# Amorphous Data-parallelism in Irregular Algorithms [*]

Keshav Pingali[1,2], Milind Kulkarni[2], Donald Nguyen[1],
Martin Burtscher[2], Mario Mendez-Lojo[2], Dimitrios Prountzos[1], Xin Sui[3], Zifei Zhong[1]

[1]Department of Computer Science, [2]Institute for Computational Engineering and Sciences and [3]Department of Electrical and Computer Engineering
The University of Texas at Austin

## Abstract

Most client-side applications running on multicore processors are likely to be irregular programs that deal with complex, pointer-based data structures such as large sparse graphs and trees. However, we understand very little about the nature of parallelism in irregular algorithms, let alone how to exploit it effectively on multicore processors.

In this paper, we show that, although the behavior of irregular algorithms can be very complex, many of them have a generalized data-parallelism that we call amorphous data-parallelism. The algorithms in our study come from a variety of important disciplines such as data-mining, AI, compilers, networks, and scientific computing. We also argue that these algorithms can be divided naturally into a small number of categories, and that this categorization provides a lot of insight into their behavior. Finally, we discuss how these insights should guide programming language support and parallel system implementation for irregular algorithms.

## 1. Introduction

*Science is organized knowledge.* — Immanuel Kant

The parallel programming community has a deep understanding of parallelism and locality in dense matrix algorithms. However, outside of computational science, most algorithms are *irregular*: that is, they are organized around pointer-based data structures such as trees and graphs, not dense arrays. Figure 1 is a list of algorithms from important problem domains. Only finite-difference codes use dense matrices; the rest use trees and sparse graphs. Unfortunately, we currently have few insights into parallelism and locality in irregular algorithms, and this has stunted the development of techniques and tools that make it easier to produce parallel implementations of these algorithms.

Domain specialists have written parallel programs for some of the algorithms in Figure 1 (see [6, 22, 31, 32, 45, 52, 55] among others). There are also parallel graph libraries such as Boost [1] and Stapl [19]. These efforts are mostly problem-specific, and it is difficult to extract broadly applicable abstractions, principles, and mechanisms from these implementations. Automatic parallelization using points-to analysis [30] and shape analysis [21, 25] has been successful for irregular programs that use trees, such as n-body methods. However, most applications in Figure 1 are organized around large sparse graphs with no particular structure.

These difficulties have seemed insurmountable, so irregular algorithms remain the Cinderella of parallel programming in spite of their increasingly important role in applications. We believe that a new strategy is required to circumvent these difficulties: instead of continuing to investigate the low-level details of pointer manipulations in irregular programs, we must study the patterns of parallelism and locality in irregular algorithms and use these insights to guide the development of language, compiler, and runtime systems support for irregular programs. In this paper, we focus on *iterative* irregular algorithms, a category that includes all the algorithms in Figure 1. We exclude divide-and-conquer algorithms from our study.

| Application/domain | Algorithms |
|---|---|
| Data-mining | Agglomerative clustering [59], k-means [59] |
| Bayesian inference | Belief propagation [43], Survey propagation [43] |
| Compilers | Iterative dataflow algorithms [3], elimination-based algorithms [3] |
| Functional interpreters | Graph reduction [48], static and dynamic dataflow [5, 14] |
| Maxflow | Preflow-push [12], augmenting paths [12] |
| Minimal spanning trees | Prim's [12], Kruskal's [12], Boruvka's [17] algorithms |
| N-body methods | Barnes-Hut [8], fast multipole [24] |
| Graphics | Ray-tracing [20] |
| Network science | Social network maintenance [28] |
| System modeling | Petri-net simulation [47] |
| Event-driven simulation | Chandy-Misra-Bryant algorithm [45], Time-warp algorithm [32] |
| Meshing | Delaunay mesh generation [10], refinement [55], Metis-style graph partitioning [33] |
| Linear solvers | Sparse MVM [23], sparse Cholesky factorization [22] |
| PDE solvers | Finite-difference codes [23] |

**Figure 1.** Sparse graph algorithms from different domains

This paper makes the following contributions.

- We define a generalized data-parallelism, which we call *amorphous data-parallelism*, and argue that it is ubiquitous in irregular algorithms. Data-parallelism, defined by Hillis and Steele as "parallelism [that] comes from simultaneous operations across large sets of data" [29], is the most important kind of parallelism in dense matrix algorithms. Unlike data-parallelism in regular algorithms, which is formulated in terms of distances and directions in iteration spaces [34], amorphous data-parallelism is defined in a data-centric way in terms of neighborhoods of data structure elements.
- We show that irregular algorithms such as the ones listed in Figure 1 can be organized into a small number of categories, and we argue that these categories provide insight into the patterns of amorphous data-parallelism in these algorithms.
- We discuss programming language abstractions and systems support for making it easier to produce parallel implementations of irregular algorithms. These abstractions are similar in

[*] This work is supported in part by NSF grants 0719966, 0702353, 0615240, 0541193, 0509307, 0509324, 0426787 and 0406380, as well as grants from IBM and Intel Corporation.

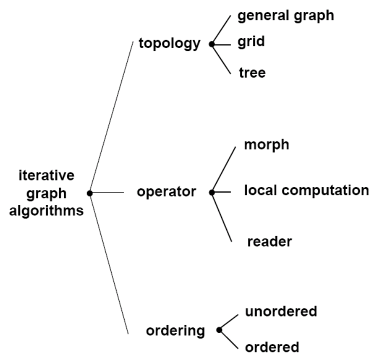


**Figure 2.** Iterative graph algorithms

spirit to those in the relational database model in which the complexities of meta-data such as B-trees and other pointer-based index structures are hidden from programmers by providing them with the abstraction of relations, which are just tabular data. The Google map-reduce model takes a similar approach [13].

Without loss of generality, the discussion in this paper will be framed largely in terms of graphs and graph algorithms. Trees are obviously a special case of graphs. An extreme case of a sparse graph is a graph with some number of nodes and no edges; if nodes are labeled with values, this structure is isomorphic to a set or multiset. At the other extreme, we have cliques, which are graphs that have an edge between every pair of nodes; these are isomorphic to square dense matrices.

The rest of this paper is organized as follows. In Section 2, we introduce high-level abstractions for thinking about irregular algorithms and show how these abstractions permit irregular algorithms to be organized into a small number of categories. In Section 3, we introduce the concept of amorphous data-parallelism and show how it might be exploited. In Section 3.1, we describe a baseline execution model based on transactional semantics that exploits amorphous data-parallelism. In Section 3.2, we describe how algorithmic structure can be exploited to reduce execution overheads. Sections 4–6 discuss different categories of irregular algorithms in more detail and show how amorphous data-parallelism arises in these algorithms. Finally, Section 7 discusses how these results might guide future research in systems support for irregular programs.

## 2. Iterative graph algorithms

This section describes a framework for thinking about iterative graph algorithms, which is shown in Figure 2. It also introduces the notions of *active nodes*, *neighborhoods*, and *ordering*, which are central to the rest of the paper.

### 2.1 Graphs and graph topology

We use the term *graph* to refer to the usual directed graph *abstract data type* (ADT) that is formulated in terms of (i) a set of nodes $V$, and (ii) a set of edges ($\subseteq V \times V$) between these nodes. Undirected edges are modeled by a pair of directed edges in the standard fashion. In some of our applications, nodes and edges are labeled with values. In this paper, all graph algorithms will be written in terms of this ADT.

In some algorithms, the graphs have a special structure that can be exploited by the implementation. *Trees* and *grids* are particularly important. Grids are usually represented using dense arrays, but note that a grid is actually a very sparse graph (a 2D grid point that is not on the boundary is connected to four neighbors).

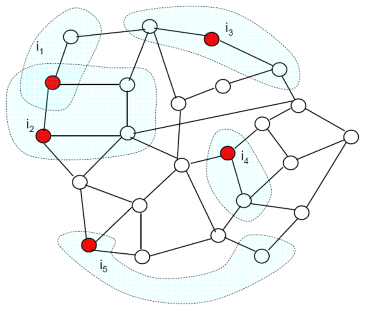


**Figure 3.** Active elements and neighborhoods

We will not discuss a particular *concrete representation* for the graph ADT. In the spirit of the relational database model, an implementation is free to choose a concrete representation that is best suited for a particular algorithm and machine: for example, grids and cliques may be represented using dense arrays, while sparse graphs may be represented using adjacency lists. For general graphs, our implementation uses a logically partitioned graph representation in which each processor or core is assigned one or more partitions; the metadata is also partitioned so a processor can read/write data in its own partition without coordinating with other processors.

### 2.2 Active elements and neighborhoods

At each point during the execution of a graph algorithm, there are certain nodes or edges in the graph where computation might be performed. Performing a computation may require reading or writing other nodes and edges in the graph. The node or edge on which a computation is centered will be called an *active element*, and the computation itself will be called an *activity*. To keep the discussion simple, we assume from here on that active elements are nodes. Borrowing terminology from the literature on cellular automata [56], we refer to the set of nodes and edges that are read or written in performing the computation as the *neighborhood* of that active node. Figure 3 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 the set of its neighbors in the graph.

It is convenient to think of an activity as the application of an *operator* to the neighborhood of an active element. It is useful to distinguish between three kinds of operators:

- *Morph:* A morph operator may modify the structure of the neighborhood by adding or deleting nodes and edges, and may also update values on nodes and edges.
- *Local computation:* A local computation operator may update values stored on nodes or edges in the neighborhood, but does not modify the graph structure.
- *Reader:* A reader does not modify the neighborhood in any way.

We illustrate these concepts using a few algorithms from Figure 1. The preflow-push algorithm [12] is a maxflow algorithm in directed graphs, described in detail in Section 5.2. During the execution of this algorithm, some nodes can temporarily have more flow coming into them than is going out. These are the active nodes (in fact, we borrowed the term "active node" from the literature on preflow-push algorithms). The algorithm repeatedly selects an active node $n$ and tries to increase the flow along some outgoing edge ($n \rightarrow m$) to eliminate the excess flow at $n$ if possible; if it succeeds, the residual capacity on that edge is updated. Therefore, the neighborhood consists of $n$ and all of its outgoing edges, and

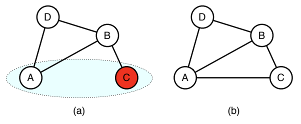


**Figure 4.** Forming a new link in a social network

the operator is a local computation operator. Increasing the flow to $m$ may cause $m$ to become active; if $n$ still has excess in-flow, it remains an active node.

A second example arises in algorithms that examine social networking graphs to determine communities [28]. Nodes in the social network represent people, and edges connect people who have communicated directly. Consider the social network in Figure 4(a). If person A sends an email to person B, who then forwards that email to person C, person C can now send an email directly to person A, creating a new link in the graph, as seen in Figure 4(b). This is an example of a morph operator where the neighborhood is a disconnected portion of the graph.

The final example is event-driven simulation. In this application, nodes in the graph represent processing stations, and edges represent channels along which nodes exchange messages with timestamps. A processing station consumes incoming messages in time order, and for each message it consumes, it performs some processing that may change its internal state and then produces zero or more messages on its outgoing edges. In this application, active nodes are nodes that have messages on their input channels, and the operator is a local computation operator.

Some algorithms may exhibit *phase* behavior: they use different types of operators at different points in their execution. Barnes-Hut [8] is an example: the tree-building phase uses a morph, as explained in Section 4.1, and the force computation uses a reader, as explained in Section 6.

## 2.3 Ordering

In general, there are many active nodes in a graph, so a sequential implementation must pick one of them and perform the appropriate computation.

In some algorithms, the implementation is allowed to pick *any* active node for execution. This is an example of Dijkstra's *don't-care non-determinism* [15], also known as *committed-choice non-determinism*. For some of these algorithms, the output is independent of these implementation choices. This is referred to as the Church-Rosser property, since the most famous example of this behavior is $\beta$-reduction in $\lambda$-calculus [7]. Dataflow graph execution [5, 14] and the preflow-push algorithm [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, described in more detail in Section 4.1, and Petri nets [47] are examples. The final mesh produced by Delaunay mesh refinement depends on the order in which badly-shaped triangles are chosen for refinement, but all outcomes are acceptable (note that the preflow-push algorithm exhibits this kind of behavior if the algorithm outputs the minimal cut as well as the maximal flow).

In contrast, some algorithms dictate an order in which active nodes must be processed. 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 sometimes be a partial order.

## 2.4 Programming notation

A natural way to program these algorithms is to use *worklists* to keep track of active nodes. Processing an item from a worklist can create new work, so it must be possible to add items to a worklist while processing other items of that worklist.

When active nodes are not ordered, the worklist is conceptually an unordered *set*, and the algorithm iterates over this set of active nodes in some order, performing computations. Since operators may create new active nodes, it must be possible to add new elements to the set while iterating over it, as is done in Galois set iterators [36]. In a sequential implementation, one concrete representation for a worklist is a linked-list.

For algorithms in which active nodes are ordered, we use an ordered set iterator that iterates in set order over the set of active nodes. As before, it must be possible to add elements to the ordered set while iterating over it. In this case, the worklist is a priority queue, and one concrete representation is a heap [12].

In describing algorithms, we will use the following notation.

DEFINITION 1. *We refer to the following two constructs as* unordered and ordered Galois set iterators *respectively.*

- **for each Element e in Set S { B(e) }**
  The loop body B(e) is executed for each element e of Set S. Since set elements are not ordered, this construct asserts that in a serial execution of the loop, the iterations can be executed in any order. There may be dependences between the iterations, but any serial order of executing iterations is permitted. When an iteration executes, it may add elements to S.
- **for each Element e in OrderedSet S { B(e) }**
  This construct iterates over an ordered set S of active elements. It is similar to the Set iterator above, except that the execution order must respect the order on elements of S.

Note that these iterators have a well-defined sequential semantics.

## 2.5 Discussion

The metaphor of operators acting on neighborhoods is reminiscent of notions in term-rewriting systems, and graph grammars in particular [16, 42]. The semantics of functional language programs are usually specified using term rewriting systems (TRS) 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 [16, 42]. A graph rewrite rule is defined as a morphism in the category **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 as shown in Figure 5 [42]. 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:L $\rightarrow$ R at a redex m:L $\rightarrow$ G leads to a direct derivation (r,m):G $\Rightarrow$ H given by the pushout in Figure 5. The framework presented in this section is more general because the behavior of operators is not limited to "syntactic" rewrite rules; for example, it is not clear that Delaunay mesh refinement, described in Section 4.1, can be specified using graph grammars.

## 3. Amorphous data-parallelism

Figure 3 shows intuitively how opportunities for exploiting parallelism arise in graph algorithms. We first consider the case when active nodes are not ordered. The neighborhoods of activities $i_3$

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

**Figure 5.** Graph rewriting: single-pushout approach

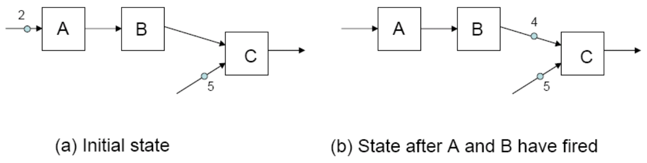


**Figure 6.** Conflicts in event-driven simulation

and $i_4$ are disjoint, so these activities can be executed concurrently (subject to constraints imposed by the concrete representation of the graph). The neighborhoods of $i_1$ and $i_2$ overlap, but nevertheless, they can be performed concurrently if elements in the intersection of the two neighborhoods are not modified by either activity. These examples suggest the following definition.

DEFINITION 2. *Activities $i_1$ and $i_2$ are said to* conflict *if there is an element in the intersection of their neighborhoods that is modified by one of the activities.*

If active nodes are not ordered, activities that do not conflict can be performed in parallel. In general, the only way to determine neighborhoods is to execute the activities, so it is necessary to use optimistic or speculative parallel execution.

The situation is more complex if active nodes are ordered. Consider the event-driven simulation shown in Figure 6. Suppose that in a sequential implementation, node A fires and produces a message with time-stamp 3, and then node B fires and produces a message with time-stamp 4. Notice that node C must consume the message with time-stamp 4 *before* it consumes the message with time-stamp 5. In the parallel context, we see that the neighborhoods for the activities at nodes A and C are disjoint, so these activities do not conflict according to Definition 2, but it is obvious that they cannot be executed concurrently without violating the sequential semantics of the program. This example shows the subtlety of exploiting parallelism when active nodes are ordered, and nodes can become active dynamically: the flap of a butterfly's wings in Brazil can set off a tornado in Texas [41]. However, notice that if the message from B to C had a time-stamp greater than 5, it would have been legal to execute the activities at nodes A and C in parallel! The solution is to adopt the approach used in out-of-order processors to exploit instruction-level parallelism [26]: when active nodes are ordered, activities can be executed speculatively in parallel, but they must commit in order, which ensures that the sequential semantics of the program is respected while exploiting potential parallelism.

DEFINITION 3. *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 standard 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.

### 3.1 Baseline execution model

Amorphous data-parallelism can be exploited by using a system like the Galois system [36]. A master thread begins executing the program. When this thread encounters a Galois set iterator, it enlists the assistance of some number of worker threads to execute iterations concurrently with itself. Threads grab active nodes from the workset, and speculatively execute iterations, making back-up copies of all modified objects to permit rollback if a conflict is detected. Conflict detection and rollback is implemented by the runtime system using a variation of software transactional memory [27] that exploits commutativity of abstract data type operations [36]. Intuitively, each node and edge in the graph has an associated logical lock that must be acquired by a thread to operate on that element. If a thread tries to acquire a lock that is already owned by another thread, a conflict is reported, and the runtime system takes appropriate action for recovery. Otherwise, locks are released when the iteration terminates. For ordered set iterators, the runtime system ensures that the iterations commit in the set order. All threads are synchronized using barrier synchronization at the end of the iterator.

We have built a tool called ParaMeter, implemented on top of the Galois system, that can provide estimates of the amount of amorphous data-parallelism in irregular algorithms [37]. For algorithms in which active nodes are not ordered, it repeatedly finds a maximal independent set of activities, and executes all these activities in a single step. The number of activities processed in each step is therefore an abstract measure of the parallelism in the application, and is called the *available parallelism*. For ordered work-sets, the time step when an activity commits may be different from when it is executed. Parameter assign the activity to the time step in which it executed, provided it ultimately commits. ParaMeter generates *parallelism profiles*, which are graphs that show how the number of independent activities varies with the time step. They can provide substantial insight into the nature of amorphous data-parallelism, as we discuss later in this paper.

### 3.2 Exploiting structure to optimize execution

In practice, handwritten parallel implementations of irregular algorithms rarely use the complex machinery described in Section 3.1 for supporting optimistic parallel execution (the Timewarp algorithm [32] for discrete-event simulation is probably the only exception to this rule). This is because most algorithms have a lot of structure that can be exploited to avoid using many of these mechanisms. As we will see, the classification introduced in Section 2 provides the right framework for describing structure.

DEFINITION 4. *A* cautious operator *is an operator that reads all the elements in its neighborhood before it modifies any element in its neighborhood.*

Cautious operators have the useful property that the neighborhood of an active node can be determined by executing the operator up to the point where it starts to make modifications to the neighborhood. Therefore, the neighborhoods of all active nodes can be determined before any modifications are made to the graph.

#### 3.2.1 Zero-copy implementation

For algorithms in which (i) the operator is cautious, and (ii) active elements are unordered, optimistic parallel execution can be implemented without making data copies for recovery in case of conflicts. Most of the algorithms with unordered active elements discussed in this paper fall in this category.

An efficient implementation of such algorithms is the following. Logical locks are associated with nodes and edges and are acquired during the read phase of the operator. If a lock cannot be acquired,

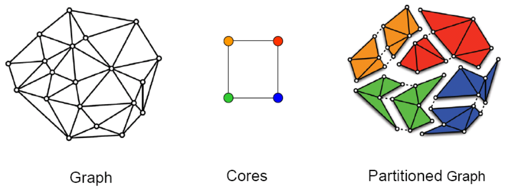


**Figure 7.** Logical partitioning

there is a conflict and the computation is rolled back simply by releasing all locks acquired up to that point; otherwise, all locks are released when the computation terminates. We will refer to this as a *zero-copy* implementation.

Zero-copy implementations should not be confused with two-phase locking, since under two-phase locking, updates to locked objects can be interleaved arbitrarily with acquiring locks on new objects. Strict two-phase locking eliminates the possibility of cascading rollbacks, while cautious operators are more restrictive and permit the implementation of optimistic parallel execution without the need for data copying. Notice that if active elements are ordered, data copying is needed even if the operator is cautious since conflicting active elements may be created dynamically, as discussed earlier using Figure 6 as an example.

### 3.2.2 Data-partitioning and lock coarsening

In the baseline implementation, the workset can become a bottleneck if there are a lot of worker threads since each thread must access the workset repeatedly to obtain work. For algorithms in which (i) the data structure topology is a general graph or grid (*i.e.*, not a tree), and (ii) active elements are unordered, data partitioning can be used to eliminate this bottleneck and to reduce the overhead of fine-grain locking of graph elements. The graph or grid can be partitioned *logically* between the cores: conceptually, each core is given a color, and a contiguous region of the graph or grid is assigned that color. Instead of associating locks with nodes and edges, we associate locks with regions; to access an element in a region, a core needs to acquire the lock associated with that region. Instead of a centralized workset, we can implement one workset per region, and assign work in a data-centric way: if an active element lies in region R, it is put on the workset for region R, and it will be processed by the core associated with that region. If an active element is not near a region boundary, its neighborhood is likely to be confined to that region, so for the most part, cores can compute independently. To keep core utilization high, it is desirable to over-decompose the graph or grid into more regions than there are cores to increase the likelihood that a core has work to do even if some of its regions are locked by other cores.

For algorithms like Delaunay mesh refinement in which (i) the data structure topology is a general graph or grid, (ii) the operator is cautious, and (iii) active elements are unordered, an implementation can use data-partitioning, lock-coarsening, and a zero-copy implementation. This can lead to extremely efficient implementations [38].

One disadvantage of this approach is load imbalance since it is difficult to partition the graph so that computational load is evenly distributed across the partitions, particularly because active nodes can be created dynamically. Application-specific graph partitioners might be useful; dynamic load-balancing using a strategy like work-stealing [2] is another possibility.

### 3.2.3 Scheduling

In the baseline implementation, there is no coordination between the computations at different active nodes. We call this *autonomous scheduling* for obvious reasons. For many algorithms, it is possible

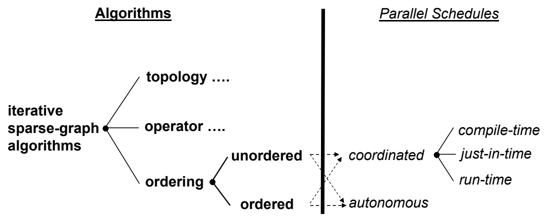


**Figure 8.** Scheduling strategies

to produce *coordinated* schedules that exploit parallelism without using dynamic scheduling of iterations. Figure 8 shows a number of coordinated scheduling strategies that differ mainly in the binding time of the scheduling decisions.

- *Run-time coordination:* For algorithms in which (i) the operator is cautious and (ii) active nodes are unordered, a bulk-synchronous execution strategy is possible [57]. The algorithm is executed over several steps. In each step, (i) the neighborhoods of all active nodes are determined by partially executing the operator, (ii) an interference graph for the active nodes is constructed, (iii) a maximal independent set of nodes in the interference graph is determined, and (iv) the set of independent active nodes is executed without synchronization. This approach has been used by Gary Miller's group to parallelize Delaunay mesh refinement [31]. It can be used even if neighborhoods cannot be determined exactly, provided we can compute over-approximations to neighborhoods.
- *Just-in-time coordination:* A more efficient version of run-time coordination is possible if the algorithm satisfies certain additional properties. In many algorithms, the graph computation is inside a time loop, so it has to be performed several times, and it may be possible to compute the schedule for active nodes just once. Sufficient conditions are (i) the operator performs local computation, (ii) active nodes are unordered, and (iii) active nodes and neighborhoods depend only on the structure of the graph and can be determined once the graph structure is available. This is essentially the *inspector-executor* approach used in sparse iterative solvers [61].
- *Compile-time coordination:* A more efficient version of just-in-time coordination is possible if the graph structure is known at compile-time, as is the case with grids. In this case, the schedule can be determined completely at compile-time. This approach is used in stencil computations (the graph is a grid) and dense linear algebra computations (the graph is a clique).

## 3.3 Discussion

It is possible to define conflicts more aggressively than is done in Definition 2. For example, the neighborhoods of two activities may have a node in common, but if the node has multiple variables and the activities write to different variables in that node, the activities can be performed in parallel. This situation arises in Jacobi iteration, discussed in Section 5, in which each node has two variables called *old* and $new$. Even if both activities write to the same variable in a node, it may be possible to execute them concurrently if the updates to that variable commute, as is the case with reductions. To keep the discussion simple, we have not considered these variations.

The pattern of parallelism described in this section can be called *inter-operator* parallelism because it arises from applying an operator at multiple sites in the graph. There are also opportunities to exploit parallelism *within* a single application of an operator that we call *intra-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 activi-

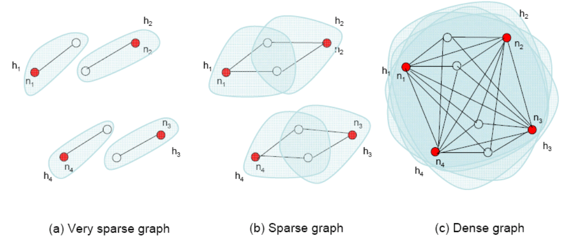


**Figure 9.** Sparsity and inter/intra-operator parallelism

ties do not interfere; in these problems, intra-operator parallelism is usually fine-grain, instruction-level parallelism. Conversely, when neighborhoods are large, activities are likely to conflict and intra-operator parallelism may be more important. Both the operator and the sparsity of the graph play a role. Figure 9 shows a series of graphs with the same number of nodes but with increasing density from left to right. If the operator updates values at the neighbors of active nodes, inter-operator parallelism decreases and intra-operator parallelism increases as the graph becomes denser. In this problem, inter/intra-operator parallelism is an example of *nested data-parallelism* introduced by Blelloch in the context of the functional language NESL [9]. 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 4.2, so the only parallelism is intra-operator parallelism.

## 4. Morph algorithms

In this section, we discuss amorphous data-parallelism in algorithms that may morph the structure of the graph by adding or deleting nodes and edges. Although morph operators can be viewed abstractly as replacing sub-graphs with other sub-graphs, it is more insightful to classify them as follows.

- *Refinement*: Refinement operations make the graph bigger by adding new nodes and edges. In particular, algorithms that build trees top-down, such as Barnes-Hut [8] and Prim's MST algorithm [12], perform refinement.
- *Coarsening*: Coarsening operations cluster nodes or sub-graphs together, replacing them with new nodes that represent the cluster. In particular, algorithms that build trees bottom-up, such as Boruvka's algorithm [17], perform coarsening.
- *General morph*: All other operations that modify the graph structure fall in this category. Graph reduction of functional language programs is an example.

The type of morph used by an algorithm induces a particular parallelism profile. Generally, algorithms using coarsening morphs start out with a high degree of inter-operator parallelism, but the parallelism decreases as the graph becomes smaller and smaller. Conversely, refinement algorithms typically exhibit increasing inter-operator parallelism over time, until they run out of work.

### 4.1 Refinement

***Graphs*** Delaunay mesh refinement is an important algorithm used in mesh generation and graphics. The Delaunay triangulation for a set of points in the plane is the triangulation such that each triangle satisfies certain quality constraints. For a given Delaunay mesh, this is accomplished by *iterative mesh refinement*, which successively fixes "bad" triangles (those that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating. Figure 10 illustrates this process; the darker shaded triangles are "bad" triangles. To fix a triangle, a new point is added at the circumcenter and some neighbors, known as the *cavity*, may need to be re-triangulated. In the figure, the cavities are the lighter shaded triangles. Re-triangulating a cavity may generate new bad triangles but it can be shown that 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 [10]. Figure 11 shows the pseudocode for mesh refinement.

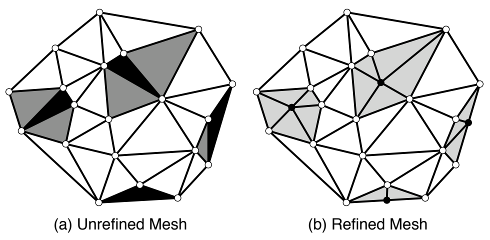


**Figure 10.** Fixing a bad element

```
Mesh mesh = /* read in initial mesh */;
Workset ws;
ws.add(mesh.badTriangles());
for each Element e in ws {
  if (e no longer in mesh) continue;
  Cavity c = new Cavity(e);
  c.expand(); c.retriangulate();
  Subgraph pre = c.getCavity();
  Subgraph post = c.getRetriangulatedCavity();
  mesh.replaceSubgraph(pre, post);
  ws.add(c.badTriangles());
}
```

**Figure 11.** Mesh refinement algorithm

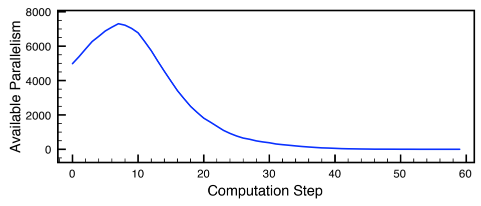
Figure 12(a) shows the parallelism profile for this application when run with an input of 100,000 triangles. As we can see, the available parallelism increases initially because new bad triangles are created by each refinement. As work is completed, the available parallelism drops off. This pattern seems common in refinement codes. Figure 12(b) shows the parallelism profile for another refinement code, Delaunay *triangulation*, which creates a Delaunay mesh from a set of points, run with an input of 10,000 points. The parallelism profile is similar in shape.

In Delaunay mesh refinement, the topology is a general graph, the operator is cautious, and active nodes are not ordered, so implementations can use autonomous scheduling with zero-copy, data-partitioning, and lock coarsening. It is also possible to use run-time coordination [31]. Verbrugge has formulated some mesh refinement operations in terms of rewrite rules [58].

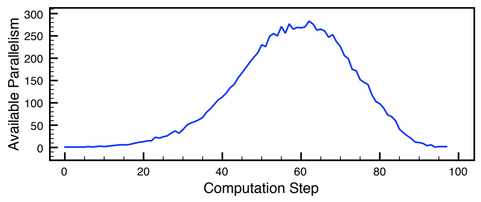
***Trees*** Figure 13 shows one implementation of Prim's algorithm for computing minimum spanning trees (MSTs) in undirected graphs. The algorithm builds the tree top-down and illustrates the use of ordered worksets. Initially, one node is chosen as the "root" of the tree and added to the MST (line 1), and all of its edges are added to the ordered workset, ordered by weight (line 3). In each iteration, the smallest edge, $(s, t)$ is removed from the workset. 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 (line 6), and all edges $(t, u)$ are added to the workset, provided $u$ is also not in the MST (lines 7-9). 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 it is organized around edges rather than nodes. The standard algorithm requires a priority queue in which node priorities can be decreased dynamically. It is unclear that it is worthwhile generalizing ordered-set iterators to permit this.

*Graph search* algorithms such as 15-puzzle solvers implicitly build spanning trees that may not be minimum spanning trees.



(a) Available parallelism in Delaunay mesh refinement



(b) Available parallelism in Delaunay triangulation

**Figure 12.** Available parallelism in two refinement algorithms

```
tree.setRoot(r); //r: arbitrary vertex
//priority queue worklist
//contains edges ordered by edge weights
OrderedWorkset ws;
for each edge (r,t,weight) do
    add (r,t,weight) to ws
for each edge (s,t,weight) in ws do //s is in tree
    {if (t.inTree()) continue;
     t.setParent(s);
     for each edge (t,u,weight) do
       if (! u.inTree())
              add (t,u,weight) to ws
   }
```

**Figure 13.** Top-down tree construction: Prim's MST algorithm

This can be coded by replacing the ordered-set iterator in Prim's algorithm with an unordered set iterator.

Top-down tree construction is also used in *n-body methods* like Barnes-Hut and fast multipole [8]. 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 as needed to maintain the invariant that a leaf node can contain at most a single particle. The work-set in this case is the set of particles, and it is unordered because particles can be inserted into the tree in any order. The operator is cautious, so a zero-copy implementation can be used. In practice, the tree-building phase of n-body methods does not take much time compared to the time for force calculations (see Section 6).

### 4.2 Coarsening

There are three main ways of doing coarsening.

- *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.
- *Node elimination*: A node is eliminated from the graph, and edges are added as needed between each pair of its erstwhile neighbors.
- *Subgraph contraction*: A sub-graph is collapsed into a single node and redundant edges are removed from the graph.

Coarsening morphs are commonly used in clustering algorithms in data-mining and in algorithms that build trees bottom-up.

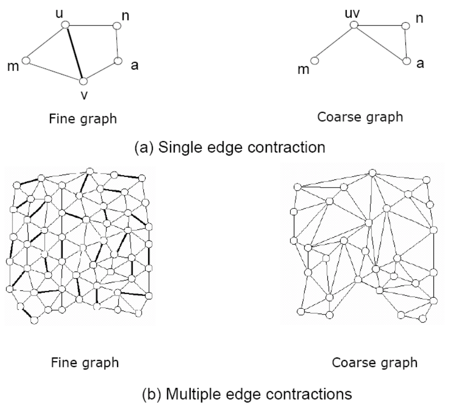


**Figure 14.** Graph coarsening by edge contraction

```
Graph g = /* read in input graph */
do {  //coarsen the graph by one level
  Graph cg = new Graph(g);  //copy g
  for each Node n in cg.getNodes() {
    Edge e = cg.getUnmatchedNeighbor(n);
    contract(e);
  }
  g = cg;
} while (!g.coarseEnough());

contract(Edge e) {
  //create representative node l, add it to graph
  Node l = cg.createNode();
  //for each neighbor of e.u, add an edge to l
  for each Node p in g.getNeighbors(e.u) {
    //add edge (l, p) or adjust weight if edge exists
    cg.buildEdge(l, p);
   }
  // ... repeat for neighbors of e.v ...
  //remove e.u & e.v from graph
  cg.removeNode(e.u);
  cg.removeNode(e.v);
}
```

**Figure 15.** Edge contraction in Metis

#### 4.2.1 Edge contraction

In Figure 14(a), the edge $(u, v)$ is contracted by fusing nodes $u$ and $v$ into a single new node labeled $uv$, and eliminating redundant edges. When redundant edges are eliminated, the weight on the remaining edge is adjusted in application-specific ways (in the example, the weight on edge $(m, uv)$ is some function of the weights on edges $(m, u)$ and $(m, v)$). As each edge contraction operation affects a relatively compact neighborhood, it is apparent that, in sufficiently large graphs, many edge contraction operations can happen in parallel, as shown in Figure 14(b). The darker edges in the fine graph show the edges being contracted, and the coarse graph shows the result of simultaneously contracting all selected edges.

***Graphs:*** Graph coarsening by edge contraction is a key step in important applications such as *graph partitioning*. For example, the Metis graph partitioner [33] applies edge contraction repeatedly until a coarse enough graph is obtained; the coarse graph is then partitioned, and the partitioning is interpolated back to the original graph.

The problem of finding edges that can be contracted in parallel is related to the famous graph problem of finding maximal matchings in general graphs [12]. Efficient heuristics are known for computing maximal cardinality matchings and maximal weighted matchings in graphs of various kinds. These heuristics can be used for coordinated scheduling. In practice, simple heuristics based on randomized matchings seem to perform well, and they are well-

```
 1: Graph g = /* input graph */;
 2: MST mst = new MST(g); //initialize MST from g
 3: Workset ws = new Workset(g.getNodes());
 4: for each Node n in ws {
 5:   Edge e = minWeight(g.getNeighbors(n));
 6:   Node l = contract(e); //contract edge e, forming l
 7:   // Add edge e to the MST
 8:   mst.addEdge(e);
 9:   //add new node back to worklist
10:   ws.add(l);
11: }
```

**Figure 16.** Edge contraction in bottom-up tree construction: Boruvka's algorithm for minimum spanning trees

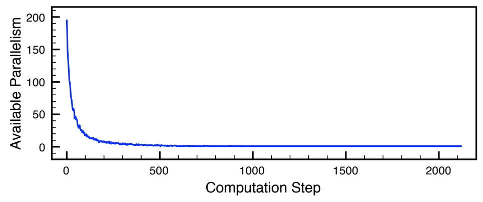


**Figure 17.** Available parallelism in Boruvka's algorithm.

suited for autonomous scheduling. Figure 15 shows pseudocode for graph coarsening as implemented within the Metis graph partitioner [33], which uses randomized matching. Each node finds an "unmatched" edge: an incident edge whose other node has not yet been involved in a coarsening operation yet (line 5), and then this edge is contracted. The operator is cautious, and active nodes are unordered.

***Trees:*** Some tree-building algorithms are expressed in terms of edge contraction. Boruvka's algorithm computes MSTs by performing edge contraction to build trees bottom-up [17]. It does so by performing successive coarsening steps. The pseudocode for the algorithm is given in Figure 16.

The active nodes are the nodes remaining the graph, and the MST is initialized with every node in the graph forming its own, single-node tree (line 2). In each iteration, an active node $u$ finds the minimum weight edge, $(u, v)$ incident on it (line 5), which is contracted to form a new node, $l$, (line 6). This is similar to the code in Figure 15 with the following exception: if there exist edges $(m, u)$ and $(m, v)$ in the fine graph, the edge $(m, l)$ in the updated graph will have weight equal to the minimum weight of the two original edges. The edge $(u, v)$ is then added to the MST to connect the two previously disjoint trees which contained $u$ and $v$ (line 8). Finally, $l$ is added back to the workset (line 10). This procedure continues until the graph is fully contracted. At this point, the MST structure represents the correct minimum spanning tree of the graph.

Figure 17 shows the parallelism profile for Boruvka's algorithm run on a random graph of 10,000 nodes, with each node connected to, on average, 5 others. We see that the amount of parallelism starts high, but as the tree is built, the number of active elements decreases and the likelihood that coarsening decisions are independent decreases correspondingly.

Interestingly, Kruskal's algorithm [12] also finds MSTs by performing edge contraction, but it iterates over an ordered workset of edges, sorted by edge weight.

### 4.2.2 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 these neighbors a clique, adjusting weights on the remaining nodes and edges appropriately. In Figure 18, node $u$ is eliminated, and edges $(x, z)$ and $(y, z)$ are inserted to make $\{x, y, z\}$ a clique. This is obviously a cautious oper-

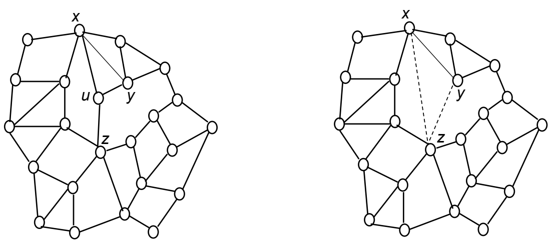


**Figure 18.** Graph coarsening by node elimination

ator. 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 [22]. The new edges inserted by node elimination are called *fill* since they correspond to zeroes in the matrix representation of the original graph that become non-zeros as a result of the elimination process. Different node elimination orders result in different amounts of fill in general, so ordering nodes for elimination to minimize fill is an important problem. The problem is NP-complete so heuristics are used in practice. One obvious heuristic is *minimal-degree ordering*, which is a greedy ordering that always picks the node with minimal degree at each elimination step. After some number of elimination steps, the remaining graph may be quite dense, and at that point, high-performance implementations switch to dense matrix techniques to exploit intra-operator parallelism and locality. A subtle variation of just-in-time coordination is used to schedule computations; in the literature, the inspector phase is called *symbolic factorization* and the executor phase is called *numerical factorization* [22].

### 4.2.3 Sub-graph contraction

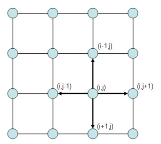
It is also possible to coarsen graphs by contracting entire sub-graphs at a time. In the compiler literature, *elimination-based algorithms* perform dataflow analysis on control-flow graphs by contracting "structured" sub-graphs whose dataflow behaviors have a concise description. Dataflow analysis is performed on the smaller graph, and interpolation is used to determining dataflow values within the contracted sub-graph. This idea can be used recursively on the smaller graph. Sub-graph contraction can be performed in parallel. This approach to parallel dataflow analysis has been studied by Ryder [40] and Soffa [35].

## 4.3 General morph

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

***Graphs:*** Graph reduction of functional language programs, and algorithms that maintain social networks fall in this category, as discussed in Section 3.

***Trees:*** Some algorithms build trees using complicated structural manipulations that cannot be classified neatly as top-down or bottom-up construction. For example, when a set of values is inserted into a heap, the final data structure is a tree, but each insertion may perform complex manipulations of the tree [12]. The structure of the final heap depends on the order of insertions, but for most applications, any of these final structures is adequate, so this is a case of don't-care non-determinism, which can be expressed using an unordered set iterator. This application may benefit from transactional memory because each insertion potentially modifies a large portion of the heap, but the modifications made by many insertions may be confined to a small portion of the heap.



```
//initialize array A
//5-point stencil over A
for time = 1, nsteps:
 for <i,j> in [2,n-1]x[2,n-1]:
  state(i,j).new=0.20*(state(i,j).old
   +state(i-1,j).old+state(i+1,j).old
   +state(i,j-1).old+state(i,j+1).old)
 //copy result back into A
 for <i,j> in [2,n-1]x[2,n-1]:
  state(i,j).old = state(i,j).new
```

(a) 5-point stencil (b) Code for 5-point stencil on $n \times n$ grid

**Figure 19.** Jacobi iteration on a 2D grid

```
Graph g = /* N nodes */
Workset ws = /* all nodes in g */
for each Node i in ws {
  for each Node j in g.neighbors(i) {
    state(i).y = state(i).y + A(i,j)*state(j).x;
  }
}
```

**Figure 20.** Sparse matrix-vector product

## 5. Local computation algorithms

Instead of modifying the structure of the graph, many graph algorithms operate by labeling nodes or edges with data values and updating these values repeatedly until some termination condition is reached. Updating the data on a node or edge may require reading or writing data values in the neighborhood. In some applications, data for the updates are supplied externally. These algorithms are called *local computation* algorithms in this paper. To avoid verbosity, we assume in the rest of the discussion that values are stored only on nodes of the graph, and we refer to an assignment of values to nodes as the *state* of the system.

Local computation algorithms come in two flavors.

- *Structure-driven algorithms:* In structure-driven algorithms, active nodes are determined completely by the structure of the graph. Cellular automata [60] and finite-difference algorithms like Jacobi and Gauss-Seidel iteration [23] are well-known examples.
- *Data-driven algorithms:* In these algorithms, updating the value at a node may trigger updates at one or more neighboring nodes. Therefore, nodes become active in a very data-dependent and unpredictable manner. Message-passing AI algorithms such as survey propagation [43] and algorithms for event-driven simulation [45] are examples.

### 5.1 Structure-driven algorithms

***Grids*** Cellular automata are perhaps the most famous example of structure-driven computations over grids. Grids in cellular automata usually have 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 $c$. A similar state update scheme is used in finite-difference methods for the numerical solution of partial differential equations, 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. Figure 19 shows a Jacobi iteration that uses a neighborhood called the *five-point stencil.* Each grid point holds two values called `old` and `new` that are updated at each time-step. Many other neighborhoods (stencils) are used in cellular automata [56] and finite-difference methods. All these algorithms perform unordered iteration over the nodes of a grid using a cautious operator, so they can be parallelized easily.

A disadvantage of Jacobi iteration is that it requires two arrays for its implementation. 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 in Jacobi iteration, 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, so the single global update step of Jacobi iteration is replaced by a sequence of smaller steps, each of which performs in-place updates to all nodes of a given color. For a five-point stencil, two colors suffice because grids are two-colorable, and this is the famous red-black ordering. These algorithms can obviously be expressed using unordered set iterators with one loop for each color.

In all these algorithms, active nodes can be determined from the grid structure, and the grid structure is invariant. As a result, compile-time scheduling can be very effective for coordinating the computation. Most parallel implementations of stencil codes partition the grid into blocks, and each processor is responsible for updating the nodes in one block. This minimizes inter-processor communication. Blocks are updated in parallel, with barrier synchronization used between time steps. In terms of the vocabulary of Figure 8, this is an example of coordinated, compile-time scheduling.

***General graphs*** The richest source of structure-driven algorithms on general graphs are iterative solvers for sparse linear systems, such as the conjugate gradient and GMRES methods. The key operation in these solvers 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. The state of a node $i$ consists of the current values of $x[i]$ and $y[i]$, and at each time step, $y[i]$ is updated using the values of $x$ at the neighbors of $i$. This is simply unordered iteration over the nodes in the graph, as seen in Figure 20.

The code for sparse MVM is clearly quite similar to the code for Jacobi iteration in Figure 19. In both cases, the current node is updated by reading values at its neighbors. The key difference is that in sparse MVM, the graph structure is not known until run-time. To handle this, parallel implementations of sparse MVM use the *inspector-executor* approach [61]: once the inputs are read, the structure of $A$ is analyzed to partition the graph and schedule the parallel execution. As the same $A$ is often used for multiple MVMs, the cost of partitioning and scheduling at run-time can be amortized over many iterations. In terms of the vocabulary of Figure 8, sparse iterative solvers can be handled using coordinated, just-in-time scheduling.

In some applications, the graph in which local computations must be performed is known only during the execution of the program. The augmenting paths algorithm for maxflow computations is an example [12]. This is an iterative algorithm in which each iteration tries to find a path from source to sink in which every edge has non-zero residual capacity. Once such a path is found, the global flow is augmented by an amount $\delta$ equal to the minimal residual capacity of the edges on this path, and $\delta$ is subtracted from the residual capacity of each edge. This operation can be performed in parallel as a structure-driven local computation. However, the structure of the path is discovered during execution, so this computation must be performed either autonomously or by coordinated scheduling at run-time.

```
Workset ws = /* nodes with excess flow */;
for each Node n in ws {
  for each Edge e in g.getOutgoingEdges(u) {
    //push flow from n along e = (u, v), updating
    //capacity of e, record excess flow
    flow = push(n, e);
    if (flow > 0) ws.add(e.v);
  }
  relabel(n); //raise n's height if necessary
  if (n.excess > 0)
    ws.add(n); //put n back if still active
}
```

**Figure 21.** Key kernel in preflow-push algorithm

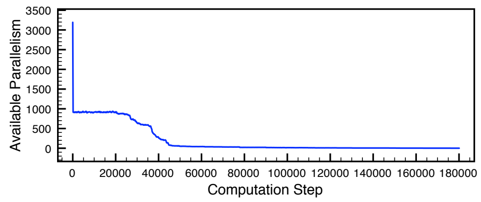


**Figure 22.** Available parallelism in preflow-push.

## 5.2 Data-driven algorithms

In some graph algorithms, the pattern of node updates is not determined by the structure of the graph; instead, updates are made initially to some nodes, and these updates trigger updates at neighboring nodes, so the updates ripple through the graph in a data-driven fashion. In the parlance of Section 2.2, some nodes begin as active nodes and processing them causes other nodes to become active. Examples include the preflow-push algorithm [4] for the maxflow problem, AI message-passing algorithms such as belief propagation and survey propagation [43], and discrete-event simulation [32, 45]. In other algorithms, such as some approaches to solving spin Ising models [51], the pattern of node updates is determined by externally-generated data such as random numbers.

Data-driven algorithms are naturally organized around worksets that maintain the sets of active nodes where updates need to be performed. While graph structure is typically less important than in structure-driven algorithms, it can nonetheless be useful in 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 famous *loopy propagation problem* [43]. Grid structure can be exploited in spin Ising solvers to reduce synchronization [51]. In this section, we describe the preflow-push algorithm, which uses an unordered workset, and discrete-event simulation, which uses an ordered workset. Both these algorithms work on general graphs.

***Preflow-push algorithm*** The preflow-push algorithm is used to solve the maxflow problem in general graphs. The word *preflow* refers to the fact that nodes are allowed to have excess in-flow at intermediary stages of the algorithm (unlike the augmenting paths algorithm, which maintains a valid flow at all times [12]). These are the active nodes in this algorithm. The algorithm performs two operations, *push* and *relabel*, on active nodes until termination.

Figure 21 shows the key loop in the preflow push algorithm (for simplicity, we have omitted the preprocessing and postprocessing code). The algorithm does not specify an order for applying the push or relabel operations, and hence it can be expressed using an unordered workset.

Figure 22 shows the available parallelism in preflow push for a 512x512 grid-shaped input graph. In preflow push, as in most local computation codes, the amount of parallelism is governed by the amount of work, as the likelihood that two neighborhoods overlap remains constant throughout execution. This can be seen by the relatively stable amount of parallelism at the beginning of computation; then, as the amount of work decreases, the parallelism does as well.

```
OrderedWorkset ows; //ordered by time
ows.add(initialEvents);
for each Event e in ows {
  newEvents = process(e);
  ows.add(newEvents);
}
```

**Figure 23.** Discrete-event simulation

***Discrete-event simulation*** In discrete-event simulation, the goal is to simulate a physical system consisting of one or more processing stations that operate autonomously and interact with other stations by sending and receiving messages. Such a system can be modeled as a graph in which nodes represent processing stations, and edges represent communication channels between processing stations along which messages are sent. In most applications such as circuit simulation or network simulation, a processing station interacts directly with only a small number of other processing stations, so the graph is very sparse.

Sequential discrete-event simulation is usually organized around a data structure called an *event list* that maintains a list of messages with their associated times of receipt. At each step, the earliest message in the event list is removed, and the action of the receiving station is simulated. This may cause other messages to be sent at future times; if so, these are added to the event list. Event-driven simulation can be written using an ordered set iterator, where messages are three-tuples $\langle value, edge, time \rangle$ and an ordered set maintains events sorted in time order, as shown in Figure 23.

In the literature, there are two approaches to parallel event-driven simulation, called conservative [45] and optimistic [32] event-driven simulation. 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 then operate autonomously without fear of deadlock, and there is no need to maintain an ordered event list. This is example of algorithm reformulation that replaces an ordered workset with an unordered workset. Time-warp, the optimistic simulation, is essentially an optimistic parallel implementation of an ordered set iterator. However, the global event list is eliminated in favor of periodic sweeps through the system to update "global virtual time" [32].

## 5.3 Discussion

Iterative methods for computing fixpoint solutions to systems of equations can usually be formulated in both structure-driven and data-driven ways. However, the convergence properties are usually different, so we consider them to be different algorithms. In simple problem domains, structure-driven and data-driven algorithms may produce the same final output. A classic example of this is iterative data-flow analysis. The system of equations can be solved by iterating over all the nodes until a fixed point is reached; this is a structure-driven approach. Alternately, the classic worklist algorithm only processes a node if its predecessors' output values changed; this is a data-driven approach [3]. In both cases, the algorithm can be parallelized in the same manner as other local computation algorithms.

***Speeding up local computation algorithms*** Local computation algorithms can often be sped up by a preprocessing step that coarsens the graph. This is the idea behind Allen and Cocke's *interval-based algorithm for dataflow analysis* [3]. This algorithm performs a preprocessing step in which it finds and collapses intervals, which are special single-entry, multiple-exit loop structures in the control-flow graph of the program, until an *irreducible* graph is obtained. If the irreducible graph consists of a single node, the

```
Scene s; //scene description
Framebuffer fb; //image to generate
Worklist ws = /* initial rays */
for each Ray r in ws {
  r.trace(s); //trace ray through scene
  ws.add(r.spawnRays());
}
for each Ray r in ws {
  fb.update(r); //update image with ray's contribution
}
```

**Figure 24.** Ray-tracing

solution to the dataflow problem can be read off by inspection. Otherwise, iterative dataflow analysis is applied to the irreducible graph.

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. The algorithm maintains a value called *height* at each node, which is a lower bound on the distance to the sink. Preflow-push can be sped up by periodically performing a breadth-first search from the sink to update the height values with more accurate information [4]. In the extreme, global information can be used in every iteration. For example, iterative linear solvers are often sped up by *preconditioning* the matrix with another matrix that is often obtained by a graph coarsening computation such as incomplete LU or Cholseky factorization [23].

## 6. Reader algorithms

Algorithms in the reader category perform multiple traversals over a fixed graph (*i.e.*, the graph structure is fixed and the values stored at nodes and edges are constant). Each of these traversal operations may contain some state that is updated, but, crucially, this state is private. Thus, traversals are independent and can be performed concurrently. The Google map-reduce [13] model is an example of this pattern in which the fixed graph is a set or multi-set.

### 6.1 Examples

Force computation in n-body algorithms like Barnes-Hut [8] is an example of the reader pattern. The construction of the oct-tree is an example of tree refinement, as was discussed in Section 4.1. Once the tree is built, the force calculation iterates over particles, computing the force on each particle by making a top-down traversal of a prefix of the tree. The force computation step of the Barnes-Hut algorithm is completely parallel since each point can be processed independently. The only shared state in this phase of the algorithm is the oct-tree, which is not modified.

Another classic example of the reader pattern is ray-tracing. The simplified pseudocode is shown in Figure 24. 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 iterated over and used to construct the rendered image. Ray-tracing is also embarrassingly parallel, as is evident from the lack of updates to the scene information.

These two algorithms hint at a general pattern that underlies reader algorithms. Each reader algorithm consists of two graphs and proceeds as follows: for each node $n$ in the first graph, the nodes and edges of the second graph are traversed, updating the state of $n$. Because updates only occur to nodes from the first graph, and these nodes do not affect one another, the algorithm can be trivially parallelized. In the case of Barnes-Hut, the first graph is the set of points (a degenerate graph with no edges), and the second graph is the oct-tree. In the case of ray-tracing, the first graph is the set of rays, and the second graph is the scene.

### 6.2 Enhancing locality

Because reader algorithms are embarrassingly parallel, the primary concern in achieving efficient parallelization is promoting locality. A common 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 oct-tree during the force computation and thus can be processed together to improve locality.

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 [49].

## 7. Conclusions and future work

In this paper, we introduced a generalization of data-parallelism called amorphous data-parallelism and argued that it is ubiquitous in important irregular algorithms. We also gave a classification of these algorithms and showed that this classification led to an execution model that provided insights into the parallelism in these algorithms and into mechanisms for exploiting this parallelism.

The use of graphs in parallel computing models has ancient provenance. One of the earliest graphical models is due to Karp and Miller who introduced *parallel program schema* for studying properties such as determinacy and boundedness [50]. The use of dataflow graphs as an organizational abstraction for parallel machine architectures is due to Jack Dennis [14]. This work was later extended by Arvind to the tagged-token dataflow model [5]. In all these models, graphs are representations of *computations*, whereas in our model, graphs represent data and can be mutated by operators.

The world-view presented in this paper provides a unified view of parallelism in regular and irregular algorithms; in fact, regular algorithms emerge as just a special case of irregular algorithms. In some ways, this is like the generalization in mathematics of metric spaces to general topologies in which notions like distance are replaced with general notions of neighborhoods and open sets, so metric spaces become a special (but important) case of general topologies.

In the restructuring compiler community, it is commonly believed that compile-time parallelization is the ideal, and that the inadequacies of current static analysis techniques can be remedied by developing more precise analysis techniques. The analysis in this paper suggests that optimistic or speculative parallelization may be the only general-purpose way of exploiting parallelism, particularly in algorithms in which active elements are ordered. In most of the algorithms discussed in this paper, dependences between activities are a function of the input data, so static analysis is likely to be overly conservative. However, we have also shown that for most algorithms, we do not need the full-blown machinery of optimistic parallelization as implemented, for example, in transactional memory (the Timewarp algorithm for event-driven simulation is the only exception).

In general, algorithms with ordered active elements are more difficult to parallelize than algorithms with unordered active elements. For some problems, such as event-driven simulation, the straight-forward algorithm has ordered active elements, but it is possible to reformulate the algorithm so that it uses unordered active elements. However, the unordered algorithm may have its own overheads, as in the case of event-driven simulation. It is likely that this kind of algorithm reformulation is very algorithm-specific, and that there are no general-purpose techniques for accomplishing this, but this needs to be investigated.

In the literature, there are many efforts to identify parallelism patterns [11, 44, 46, 53, 54]. The scope of these efforts is broader than ours: for example, they include divide-and-conquer algorithms in their purview, and they also seek to categorize implementation mechanisms such as whether task-queues or the master-worker approach is used to distribute work. Most of this work is descriptive, like the Linnaean taxonomy in biology; in contrast, our approach is synthetic like molecular biology: we show that there are a small number of parts from which one can "assemble" a wide range of iterative algorithms. Like us, Snir distinguishes between ordered and unordered work [54]. We make finer distinctions than the Berkeley dwarfs do. For example, dense matrix algorithms constitute a single Berkeley dwarf, but we would put iterative solvers in a different category (local computation) than dense matrix factorizations (morphs), and their parallel behavior is very different. Similarly, from our world-view, tree construction in Barnes-Hut is classified differently than particle pushing, whereas in the Berkeley dwarfs, there is a single category for n-body codes.

We hope that insights into amorphous data-parallelism in irregular algorithms will spur research into the design of abstract data types, concrete representations, and synchronization mechanisms. There is also much compiler and runtime systems work to be done. The execution model described in Section 3 should be viewed as baseline parallel implementation, which performs well for many algorithms. We now need to study hand-parallelized implementations of irregular algorithms to understand the tricks used by programmers and figure out how to incorporate them into compilers and runtime systems. For example, even applications like Delaunay mesh refinement that make complex modifications to general graphs have structure that can be used to reduce synchronization overheads, as we showed in Section 3.2. For researchers interested in finding applications for general-purpose optimistic synchronization mechanisms like transactional memory [18, 39], our studies provide examples where such mechanisms might be appropriate.

Algorithms used in compilers are largely irregular graph or tree algorithms, so it may be possible to parallelize compilers using the techniques in this paper. This would address a common criticism of restructuring compiler technology: it is not used to parallelize compilers themselves (this is humorously referred to as the "not eating our own dogfood" problem).

Finally, we note that the research methodology used by the parallel programming community relies heavily on instrumenting and running large benchmarks programs. In the spirit of Niklaus Wirth's motto "program = algorithm + data structure", we would argue that it is just as important to study algorithms and data structures because, while running instrumented programs provides data, studying algorithms and data structures provides insight.

*Acknowledgements:* We would like to thank Arvind (MIT), Don Batory (UT Austin), Jim Browne (UT Austin), Calin Cascaval (IBM Yorktown Heights), Jayadev Misra (UT Austin), David Padua (Illinois), Burton Smith (Microsoft) and Marc Snir (Illinois) for useful discussions.

## References

[1] Boost C++ libraries. `http://www.boost.org/`.

[2] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26(3):1–32, 2008.

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

[4] Richard J. Anderson and Joao C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *SPAA*, 1992.

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

[6] D.A. 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.

[7] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier Publishers, 1984.

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

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

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

[11] Murray Cole. *Algorithmic skeletons: structural management of parallel computation*. MIT Press, 1989.

[12] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford 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] Jack Dennis. Dataflow ideas for supercomputers. In *Proceedings of CompCon*, 1984.

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

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

[17] David Eppstein. Spanning trees and spanners. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier, 1999.

[18] Calin Cascaval et al. Software transactional memory: Why is it only a research toy? *CACM*, 51(11):40–46, November 2008.

[19] Ping An et al. Stapl: An adaptive, generic parallel c++ library. In *LCPC*. 2003.

[20] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990.

[21] 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.

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

[23] Gene Golub and Charles van Loan. *Matrix computations*. Johns Hopkins Press, 1996.

[24] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, August 1987.

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

[26] John Hennessy and David Patterson. *Computer Architecture: a quantitative approach*. Morgan-Kaufmann Publishers, 1996.

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

[28] Kirsten Hildrum and Philip S. Yu. Focused community discovery. In *ICDM*, 2005.

[29] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

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

[31] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse parallel Delaunay mesh refinement. In *SPAA*, 2007.

[32] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[33] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

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

[35] Robert Kramer, Rajiv Gupta, and Mary Lou Soffa. The combining DAG: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8), August 1994.

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

[37] Milind Kulkarni, Martin Burtscher, Rajshekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *ACM Symposium on Principles and Practice of Parallel Programming*, 2009.

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

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

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

[41] Edward Lorenz. Predictability- does the flap of a butterfly's wings in Brazil set off a tornado in Texas? In *Proceedings of the American Association for the Advancement of Science*, December 1972.

[42] M. Lowe and H. Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In *WG '90: Proceedings of the 16th international workshop on Graph-theoretic concepts in computer science*, pages 338–353, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

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

[44] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Publishers, 2004.

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

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

[47] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[48] Simon Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

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

[50] R.M.Karp and R.E.Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[51] Eunice E. Santos, Shuangtong Feng, and Jeffrey M. Rickman. Efficient parallel algorithms for 2-dimensional ising spin models. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 135, Washington, DC, USA, 2002. IEEE Computer Society.

[52] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[53] David Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[54] Marc Snir. http://wing.cs.uiuc.edu/group/patterns/.

[55] D. Spielman and SH Teng. Parallel Delaunay refinement: Algorithms and analyses. In *11th International Meshing Roundtable*, 2002.

[56] Tim Tyler. Cellular automata neighborhood survey. http://cell-auto.com/neighbourhood/index.html.

[57] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

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

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

[60] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[61] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44, 1995.