure 4. Active elements and neighborhoods

3.1 Active elements, neighborhoods and ordering

Active elements: At each point during the execution of a graph algorithm, there are certain nodes or edges in the graph where computation might be performed. These nodes and edges are called *active elements*; to keep the discussion simple, we assume from here on that active elements are nodes. To process an active node, an *operator* is applied to it, and the resulting computation is called an *activity*.

Neighborhoods: Performing an activity may require reading or writing other nodes and edges in the graph. Borrowing terminology from the literature on cellular automata, we refer to the set of nodes and edges that are read or written while performing an activity as the *neighborhood* of that activity. Figure 4 shows an undirected sparse 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 its neighbors in the graph.

Ordering: In general, there are many active nodes in a graph, so a sequential implementation must pick one of them and perform the appropriate activity. In some algorithms such as Delaunay mesh refinement, the implementation is allowed to pick *any* active node for execution. We call these *unordered* algorithms. In contrast, some algorithms dictate an order in which active nodes must be processed by a sequential implementation; we call these *ordered* algorithms. Event-driven simulation is an example: the sequential algorithm for event-driven simulation processes messages in global time order. The order on active nodes may be a partial order.

We illustrate these concepts using the algorithms from Section 2. In DMR, the mesh is usually represented by a graph in which nodes represent triangles and edges represent adjacency of triangles. The active nodes in this algorithm are the nodes representing bad triangles, and the neighborhood of an active node is the cavity of that bad triangle. In event-driven simulation, active nodes are stations that have messages on their input channels, and neighborhoods contain only the active node.

3.2 From algorithms to programs

A natural way to write these algorithms is to use worklists to keep track of active nodes. However, programs written directly in terms of worklists, such as the one in Figure 2, encode a particular order of processing worklist items and do not express the don't-care nondeterminism in unordered algorithms. To address this problem, we can use the Galois programming model, which is a *sequential*, object-oriented programming model (such as sequential Java), augmented with two *Galois set iterators* [40]:

DEFINITION 1. *Galois set iterators:*

- **Unordered-set iterator: foreach (e in Set S) {B(e)}**
The loop body B(e) is executed for each element e of set S. The order in which iterations execute is indeterminate and can be chosen by the implementation. There may be dependences

between the iterations. When an iteration executes, it may add elements to S.

- **Ordered-set iterator: foreach (e in OrderedSet S) {B(e)}** *This construct iterates over an ordered set S. It is similar to the unordered set iterator above, except that a sequential implementation must choose a minimal element from S at every iteration. When an iteration executes, it may add elements to S.*

```
Mesh mesh = // read in initial mesh
Workset<Triangle> ws;
ws.add(mesh.badTriangles());
foreach (Triangle t in Set ws) {
  if (t no longer in mesh) continue;
  Cavity c = new Cavity(t);
  c.expand();
  c.retriangulate();
  mesh.update(c);
  ws.add(c.badTriangles());
}
```

Figure 5. DMR using an unordered Galois set iterator

Iterators over multi-sets can be defined similarly. In principle, unordered iterators are a special case of ordered iterators, but they are sufficiently important in applications that we give them special treatment. Allowing break statements in the body of a Galois iterator permits early exits out of iterators. This can be useful for search problems. Some unordered algorithms can also be written using divide-and-conquer, as discussed in Section 7.

Set iterators were first introduced in the SETL programming language [57] and can now be found in most object-oriented languages such as Java and C++. However, *new elements cannot be added to sets while iterating over them, which is possible with Galois set iterators*. Note that the iterators have a well-defined sequential semantics. The unordered-set iterator specifies the don't-care nondeterminism of unordered algorithms.

The Galois system provides a (concurrent) data structure library, similar in spirit to the Java collections library, containing implementations of key ADTs such as graphs, trees, grids, work-sets, etc. Application programmers write algorithms in sequential Java, using the appropriate library classes and Galois set iterators. Figure 5 shows pseudocode for Delaunay mesh refinement, written using the unordered Galois set iterator. The body of the Galois set iterator is the implementation of the operator. Note that neighborhoods are defined *implicitly* by the calls to the graph ADT; for example, if the body of the iterator invokes a graph API method to get the outgoing edges from an active node, these edges become part of the neighborhood for that activity. Therefore, in general, the neighborhood of an activity is known only when that activity is completed. This has important implications for parallelization, as we discuss in Section 4.

3.3 Tao-analysis of algorithms

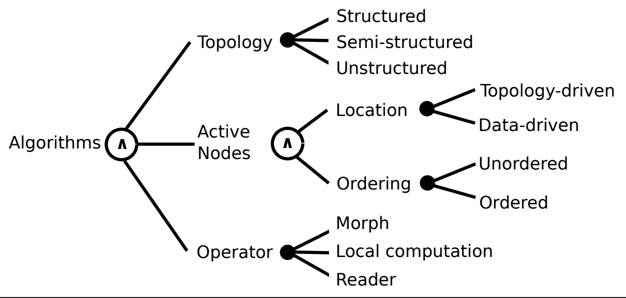


Figure 6. Structural analysis of algorithms

The operator formulation permits a natural structural analysis of algorithms along three dimensions (shown in Figure 6): Topology, Active nodes and Operator. The topology describes the data structure on which computation occurs. The active nodes dimension describes how nodes become active and how active nodes should be ordered, providing a global view of an algorithm. Finally, the operator describes the action of the operator on an active node, providing a local view of an algorithm. We call this *tao-analysis*. Exploiting this structure is key to efficient parallel implementations as we discuss in Section 4.

1. *Topology:* We classify graph topologies according to the (Kolmogorov) complexity of their descriptions. Highly structured topologies can be described concisely with a small number of parameters, while unstructured topologies require verbose descriptions. The topology of a graph is an important indicator of the kinds of optimizations available to algorithm implementations; for example, algorithms in which graphs have highly structured topologies may be amenable to static analysis and optimization.
   - *Structured:* An example of a structured topology is a graph consisting of labeled nodes and *no* edges: this is isomorphic to a set or multiset. Its topology can be described by a single number, which is the number of elements in the set/multiset. If the nodes are totally ordered, the graph is isomorphic to a sequence or stream. Cliques (graphs in which every pair of nodes is connected by a labeled edge) are isomorphic to square dense matrices (row/column numbers are derived from a total ordering of the nodes). Their topology is completely specified by a single number, which is the number of nodes in the clique. The final example we will consider in this paper is the rectangular grid; its topology is determined completely by two numbers, its height and width.
   - *Semi-structured:* We classify trees as semi-structured topologies. Although trees have useful structural invariants, there are many trees with the same number of nodes and edges.
   - *Unstructured:* General graphs fall in this category.
2. *Active nodes:* This dimension describes how nodes become active and the order in which they must be processed.
   - *Location:* Nodes can become active in a *topology-driven* or *data-driven* manner. In topology-driven algorithms, the active nodes are determined by the graph, so the execution of the operator at some active node does not cause other nodes to become active. Common examples are algorithms that iterate over all the nodes or edges of a graph. In data-driven algorithms, an activity at one node may cause other nodes to become active, so nodes become active in a data-dependent and unpredictable manner. Some examples are the preflow-push algorithm for maxflow computation in graphs and algorithms for event-driven simulation.
   - *Ordering:* As discussed above, active nodes in some algorithms are ordered whereas others are unordered.
3. *Operator:* We classify operators based on how they modify the graph.
   - *Morph:* A morph operator may modify its neighborhood by adding or deleting nodes and edges, and it may also update values on nodes and edges. The Delaunay mesh refinement operator is an example; other examples are discussed in Section 5.1.
   - *Local computation:* A local computation operator may update values stored on nodes and edges in its neighborhood, but it does not change the graph connectivity. Finite-difference computations are the classic example; other examples are discussed in Section 5.2.

$$\begin{tikzcd} L \arrow[r, "r"] \arrow[d, "m"'] & R \arrow[d, "m'"] \\ G \arrow[r, "r'"'] & H \end{tikzcd}$$

Figure 7. Graph rewriting: single-pushout approach

- *Reader:* An operator is a reader for a data structure if it does not modify it in any way. For example, the ray-tracing operator is a reader for the scene being rendered; other examples are discussed in Section 5.3.

These definitions can be generalized in the obvious way for algorithms that deal with multiple data structures. In that case, neighborhoods span multiple data structures, and the classification of an operator is with respect to a particular data structure. For example, in matrix multiplication $C = AB$, the operator is a local computation operator for C and a reader for matrices A and B.

3.4 Discussion

As mentioned above, under-specification of the order of processing active nodes in unordered algorithms is an instance of don't-care non-determinism. In some unordered algorithms, the output is independent of the order in which active nodes are processed, a property that is referred to as the *Church-Rosser property* since the most famous example of this behavior is β-reduction in λ-calculus. Dataflow graph execution [4, 15] and the preflow-push algorithm for computing maxflow [12] also exhibit this behavior. In other algorithms, the output may be different for different choices of active nodes, but all such outputs are acceptable, so the implementation can still pick any active node for execution. Delaunay mesh refinement and Petri net simulation are examples. The preflow-push algorithm also exhibits this behavior if the algorithm outputs both the min-cut and the max-flow. Even for unordered algorithms, iteration execution order may affect cache performance and the number of executed iterations, so control of iteration order is useful for efficiency. Nguyen and Pingali describe a notation and an implementation for doing this [50].

The metaphor of operators acting on neighborhoods is reminiscent of notions in term-rewriting systems, such as graph grammars in particular [18, 42]. The semantics of functional language programs are usually specified using term rewriting systems that describe how expressions can be replaced by other expressions within the context of the functional program. The process of applying rewrite rules repeatedly to a functional language program is known as string or tree reduction.

Tree reduction can be generalized in a natural way to graph reduction by using graph grammars as rewrite rules [18, 42]. A graph rewrite rule is defined as a morphism in the category $\mathbf{C}$ of labeled graphs with partial graph morphisms as arrows: $r : L \rightarrow R$, and a rewriting step is defined by a single pushout diagram [42] as shown in Figure 7. In this diagram, G is a graph, and the total morphism $m : L \rightarrow G$, which is called a *redex*, identifies the portion of G that matches L, the left-hand side of the rewrite rule. The application of a rule r at a redex m leads to a direct derivation $(r, m) : G \Rightarrow H$ given by the pushout in Figure 7.

In graph reduction, rewrite rules are applied repeatedly to the text of the *program* until a normal or head-normal form is reached; in the operator formulation on the other hand, the operator is applied to the *data structure* until there are no more active nodes. In addition, we do not require operators to be expressible as a finite set of "syntactic" rewrite rule schemas (it is not clear that Delaunay mesh refinement, for example, can be specified using graph grammars, although some progress along these lines is reported by Panangaden and Verbrugge [60]). Nevertheless, the terminology of graph grammars may be useful for providing a theoretical foundation for the operator formulation of algorithms. For example, in the context of Figure 7, the graph structure of L and R are identical for local computation operators, while for reader operators, the rewrite rule r is the identity morphism. We also note that whether operators make imperative-style in-place updates to graphs or create modified copies of the graph in a functional style can be viewed as a matter of implementation.

4. Amorphous data-parallelism

The operator formulation of an algorithm is not explicitly parallel, but Figure 4 shows intuitively how opportunities for exploiting parallelism arise in an algorithm: if there are many active nodes at some point in the computation, each one is a site where a processor can perform computation, subject to neighborhood and ordering constraints. When active nodes are unordered, the neighborhood constraints must ensure that the output produced by executing the activities in parallel is the same as the output produced by executing the activities one at a time in some order. For ordered active elements, this order must be the same as the ordering on active elements.

DEFINITION 2. *Given a set of active nodes and an ordering on active nodes,* amorphous data-parallelism *is the parallelism that arises from simultaneously processing active nodes, subject to neighborhood and ordering constraints.*

Amorphous data-parallelism is a generalization of conventional data-parallelism in which (i) concurrent operations may conflict with each other, (ii) activities can be created dynamically, and (iii) activities may modify the underlying data structure. Not surprisingly, the exploitation of amorphous data-parallelism is more complex than the exploitation of conventional data-parallelism. In this section, we present a baseline implementation that uses optimistic or speculative execution.

4.1 Baseline: speculative parallel execution

In the baseline execution model, the graph is stored in shared-memory, and active nodes are processed by some number of threads. A thread picks an active node from the work-set and speculatively applies the operator to that node, making calls to the graph API to perform operations on the graph as needed. The neighborhood of an activity can be visualized as a blue ink-blot that begins at the active node and spreads incrementally whenever a graph API call is made that touches new nodes or edges in the graph. To ensure that neighborhoods are disjoint, the concurrent graph class can use exclusive logical locks: each graph element has an exclusive lock that must be acquired by a thread before it can access that element. Locks are held until the activity terminates. If a lock cannot be acquired because it is already owned by another thread, a conflict is reported to the runtime system, which rolls back one of the conflicting activities. Lock manipulation is performed entirely by the methods in the graph class; in addition, to enable rollback, each graph API method that modifies the graph makes a copy of the data before modification, as is done in other systems that use speculation such as transactional memory and thread-level speculation [25, 29, 55, 61].

If active elements are not ordered, the activity commits when the application of the operator is complete, and all acquired locks are then released. If active elements are ordered, active nodes can still be processed in any order, but they must commit in serial order. This can be implemented using a data structure similar to a reorder buffer in out-of-order processors [40]. In this case, locks are released only when the activity commits (or is aborted). Exclusive logical locks can be implemented by using a compare-and-set instruction to mark a graph element with the id of the activity that touches it.

Neighborhoods of concurrent activities can be permitted to overlap if these activities do not modify nodes and edges in the intersection of these neighborhoods (consider activities i_3 and i_4 in Figure 4). Therefore, additional concurrency is possible if we recognize read-only data structures and do not lock their elements; reader/writer locks are another solution. The most general solution is to use commutativity conditions; this allows iterations to execute in parallel even if they perform reduction operations on shared variables, for example. To keep the discussion simple, we do not describe these alternatives here but refer the interested reader to Kulkarni *et al.* [39]. The literature on PRAM algorithms has explored similar variations such as the EREW and combining CRCW models [32, 62].

4.1.1 Parallelism profiles

For an algorithm like matrix multiplication for which a static dependence graph can be generated, it is possible to give closed-form estimates of the critical path length and the amount of parallelism at each point of execution (these estimates are usually parameterized by the size of the input). For example, in multiplying two $N \times N$ matrices using the standard algorithm, there are N^3 multiplications that can be performed in parallel. In contrast, amorphous data-parallelism in irregular algorithms is usually a function of runtime values, and closed-form estimates are not generally possible.

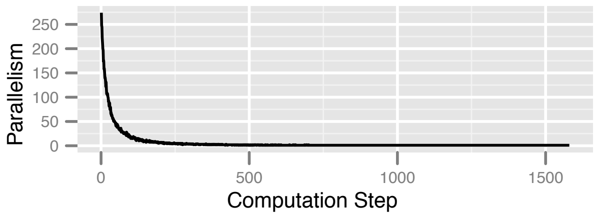
One measure of amorphous parallelism in irregular algorithms is the number of active nodes that can be processed in parallel at each step of the algorithm for a given input, assuming that (i) there is an unbounded number of processors, (ii) an activity takes one time step to execute, (iii) the system has perfect knowledge of neighborhood and ordering constraints so it only executes activities that can complete successfully, and (iv) a maximal set of non-conflicting activities is executed at each step. This is called the *available parallelism* at each step, and a graph showing the available parallelism at each step of execution of an irregular algorithm for a given input is called a *parallelism profile*. For algorithms for which a static dependence graph can be generated, the available parallelism at any step corresponds to the width of the dependence graph at that step. In this paper, we will present parallelism profiles produced by the ParaMeter tool [38].

Figure 8 shows the parallelism profiles of Boruvka's and Prim's minimal spanning tree (MST) algorithms; the input is a random graph. As explained in Section 5.1, Boruvka's algorithm is unordered while Prim's algorithm is ordered. At first sight, it is surprising that an ordered algorithm like Prim has any parallelism, but an intuitive explanation is that processing active nodes in the specified order is sufficient but not necessary to produce correct results. A parallel implementation can process active nodes out of order, and as long as no conflicts are detected before the activity commits, the output produced by the parallel execution will be correct. This is similar to how out-of-order execution processors find parallelism in sequential machine language programs. In our experience, many problems can be solved by both ordered and unordered algorithms, but the critical path is shorter for the unordered algorithm although it may perform more work than its ordered counterpart (see Hassaan *et al.* [26] for more details).

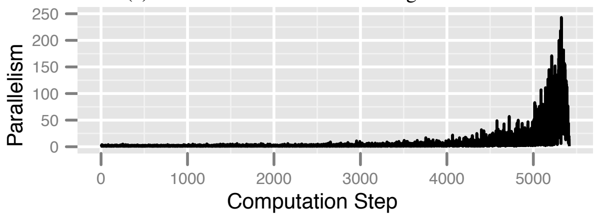
4.2 Exploiting structure to reduce overheads

The overheads of the baseline system can be reduced by exploiting structure when it is present. The following structure is very important in applications.

DEFINITION 3. *An implementation of an operator is said to be* cautious *if it reads* all *the elements of its neighborhood before it modifies* any *of them.*



(a) Boruvka's unordered MST algorithm



(b) Prim's ordered MST algorithm

Figure 8. MST parallelism profiles of a random graph with 10,000 nodes and 35,605 edges; the edges have random, uniformly distributed weights between 1 and 1,000.

Operators can usually be implemented in different ways: for example, one implementation might read node A, write to node A, read node B, and write to node B, in that order, whereas a different implementation might perform the two reads before the writes. By Definition 3, the second implementation is cautious, but the first one is not. In our experience, the natural implementations of most operators such as Delaunay mesh refinement are cautious[2]. In contrast, the operator for the well-known Delaunay triangulation algorithm of Guibas, Knuth and Sharir [23] does not have a naturally cautious implementation. It performs graph mutations called edge flips, which are done incrementally.

Unordered algorithms with cautious operator implementations can be executed speculatively without buffering updates or making backup copies of modified data because all conflicts are detected during the read-only phase of the operator execution. Blandford *et al.* [7] exploit this optimization in their DMR implementation; phrasing this optimization in terms of algorithmic structure permits a general-purpose system like Galois to use it for other algorithms (see Méndez-Lojo *et al.* [47]). Prountzos *et al.* [54] present a shape analysis for determining cautiousness using static analysis.

4.3 Exploiting structure for coordinated scheduling

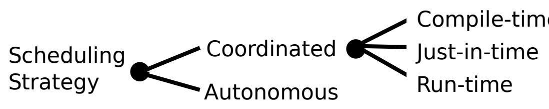


Figure 9. Scheduling strategies

The scheduling strategy implemented in the baseline system can be called *autonomous scheduling* because activities are executed in an uncoordinated way, requiring online conflict detection and rollback for correct execution. Speculative overheads are eliminated if we ensure that only non-conflicting iterations are scheduled for simultaneous execution, a strategy that we call *coordinated scheduling*. Figure 9 shows a number of coordinated scheduling strategies. All of these strategies are based on constructing a dependence graph either explicitly or implicitly, but they do so at different points during program compilation and execution.

[2] In principle, every operator has a trivial cautious implementation that touches all graph elements before beginning the computation (equivalently, it locks the whole graph). The obvious disadvantage is that this approach eliminates all parallelism.

In terms of the operator formulation, construction of the dependence graph involves the following steps: (i) determine all active nodes, (ii) determine neighborhoods and ordering of the corresponding activities, and (iii) create a partial order of activities that respects neighborhood and ordering constraints. Activities can then be executed in parallel without speculation but with proper synchronization to ensure that the partial order of activities is respected. Although this approach seems obvious, ordered, data-driven algorithms like event-driven simulation cannot in general be parallelized using this approach: the execution of one activity may cause a new node to become active, and this new activity may have higher priority than and conflict with existing activities. It is likely that optimistic parallelization is the only general-purpose approach for parallelizing ordered, data-driven algorithms, even if they are expressed in a functional language. All other classes of algorithms can be executed without speculation as described below.

4.3.1 Runtime coordination

Unordered data-driven algorithms: These algorithms can be parallelized without speculation by interleaving the construction of the dependence graph with execution of activities. The execution of the algorithm proceeds in rounds. In each round, a set of non-conflicting activities is selected and executed in parallel without synchronization. Any newly created activities are postponed to the next round at which point they are considered for execution together with unprocessed activities from the current round. Implementing this strategy requires solving two problems: (i) how do we compute the neighborhoods of activities, and (ii) given a set of activities and their neighborhoods, how do we find a set of non-conflicting activities?

In general, we must execute the operator completely to find the neighborhood of an activity, and this may cause side-effects to global data structures. If an activity is not chosen for execution in the current round (because of conflicts), these side-effects must be undone for correct execution. One solution is to privatize these updates as is done in some versions of transactional memory. In effect, activities are executed twice in each round: once with privatization to determine neighborhoods, and then again for real if they are chosen for execution in the current round. Implemented literally, this strategy is not very efficient; fortunately, most algorithms have structure that can be exploited to eliminate the need for repeated execution. In all data-driven local computation algorithms we have studied (see Section 5.2.2), the neighborhood of an activity is just the active node and its immediate neighbors in the graph. In morph algorithms, most operator implementations are cautious, so the neighborhood of an activity can be determined by executing it partially up to the point where it starts to make modifications to its neighborhood. If the activity is chosen for execution in that round, execution simply continues from that point.

Once the neighborhoods of all activities have been determined, we can build a *conflict graph* in which nodes represent the active nodes from the algorithm, and edges represent conflicts between activities. Luby's randomized parallel algorithm can be used to find a *maximal independent set* of activities [43], and this set of activities can be executed in parallel without synchronization. This approach can be viewed as building and exploiting the dependence graph level by level, with barrier synchronization between levels. The DMR implementation of Hudson *et al.* uses this approach [31].

Topology-driven algorithms: In these algorithms, activities do not create new active nodes, so both unordered and ordered algorithms can be executed using runtime coordination. Unordered topology-driven algorithms can be executed in rounds as described above. A variation of this approach can be used for ordered topology-driven algorithms. In this case, we find a maximal prefix of the sequence of active nodes (rather than a maximal independent set of the set of active nodes) such that all active nodes in the prefix have non-interfering neighborhoods, and execute these nodes in parallel. This process can then repeated with the remaining suffix of active nodes.

Notice that runtime coordination can be used for all irregular algorithms other than ordered, data-driven algorithms.

4.3.2 Just-in-time coordination

For some topology-driven algorithms, the dependence graph is independent of the labels on nodes and edges of the graph and it is a function purely of the graph topology. Therefore, a dependence graph can be generated at runtime after the input graph is given but before the program is executed. We call this strategy *just-in-time coordination*, and it is a generalization of the *inspector-executor method* of Saltz *et al.* [64].

For topology-driven algorithms, active nodes are known once the input is given, so the remaining problems are the determination of neighborhoods and ordering. In local computation algorithms amenable to just-in-time scheduling, these can usually be determined from an inspection of the graph and the resulting code is called the *inspector*. A well-known example is the implementation of sparse iterative solvers on distributed-memory computers. The distribution of the graph between processors is known only at runtime, so inspection of the graph is required to determine input dependences for use in communication schedules, as advocated by Saltz *et al.* [64]. Other examples are parallel top-down and bottom-up walks of trees. In a bottom-up walk, a recursive descent from the root sets up the dependences, and the bottom-up computations can then be done in parallel with appropriate synchronization; an example is the center-of-mass computation in n-body methods.

For topology-driven *morph* algorithms amenable to just-in-time scheduling, inspection of the graph may be insufficient since the structure of the graph can be modified during the execution of the algorithms, changing neighborhoods and dependences. For these algorithms, neighborhoods and ordering can be determined by a symbolic execution of the algorithm. Parallel sparse Cholesky factorization is the most famous example of this approach [20]. The algorithms is unordered but heuristics like minimal degree ordering are used to order the active nodes for efficiency. The algorithm is executed symbolically to compute the dependence graph (this phase is called *symbolic factorization* and the dependence graph is called the *elimination tree*). The elimination tree is then used to perform the actual factorization in parallel (this phase is known as *numerical factorization*). Symbolic factorization is done efficiently using Boolean operations, and for large matrices, it takes little time relative to numerical factorization. Just-in-time coordination cannot be used for sparse LU factorization with pivoting since its dependence graph is affected by pivoting, which depends on the data values in the matrix.

4.3.3 Compile-time coordination

Some algorithms amenable to just-in-time coordination have structured input graphs. In that case, the dependence graph, suitably parameterized by the unknown parameters of the input graph, can be produced at compile-time. For example, if the graph is a grid as it is in finite-difference methods, the dependence graph is parameterized by the dimensions of the grid. Other important examples are dense Cholesky and LU factorization (the input graph is a clique), the dense Basic Linear Algebra Subroutines (the input graph is a clique), and FFTs (sequences or sequences of sequences).

Given the importance of these algorithms in high-performance computing, it is not surprising that *automatic* generation of dependence graphs for this class of algorithms has received a lot of attention. These techniques are known as *dependence analysis* in the literature, and methods based on integer linear programming are successful for regular, dense array programs in which array subscripts are affine functions of loop indices [35]; LU with pivoting requires fractal symbol analysis [48].

Compile-time coordination is also possible regardless of the topology if the algorithm is a topology-driven local computation and the neighborhood of an activity is just the active node itself, ignoring read-only data structures. In this case, activities are trivially independent (subject to ordering constraints) because each one modifies a disjoint portion of the graph. A typical example is computing some result for each node in a graph. The output is a vector indexed by node. Each activity reads some portion of the graph but only writes to the node-specific portion of the output vector. In the literature, most PRAM algorithms and algorithms written using DO-ALL loops fall in this category.

4.4 Discussion

To the best of our knowledge, the first use of optimistic parallelization was in Jefferson's Timewarp system for event-driven simulation [33]. However, Timewarp was not a general-purpose parallel programming system. Thread-level speculation (TLS) was proposed by Rauchwerger and Padua [55] for parallelizing array programs in which subscripts could not be analyzed by the compiler. This work was very influential, and it inspired a lot of work on architectural support for speculative execution [61]. TLS systems were designed for languages like FORTRAN and C, so there was no support for don't-care non-determinism or unordered algorithms. Moreover, data abstractions do not play a role in the implementation of TLS. Another influential line of work is the transactional memory (TM) work of Herlihy, Moss, Harris and others [25, 29]. In contrast to our approach, the application programming model is *explicitly parallel*; threads synchronize using transactions, and the overheads of executing this synchronization construct are reduced using optimistic synchronization. In addition, most TM systems other than boosted systems [28] perform memory-level conflict checking rather than ADT-level conflict checking. This results in spurious conflicts that prevent efficient parallel execution. For example, if an iteration of an unordered iterator is rolled back, the concrete state of the work-set must also be restored to its original state, and all other iterations that were executed since that iteration must also be rolled back [40].

Relatively little is known about automatic generation of dependence graphs for runtime and just-in-time coordination. The discussion in this section shows that there is no sharp dichotomy between regular and irregular algorithms: instead, regular algorithms are a special case of irregular algorithms in the same way that metric spaces are a special case of general topologies in mathematics. Note that although coordinated scheduling seems attractive because there is no wasted work from mis-speculation, the overhead of generating a dependence graph may limit its usefulness even when it is possible to produce one.

5. Case studies of algorithms

In this section, we discuss how the tao-analysis described in Section 3 and the implementation strategies described in Section 4 can be applied to important algorithms. The discussion is organized by the type of the operator.

5.1 Morph algorithms

Morphs are the most complex variety of operators. Although they can be viewed abstractly as replacing sub-graphs with other graphs, it is more intuitive to classify them as follows.

- *Refinement:* A refinement operator makes the graph bigger by adding new nodes and edges, possibly removing a few nodes and edges in the process.

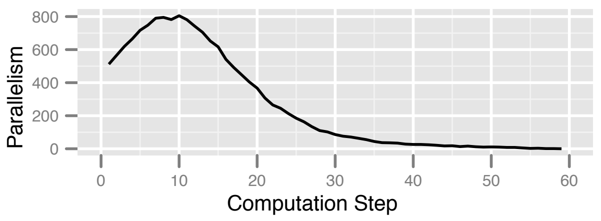


Figure 10. Available parallelism in Delaunay mesh refinement

- *Coarsening:* A coarsening operator clusters nodes or sub-graphs together, replacing them with a smaller sub-graph that represents the cluster.
- *General morph:* All other operations that modify the graph structure fall in this category.

5.1.1 Refinement

Data-driven refinement algorithms usually operate on a single graph. Most topology-driven refinement operators operate on two data structures G_i and G_o; G_i is read-only and its topology determines the active nodes in the algorithm, while G_o is morphed by each activity.

Map-reduce: In the map-reduce programming model [13], a map operation applies a function "point-wise" to each element of a set or multi-set S_i to produce another set or multi-set S_o. The active nodes are the elements of S_i, and they can be processed in any order. This is a topology-driven, unordered algorithm, which is a refinement morph for S_o since elements are added incrementally to S_o during execution.

Streams: A stream operator in languages like StreamIt [21] is a refinement morph from its input streams to its output streams (streams are non-strict sequences [53]). Stateless and stateful stream operators can be expressed using unordered and ordered iteration on sequences.

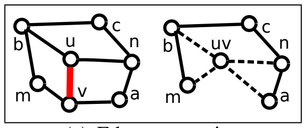
Prim's MST algorithm: Most algorithms that build trees in a top-down fashion use refinement morphs. Figure 11 shows one implementation of Prim's algorithm for computing MSTs; it is a topology-driven, ordered algorithm in which the operator is a reader for the graph g and a refinement morph for tree mst. Initially, one node is chosen as the root of the tree and added to the MST, and all of its edges are added to the ordered work-set, ordered by weight. In each iteration, the smallest edge (s, t) is removed from the work-set. Note that node s is guaranteed to be in the tree. If t is not in the MST, (s, t) is added to the tree, and all edges (t, u) are added to the work-set, provided u is not in the MST. When the algorithm terminates, all nodes are in the MST. This algorithm has the same asymptotic complexity as the standard algorithm in textbooks [12], but the work-set contains edges rather than nodes. The standard algorithm requires a priority queue in which node priorities can be decreased dynamically. It is unclear whether it is worth generalizing the implementation of ordered-set iterators in a general-purpose system like Galois to permit dynamic updates to the ordering.

N-body tree-building: Top-down tree construction is also used in n-body methods like Barnes-Hut [6] and fast multipole. The tree is a recursive spatial partitioning in which each leaf contains a single particle. Initially, the tree is a single node, representing the entire space. Particles are then inserted into the tree, splitting leaf nodes so that each leaf node contains only a single particle. The work-set in this case is the set of particles. In contrast to Prim's algorithm, this algorithm is unordered because particles can be inserted into the tree in any order.

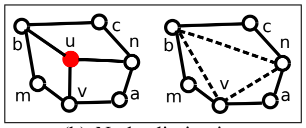
Andersen-style inclusion-based points-to analysis: This compiler algorithm [3] is an example of a refinement morph on general graphs. It builds a *points-to graph* in which nodes represent pro-

```
Graph g = // read in input graph
Tree mst; // create empty tree
mst.setRoot(r); // r is an arbitrary vertex in g
OrderedWorkset<Edge> ows; // ordered by edge weight
ows.add(g.getEdges(r));
foreach (Edge (s,t) in OrderedSet ows) {
  if (mst.contains(t)) continue;
  mst.addEdge(s,t); // s becomes parent of t
  foreach (Edge (t,u) in Set g.getEdges(t)) {
    if (!mst.contains(u)) ows.add((t,u));
  }
}
```

Figure 11. Top-down tree construction: Prim's MST algorithm



(a) Edge contraction



(b) Node elimination

Figure 12. Two kinds of coarsening operators

gram variables, and an edge (a, b) asserts that variable a may point to b at some point in the program. Each iteration of the algorithm may discover new points-to facts, and if it does, it adds new edges to the points-to graph. This is an unordered, data-driven algorithm that uses a refinement morph on the graph.

Delaunay mesh refinement: This is an unordered, data-driven morph algorithm on an unstructured graph (see Section 2).

Discussion: Refinement algorithms in which graph elements are only added and never removed are called *strong* refinement algorithms. All the algorithms listed above other than DMR are strong refinement algorithms. Section 6.3 describes how this property can be exploited to produce efficient implementations.

Although parallelism in most irregular algorithms is very input dependent, algorithms that use refinement morphs have a characteristic profile when executed with randomized input data. Figure 10 shows the parallelism profile for DMR for an input of 100,000 triangles. Available parallelism starts at some level, increases as the data structure gets larger (because there is less likelihood of conflicts), and then ramps down as work is completed. The parallelism profile for Prim's algorithm, shown in Figure 8(b), has a similar pattern—in this algorithm, the available parallelism is very small initially since only one activity succeeds in growing the tree from the root. As the MST gets larger, there are more sites where the tree can grow, so parallelism ramps up and then ramps down as the MST is completed. Tree-building in Barnes-Hut exhibits a similar parallelism profile, as can be seen in Figure 13(a).

5.1.2 Coarsening

There are three main ways of doing coarsening: edge contraction, node elimination and sub-graph contraction.

Edge contraction An edge is eliminated from the graph by fusing the two nodes at its end points and removing redundant edges from the resulting graph, as shown in Figure 12(a). When redundant edges are eliminated, the weight on the remaining edge is adjusted in application-specific ways. In sparse graphs, each edge contraction affects a relatively small neighborhood, so in sufficiently large graphs, many edge contraction operations can happen in parallel.

Boruvka's MST algorithm: Boruvka's algorithm [12] computes MSTs bottom-up by performing edge contraction on the input graph until there is only one node left. A separate graph represents the MST as it is built bottom-up. The MST is initialized with every node in the input graph forming its own single-node tree, i.e., a forest. The active nodes are the nodes remaining in the input graph. In each iteration, an active node n finds the minimum weight edge (n, m) incident on it and performs edge contraction along this edge. This edge is added to the MST where it connects two previously disjoint trees. A new node nm is created in the input graph to represent the merged nodes n and m. If there are redundant edges during edge contraction, only the least weight edge is kept. Finally, nm is added back to the work-set. When the input graph has been contracted to one node, the MST graph will be the correct minimal spanning tree of the input graph.

Figure 8(a) shows a parallelism profile for Boruvka's algorithm. Interestingly, Kruskal's algorithm [12] also finds MSTs by performing edge contraction, but it iterates over an ordered work-set of edges, sorted by edge weight.

Metis graph partitioner: Graph coarsening by edge contraction is a key step in the Metis graph partitioner [34]. This algorithm builds a sequence of successively coarser graphs until a coarse-enough graph is obtained; this graph is then partitioned, and the partitioning is interpolated back to the original graph. Each of the coarsening steps is performed by an unordered, topology-driven morph (the nodes of the finer graph are the active nodes).

The parallel implementation of Metis uses runtime coordinated scheduling. The problem of finding a maximal independent set of edges that can be contracted in parallel can be solved by finding *maximal matchings* [12]. Efficient heuristics are known for computing maximal cardinality matchings and maximal weighted matchings in graphs of various kinds. Metis uses randomized matching since that seems to perform well in practice; this strategy is well-suited for autonomous scheduling as well.

Agglomerative clustering and map/reduce: Agglomerative clustering, a well-known data-mining algorithm, also uses a coarsening morph. The input is (i) a data-set, and (ii) a measure of the distance between items in the data-set. At each step, the two closest points in the data-set are clustered together and replaced by a single new point that represents the new cluster. The location of this new point may be determined heuristically [59]. The greedy strict consensus merger (SCM) algorithm for phylogeny reconstruction implements this algorithm for a set of trees. These algorithms can be expressed using an ordered set iterator. The *reduce* operation in the map-reduce model [13] can be implemented in many ways; an in-place reduction can be implemented by an unordered iterator in which the operator replaces randomly chosen pairs of elements from the set with the result of applying the reduction operation to these elements.

Node elimination Graph coarsening can also be based on *node elimination*. Each step removes a node from the graph and inserts edges as needed between its erstwhile neighbors to make a clique, adjusting weights on the remaining nodes and edges appropriately. In Figure 12(b), node u is eliminated, and edges (b, v), (v, n) and (n, b) are inserted to make $\{b, v, n\}$ a clique. Node elimination is the graph-theoretic foundation of matrix factorization algorithms such as Cholesky and LU factorizations.

Sparse Cholesky factorization in particular has received a lot of attention in the numerical linear algebra community. The new edges inserted by node elimination are called *fill* since they correspond to zeroes in the original matrix that become non-zeros as a result of the elimination process. Different node elimination orders result in different amounts of fill in general; therefore, although the basic algorithm is unordered, practical implementations of sparse Cholesky factorization use heuristic orderings for node elimination to minimize fill. One heuristic is *minimal-degree ordering*, which greedily picks the node with minimal degree at each elimination step. Just-in-time coordination is used to schedule computations [20].

Sub-graph contraction Graphs can be coarsened by contracting entire sub-graphs at a time (edge contraction is a special but important sub-case). In the compiler literature, *elimination-based al-*

gorithms perform dataflow analysis on control-flow graphs by contracting "structured" sub-graphs whose dataflow behaviors have a concise description. Dataflow analysis is performed on the reduced graph, and within contracted sub-graphs, interpolation is used to determine dataflow values. This idea can be used recursively on the reduced graph. Sub-graph contraction can be performed in parallel. This approach to parallel dataflow analysis has been studied by Ryder [41] and Soffa [37].

5.1.3 General morph

Some applications make structural updates that are neither refinements nor coarsenings, but many of these updates may nevertheless be performed in parallel.

Some algorithms build trees using complicated structural manipulations that cannot be classified neatly as top-down or bottom-up construction. For example, when values are inserted into a heap or a red-black tree, the final data structure is a tree, but each insertion may perform complex manipulations [12]. The structure of the final heap or red-black tree depends on the order of insertions, but for most applications, any of these final structures is adequate, so this is an example of don't-care non-determinism, and it can be expressed with an unordered set iterator.

Algorithms that perform general morph operations on trees usually require optimistic parallelization. The simple conflict detection policy described in Section 4.1, which ensures disjointness of neighborhoods, is not adequate for these applications since every neighborhood contains the root of the tree. This class of algorithms may benefit from mechanisms like transactional memory that use more sophisticated conflict detection policies [29]; in fact, the parallelization of red-black tree operations is a standard microbenchmark in the literature on transactional memory.

Graph reduction of functional language programs and social network maintenance [10] are general morphs on graphs.

5.2 Local computation algorithms

We divide our discussion of local computation algorithms based on whether their active nodes are topology-driven or data-driven.

5.2.1 Topology-driven local computation algorithms

Cellular automata: Cellular automata operate on grids of one or two dimensions. Grid nodes represent cells of the automaton, and the state of a cell c at time t is a function of the states at time $t-1$ of cells in some neighborhood around c. This iterative scheme can be written using unordered iterators. A large variety of neighborhoods (stencils) are used in cellular automata.

Finite-differences: A similar state update scheme is used in finite-difference methods for the numerical solution of partial differential equations (PDEs) where it is known as Jacobi iteration. In this case, the grid arises from spatial discretization of the domain of the PDE, and nodes hold values of the dependent variable of the PDE. A disadvantage of Jacobi iteration is that it requires two arrays for its implementation to hold the states at the current and previous time steps. More complex update schemes have been designed to get around this problem. Intuitively, all these schemes blur the sharp distinction between old and new states, so nodes are updated using both old and new values. For example, *red-black ordering*, or more generally *multi-color ordering*, assigns a minimal number of colors to nodes in such a way that no node has the same color as the nodes in its neighborhood. Nodes of a given color therefore form an independent set that can be updated concurrently. These algorithms can be expressed using unordered set iterators with one loop for each color. Methods like Gauss-Seidel can be expressed using ordered set iterators.

In all these algorithms, active nodes and neighborhoods can be determined from the grid structure, which is known at compile-time. As a result, compile-time scheduling can be very effective for coordinating the computations. Most parallel implementations of Jacobi iteration partition the grid into blocks, and each processor is responsible for updating the nodes in one block. This data distribution requires less inter-processor communication than other distributions such as row or column cyclic distributions.

Tree traversals: Top-down and bottom-up are classic examples of tree traversals. The center-of-mass computation for cells in Barnes-Hut and other n-body methods are performed using a bottom-up walk over the spatial decomposition tree. This traversal can be expressed using an ordered iterator. For a tree, the amount of parallelism decreases as the computation advances up the tree, as can be seen in Figure 13(a).

Sparse MVM: The key operation in iterative linear system solvers like the conjugate-gradient method and GMRES is sparse matrix-vector multiplication (MVM), $y = Ax$, in which the matrix A is an $N \times N$ sparse matrix and x and y are dense vectors. Such a matrix can be viewed as a graph with N nodes in which there is a directed edge from node i to node j with weight A_{ij} if A_{ij} is non-zero. This algorithm can be expressed using unordered iteration over the nodes of the graph. The inspector-executor method is used to produce efficient communication schedules on distributed-memory machines, as discussed in Section 4.3. Compile-time coordination is possible on shared-memory machines since the operator is a reader for A and x, and it is trivial to determine that each iteration writes into a different element of y.

5.2.2 Data-driven local computation algorithms

In these algorithms, there is an initial set of active nodes, and performing an activity may cause other nodes to become active, so nodes become active in an unpredictable, data-driven fashion. Examples include the preflow-push algorithm [12] for the maxflow problem, AI message-passing algorithms such as belief propagation and survey propagation [44], Petri nets, and discrete-event simulation [33, 49]. In other algorithms, such as some approaches to solving spin Ising models [56], the pattern of node updates is determined by externally-generated data such as random numbers.

Although graph topology is typically less important for data-driven algorithms than it is in topology-driven ones, it can nevertheless be useful for reasoning about certain algorithms. For example, belief propagation is an exact inference algorithm on trees, but it is an approximate algorithm when used on general graphs because of the *loopy propagation problem* [44]. Grid structure is exploited in spin Ising solvers as well to reduce synchronization [56].

Preflow-push algorithm: This maxflow algorithm is the archetypal example of a data-driven local computation operator. Active nodes are nodes that have excess in-flow at intermediate stages of the algorithm. The algorithm also maintains at each node a value called *height*, which is a lower bound on the distance of that node to the sink. Two operations, *push* and *relabel*, are performed at active nodes to update the flow and height values respectively until termination. The algorithm is unordered.

Event-driven simulation: In the literature, there are two approaches to parallel event-driven simulation, called conservative and optimistic event-driven simulation [49]. Conservative event-driven simulation reformulates the algorithm so that in addition to sending data messages, processing stations also send out-of-band messages to update time at their neighbors. Each processing station can operate autonomously without fear of deadlock, and there is no need to maintain an ordered event list. This is an example of algorithm reformulation that replaces an ordered work-set with an unordered work-set. In contrast, Timewarp [33] is an optimistic parallel implementation of event-driven simulation. It can be viewed as an application-specific implementation of ordered set iterators as described in Section 4.1, with two key differences. First, threads are allowed to work on new events created by speculative activities that have not yet committed. This increases parallelism but opens up the possibility of cascading roll-backs. Second, instead of the global commit queue in the implementation of Section 4.1, there are periodic sweeps through the network to update "global virtual time" [33]. Both of these features can be implemented in a general-purpose way in a system like Galois, but more study is needed to determine if this is worthwhile.

5.2.3 Discussion

Solving fixpoint equations: Iterative methods for computing solutions to systems of equations can usually be formulated in both topology-driven and data-driven ways. A classic example is iterative dataflow analysis in compilers. Systems of dataflow equations can be solved by iterating over all the equations using a Jacobi or Gauss-Seidel formulation until a fixed point is reached; these are topology-driven approaches. Alternately, the classic worklist algorithm processes an equation only if its inputs have changed; this is a data-driven approach [1]. The underlying graph in this application is the control-flow graph, and for flow-sensitive problems, the dataflow values are implemented as labels on nodes or edges.

Speeding up local computation algorithms: Local computation algorithms can often be made more efficient by a preprocessing step that coarsens the graph, since this speeds up the flow of information across the graph. The multigrid method for solving linear systems is a classic example. This idea is also used in dataflow analysis of programs [1]. An elimination-based dataflow algorithm is used first to coarsen the graph. If the resulting graph is a single node, the solution to the dataflow problem is read off by inspection; otherwise, iterative dataflow analysis is applied to the coarse graph. In either case, the solution for the coarse graph is interpolated back into the original graph. One of the first hybrid dataflow algorithms along these lines was Allen and Cocke's *interval-based algorithm* for dataflow analysis. This algorithm finds and collapses intervals, which are single-entry, multiple-exit loop structures in the control-flow graph of the program.

Some local computation algorithms can be sped up by periodically computing and exploiting global information. The *global-relabel* heuristic in preflow-push is an example of such an operation. Preflow-push can be sped up by periodically performing a breadth-first search from the sink to update the height values [12]. In the extreme, global information can be used in every iteration. For example, iterative linear solvers often *precondition* the matrix iterations with another matrix that is obtained by a graph coarsening computation such as incomplete LU or Cholesky factorization.

5.3 Reader algorithms

An operator that reads a graph without modifying it in any way is a reader operator for that graph. Operators that perform *multiple* traversals over a fixed graph are a particularly important category. Each traversal maintains its own state, which is updated during the traversal, so the traversals can be performed concurrently. Force computation in n-body algorithms like Barnes-Hut [6] is an example. The force calculation is an unordered iteration over particles that computes the force on each particle by making a top-down traversal of a prefix of the tree. This step is completely parallel since each particle can be processed independently. The only shared data structure in this phase is the octree, which is not modified.

Another classic example is ray-tracing. Each ray is traced through the scene. If the ray hits an object, new rays may be spawned to account for reflections and refraction; these rays must also be processed. After the rays have been traced, they are used to construct the rendered image.

The key concern in implementing such algorithms is exploiting locality. One approach to enhancing locality in algorithms such as ray-tracing is to "chunk" similar work together. Rays that will propagate through the same portion of the scene are bundled together and processed simultaneously by a given processor. Because each ray requires the same scene data, this approach can enhance cache locality. A similar approach can be taken in Barnes-Hut: particles in the same region of space are likely to traverse similar parts of the octree during the force computation and thus can be processed together to improve locality [58].

In some cases, reader algorithms can be more substantially transformed to further enhance locality. In ray-tracing, bundled rays begin propagating through the scene in the same direction but may eventually diverge. Rather than using an *a priori* grouping of rays, groups can be dynamically updated as rays propagate through a scene, maintaining locality throughout execution [52].

5.4 Discussion

Some irregular applications have multiple phases, and each phase may use a different operator. N-body methods like Barnes-Hut are good examples; the tree-building phase uses a refinement morph whereas the force computation phase uses a reader operator since it does not modify the spatial decomposition tree. This is discussed in more detail in Section 6.1.

The structural analysis of algorithms presented in this section is different from the many efforts in the literature to identify *parallelism patterns*, such as the work of Mattson *et al.* [45], Snir's Parallel Processing Patterns [36] and the Berkeley motifs [51]. These approaches place related algorithms into categories (dense linear algebra, n-body methods, etc.), whereas this paper proposes structural decompositions of irregular algorithms. For example, n-body methods fall into a single category in the Berkeley motifs whereas we distinguish between the operators in the different phases of n-body methods. To use a biological metaphor, existing approaches produce classifications like the Linnaean taxonomy in biology, whereas the approach in this paper is more like molecular biology, producing "genetic profiles" of algorithms. On the other hand, existing classifications are broader in scope than ours since they seek to categorize implementation mechanisms such as whether task-queues or the master-worker approach is used to distribute work.

6. Case studies of applications

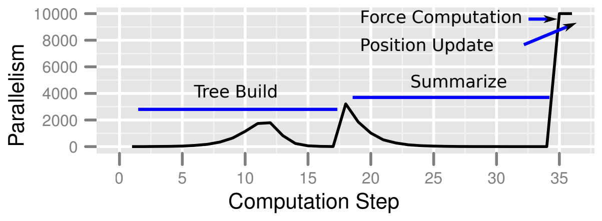
In this section, we present experimental results for three algorithms introduced previously: Barnes-Hut n-body simulation, Delaunay mesh refinement and inclusion-based points-to analysis. These algorithms were implemented using the Galois system, and illustrate how the principles of tao-analysis (in particular, Section 4.3) can be used to optimize performance.

We used two machines for our experimental results. The first contains two quad-core Intel Xeon X5570 processors for a total of eight cores. The second contains four six-core Intel Xeon X7540 processors for a total of 24 cores. Both run Linux 2.6.32.

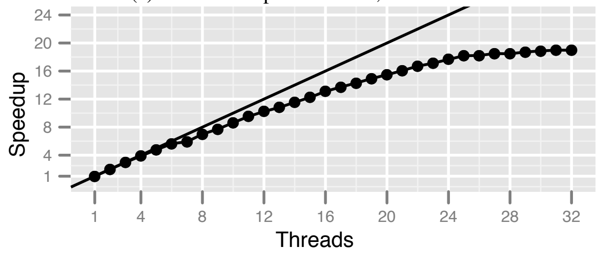
6.1 Barnes-Hut n-body simulation

Barnes-Hut shows how tao-analysis can be used to leverage compile-time optimizations even though the Galois system uses optimistic parallelization as its baseline. Figure 13(a) shows the parallelism profile for one time-step of the algorithm. There are four phases, each with a different structural classification.

1. *Tree build:* The first phase builds the spatial decomposition tree top-down by inserting particles into the tree and splitting nodes as needed so there is one particle per leaf node (topology: tree, operator: refinement morph, active nodes: topology-driven and unordered). Note that the parallelism profile is similar to the parallelism profiles of other refinement morphs shown in Figure 10.



(a) Parallelism profile for 10,000 bodies



(b) Speedup for 1,000,000 bodies; sequential runtime is 122.2 s.

Figure 13. Parallelism profile and speedup for Barnes-Hut. All inputs initialized using the Plummer model.

2. *Summarize:* The second phase computes the center-of-mass and total mass of each internal node using a bottom-up walk of the octree (tree, local computation, topology-driven and ordered).
3. *Force computation:* The third phase computes the force acting upon each particle by traversing parts of the octree (tree, reader, unordered).
4. *Force update:* The last phase moves each particle to a new position (set, local computation, topology-driven and unordered).

Figure 13(b) shows the speedup on the 24 core machine. Most of the execution time is spent in the force computation phase. By default, the Galois system would use speculative execution for this phase, but tao-analysis reveals that it is possible to use compile-time coordinated scheduling. Using this information, we can execute activities non-speculatively, taking advantage of reduced runtime overheads. The machine supports simultaneous multithreading (SMT), and we show a few data points with more than 24 threads, which correspond to adding SMT threads. Performance scales well up to 24 threads. After that point, each additional SMT thread improves performance slightly, which suggests that there is room for locality improvements.

6.2 Delaunay mesh refinement

Figure 14 shows results for Delaunay mesh refinement on the 24 core machine. We take advantage of the cautious property of the operator implementation to reduce some overheads of speculative execution. As before, we show data points for additional SMT threads as well. This application is floating-point intensive. Performance scales well up to 24 threads, but after that point, SMT threads share floating-point resources, which limits continued scalability.

6.3 Inclusion-based points-to analysis

Inclusion-based points-to analysis is an example of how properties of operators can be used to reduce synchronization overheads. The algorithm performs a context-insensitive, flow-insensitive analysis that determines the variables that a pointer variable might point to during the execution of a program. A pass through the program generates a system of set constraints that implicitly defines the points-to set for each variable in the program. These constraints can be represented as a graph in which nodes represent variables, and edges represent constraints. The process of solving these set

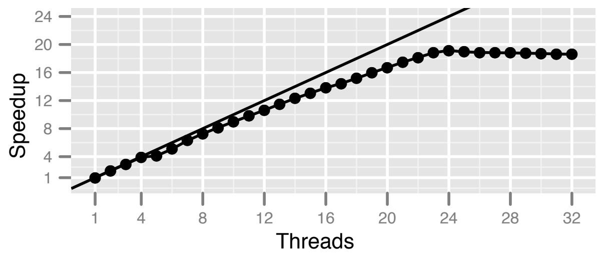
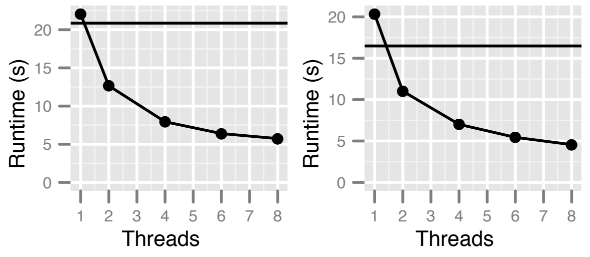


Figure 14. Speedup for Delaunay mesh refinement. The input is randomly generated. Initially, there are 1,999,998 triangles (951,964 bad triangles). Sequential runtime is 22.2 s.



(a) Gimp input (b) Mplayer input

Figure 15. Points-to analysis times. Horizontal line is performance of the reference implementation. Best speedup over reference for gimp and mplayer are 3.63 and 3.62 respectively.

constraints can be viewed in terms of the application of three graph rewrite rules to the constraint graph [46].

Each rewrite rule adds edges to the graph but does not remove existing nodes or edges, so the operator is a strong refinement morph. The baseline implementation described in Section 4.1 would acquire abstract locks on all the nodes participating in the rewrite rule. However, because the operator is a strong refinement morph, it is only necessary to acquire a lock on the node at which the new edge is added, reducing synchronization costs.

Figure 15 shows the performance of this approach on the eight core machine compared to a highly optimized sequential reference implementation written by Hardekopf [24]. The results are for two of the benchmarks from Hardekopf's suite. This is the first successful parallelization of Andersen-style points-to analysis.

7. Extensions to the amorphous data-parallelism model

In this section, we discuss two kinds of parallelism that are not exploited by the basic amorphous data-parallel execution model described in Section 4.1: nested amorphous data-parallelism and pipeline parallelism. We also discuss how the task-parallel execution model fits into our framework.

7.1 Nested amorphous data-parallelism

The pattern of parallelism described in this paper arises from applying an operator at multiple active nodes in a graph as shown in Figure 4. Since the neighborhood of each activity is some region of the graph, an activity can itself be executed in parallel; we call this parallelism *intra-operator parallelism* to distinguish it from the first variety of parallelism that we can call *inter-operator parallelism*. Inter-operator parallelism is the dominant parallelism pattern in problems for which neighborhoods are small compared to the overall graph since it is likely that most activities do not conflict; in these problems, intra-operator parallelism is usually fine-grain, instruction-level parallelism. Conversely, when each neighborhood is a large part of the graph, activities are likely to conflict and intra-operator parallelism may be more important. Inter/intra-operator parallelism is an example of *nested data-parallelism* [8].

The sparsity of the graph usually plays a major role in this balance between inter- and intra-operator parallelism. For many operators, neighborhoods include all the neighbors of the active node, so if the graph is very densely connected, a single neighborhood may encompass most of the graph and intra-operator parallelism is dominant. An extreme case is the factorization of dense matrices: the underlying graph is a clique, and each factorization step updates the entire graph, as explained in Section 5.1.2, so the only parallelism is intra-operator parallelism. In factorizing sparse matrices, inter-operator parallelism dominates for the first few steps; after some number of coarsening steps, the residual graph becomes dense enough that most high-performance sparse matrix codes switch to dense matrix techniques to exploit intra-operator parallelism [20].

7.2 Pipeline parallelism

In some applications, a data structure is produced incrementally by a computation called the *producer* and read incrementally by another computation called the *consumer*. If it is possible to overlap the executions of the producer and consumer, the resulting parallelism is called pipeline parallelism. The key problem in exploiting pipeline parallelism is determining when the producer has completed all the modifications it will ever make to an element the consumer wants to read. This problem is undecidable in general, so one approach to exploiting pipeline parallelism is to execute the consumer speculatively in parallel with the producer, rolling the consumer back if the producer overwrites data read by the consumer. This approach is unlikely to be practical since the consumer cannot commit any work until the producer has finished execution.

Pipeline parallelism becomes practical if the producer is a strong refinement morph since in that case, each iteration of the producer may add new elements to the graph but it does not modify elements that have already been produced. As long as the consumer does not test for the absence of an element, the producer and consumer can be executed safely in parallel. This amorphous data-parallel view of pipeline parallelism generalizes the standard view of pipeline parallelism in which the shared data structure is restricted to be a stream (sequence), as in StreamIt [21]; for example, we can pipeline the execution of a client of points-to analysis information with the execution of the points-to analysis algorithm described in Section 6.3 since the operator is a refinement morph.

7.3 Task parallel execution

Task parallel execution can be used to exploit amorphous data-parallelism by using a divide-and-conquer formulation of algorithms rather than an iterative formulation. Divide-and-conquer algorithms are naturally data-centric since they partition the data structure and execute the algorithm recursively on each partition; intuitively, the division step is a way of clustering active nodes based on their location in the data structure. If all data dependences are subsumed by call/return control dependences, the recursive calls can be executed safely in parallel. The Cilk project has explored this approach to exploiting parallelism [9].

This approach obviously requires an efficient partitioner for the data structure. Partitioning is straightforward for structured and semi-structured topologies, so most task-parallelism studies have focused on algorithms that deal with sets, sequences (arrays), cliques (dense matrices), grids and trees. Partitioning general graphs is more difficult, and for many algorithms, the partitioning step can be more expensive than the algorithm itself. Furthermore, ordered algorithms like event-driven simulation do not lend themselves to a natural divide-and-conquer formulation. Therefore, our view of task parallel execution is that it is a way of exploiting amorphous data-parallelism in unordered algorithms that deal with structured and semi-structured topologies: at each stage of the division process, activities whose neighborhoods lie entirely within a partition are executed during the processing of that partition while activities whose neighborhoods span partitions at that level can be executed when returning from the recursive calls.

8. Conclusions

Dependence graphs have been used for more than thirty years by the parallel programming community to reason about parallelism in algorithms. In this paper, we argued that this program-centric abstraction is inadequate for most irregular algorithms. To address this problem, we proposed a data-centric formulation of algorithms called the operator formulation, which led to a structural analysis of algorithms called tao-analysis. Tao-analysis reveals that a generalized form of data-parallelism called amorphous data-parallelism is ubiquitous in algorithms, and that depending on the tao-structure of the algorithm, this parallelism may be exploited by compile-time, inspector-executor or optimistic parallelization, thereby unifying these seemingly unrelated parallelization techniques. Parallelization of regular algorithms is just one special case, and tao-analysis allows many application-specific optimizations to be generalized. An extensive survey of key algorithms and experimental results from three applications provided evidence for these claims.

Parallel programming today is burdened by abstractions like dependence graphs that are inadequate for many algorithms and by a plethora of application-specific optimizations and isolated mechanisms. In many ways, this state of the art resembles alchemy rather than a science like chemistry. To move forward, we need a systematic way of understanding parallelism and locality—a way to generalize from specific cases and a way to apply generalizations to specific cases. In short, we need a science of parallel programming. We believe that the operator formulation and tao-analysis of algorithms are key elements of such a science.

साकारमनृतं विद्धि निराकारं तु निश्चलं ।
एतत्तत्त्वोपदेशेन न पुनर्भवसंभवः ॥

That which has form is not real; only the amorphous endures.
When you understand this, you will not return to illusion.

— Ashtavakra Gita(1:18)

References

[1] A. Aho, R. Sethi, , and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *LCPC*, 2003.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] Arvind and R.S.Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers*, 39(3), 1990.

[5] D. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[6] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4), December 1986.

[7] D. K. Blandford, G. E. Blelloch, and C. Kadow. Engineering a compact parallel Delaunay algorithm in 3D. In *Symposium on Computational Geometry*, pages 292–300, 2006.

[8] G. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[10] U. Brandes and T. Erlebach, editors. *Network Analysis: Methodological Foundations*. Springer-Verlag, 2005.

[11] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *SCG*, 1993.

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

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[14] B. Delaunay. Sur la sphere vide. *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk,*, 7:793–800, 1934.

[15] J. Dennis. Dataflow ideas for supercomputers. In *CompCon*, 1984.

[16] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[17] P. Diniz and M. Rinard. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM TOPLAS*, 19(6), 1997.

[18] H. Ehrig and M. Löwe. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science*, 109:123–143, 1993.

[19] R. Ghiya and L. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[20] J. R. Gilbert and R. Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[21] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[22] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *OOPSLA*, 2005.

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

[24] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, 2007.

[25] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[26] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *PPoPP*, 2011.

[27] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE TPDS*, 1(1):35–47, January 1990.

[28] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[29] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[30] S. Horwitz, P. Pfieffer, and T. Reps. Dependence analysis for pointer variables. In *PLDI*, 1989.

[31] B. Hudson, G. L. Miller, and T. Phillips. Sparse parallel Delaunay mesh refinement. In *SPAA*, 2007.

[32] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[33] D. R. Jefferson. Virtual time. *ACM TOPLAS*, 7(3), 1985.

[34] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *JPDC*, 48(1):96–129, 1998.

[35] K. Kennedy and J. Allen, editors. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2001.

[36] F. Kjolstad and M. Snir. Ghost cell pattern. In *Workshop on Parallel Programming Patterns*, 2010.

[37] R. Kramer, R. Gupta, and M. L. Soffa. The combining DAG: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8), August 1994.

[38] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. How much parallelism is there in irregular applications? In *PPoPP*, 2009.

[39] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *PLDI*, 2011.

[40] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[41] Y. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In *Supercomputing*, pages 236–247, 1992.

[42] M. Lowe and H. Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In *Workshop on Graph-theoretic concepts in computer science*, 1991.

[43] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15, 1986.

[44] D. Mackay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[45] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Publishers, 2004.

[46] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel Anderson-style points-to analysis. In *OOPSLA*, 2010.

[47] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP*, 2010.

[48] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM TOPLAS*, March 2003.

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

[50] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS*, 2011.

[51] D. Patterson, K. Keutzer, K. Asanovica, K. Yelick, and R. Bodik. Berkeley dwarfs. `http://view.eecs.berkeley.edu/`.

[52] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH*, 1997.

[53] K. Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM Trans. Program. Lang. Syst.*, 7, April 1985.

[54] D. Prountzos, R. Manevich, K. Pingali, and K. McKinley. A shape analysis for optimizing parallel graph programs. In *POPL*, 2011.

[55] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[56] E. E. Santos, S. Feng, and J. M. Rickman. Efficient parallel algorithms for 2-dimensional Ising spin models. In *IPDPS*, 2002.

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

[58] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal Of Parallel and Distributed Computing*, 27, 1995.

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

[60] C. Verbrugge. *A Parallel Solution Strategy for Irregular, Dynamic Problems*. PhD thesis, McGill University, 2006.

[61] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.

[62] U. Vishkin *et al*. Explicit multi-threading (xmt) bridging models for instruction parallelism. In *SPAA*, 1998.

[63] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[64] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44, 1995.