Ordered *vs.* Unordered: a Comparison of Parallelism and Work-efficiency in Irregular Algorithms *

M. Amber Hassaan
Electrical and Computer Engineering
University of Texas at Austin
amber@mail.utexas.edu

Martin Burtscher
Department of Computer Science
Texas State University-San Marcos
burtscher@txstate.edu

Keshav Pingali
Department of Computer Science
University of Texas at Austin
pingali@cs.utexas.edu

Abstract

Outside of computational science, most problems are formulated in terms of *irregular* data structures such as graphs, trees and sets. Unfortunately, we understand relatively little about the structure of parallelism and locality in irregular algorithms. In this paper, we study several algorithms for four such problems: discrete-event simulation, single-source shortest path, breadth-first search, and minimal spanning trees. We show that these algorithms can be classified into two categories that we call *unordered* and *ordered*, and demonstrate experimentally that there is a trade-off between parallelism and work efficiency: unordered algorithms usually have more parallelism than their ordered counterparts for the same problem, but they may also perform more work. Nevertheless, our experimental results show that unordered algorithms typically lead to more scalable implementations, demonstrating that less work-efficient irregular algorithms may be better for parallel execution.

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: Irregular Algorithms, Amorphous Data-parallelism, Parallel Breadth-first Search, Single-Source Shortest Path, Discrete-Event Simulation, Minimal Spanning Tree, Multicore processors, Galois system.

1. Introduction

Over the past two decades, the parallel programming community has acquired a deep understanding of the patterns of parallelism and locality in *regular*, dense matrix algorithms. These insights have led to the development of many languages, tools and techniques that have made it easier to develop high-performance implementations of regular algorithms [5, 14, 26].

Outside of computational science, most problems are *irregular* since the underlying data structures are pointer-based data structures such as trees and graphs. At present, we have few insights into the structure of parallelism and locality in irregular algorithms, and this has stunted the development of techniques and tools for programming these algorithms in parallel [6]. A major complication is that dependences in irregular algorithms are functions of runtime data values such as the inputs to the program. Therefore, in general, one cannot generate a useful static dependence graph for an irregular algorithm, and it is not possible to come up with asymptotic estimates of parallelism in terms of the size of the problem (in contrast, we can say that there are $O(N^3)$ scalar multiplications that can be done concurrently when multiplying two $N \times N$ dense matrices, for example).

In previous work, we have shown that these problems can be circumvented by taking a *data-centric* view called the *operator formulation of algorithms* in which an algorithm is expressed in terms of its action on data structures [27]. In graph algorithms, the unit of action is the application of an operator to an *active node*, which is a site in the graph where computation needs to be performed. Usually, there are many active nodes in a graph and there may be ordering constraints in the processing of these nodes. In *unordered* algorithms such as Delaunay mesh refinement and preflow-push maxflow computation, active nodes can be processed in any order; in *ordered* algorithms such as event-driven simulation, there is a (partial) order on active nodes that dictates the order in which these nodes must be processed. Applications formulated in this way can be executed in parallel by processing multiple active nodes speculatively [20]. This parallelism, which is called *amorphous data-parallelism* [27], can be exploited by the Galois system, described in more detail in Section 2.

Although many irregular problems can be solved using both ordered and unordered algorithms, the trade-offs between these algorithms are not well understood. In this paper, we study ordered and unordered algorithms for four irregular problems: breadth-first search, discrete-event simulation, minimal spanning tree, and single-source shortest path. This paper makes the following contributions.

- We show that contrary to popular belief, many ordered algorithms exhibit substantial amorphous data-parallelism.
- We find that for the same problem, unordered algorithms have shorter critical paths than ordered algorithms, but may perform more work than ordered algorithms do. Unless care is taken, the amount of additional work may grow with the number of threads used to the execute the program, so it is not always obvious what the best parallel algorithm is for a given problem.
- Fortunately, careful scheduling of computations in unordered algorithms can limit the amount of extra work performed by these algorithms. This allows us to produce scalable parallel implementations of unordered algorithms. However, we do not currently have effective techniques for producing scalable implementations of ordered algorithms.

The rest of this paper is organized as follows. Section 2 reviews amorphous data-parallelism and the Galois and ParaMeter systems. Section 3 describes ordered and unordered algorithms for breadth-first search, single-source shortest path, discrete-event simulation, and minimal spanning tree, and reports measurements of the amount of amorphous data-parallelism in these algorithms. Section 4 dis-

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02. . . $10.00

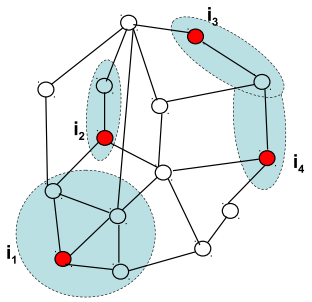


Figure 1. Amorphous data-parallelism in graph algorithms. Dark colored nodes are *Active* nodes and shaded regions are *Neighborhoods*. $i_1, ..., i_4$ are *Activities*

cusses implementation issues for ordered and unordered algorithms. Section 5 presents experimental results obtained by running efficient implementations of unordered algorithms on a multicore processor. Section 6 discusses related work. Section 7 summarizes our conclusions.

2. Amorphous data-parallelism

We use the graph shown in Figure 1 to explain the operator formulation of irregular algorithms, and introduce the Galois system. At each point during the execution of such an algorithm, there are certain nodes or edges in the graph where computation might be performed. Performing a computation may require reading or writing other nodes and edges in the graph. The node or edge on which a computation is centered is called an *active element*, and the computation itself is called an *activity*. It is convenient to think of an activity as resulting from the application of an *operator* (graph transformer) to the active element. The set of nodes and edges that are read or written in performing the activity is called the *neighborhood* of that activity. In Figure 1, the filled nodes represent active nodes, and shaded regions represent the neighborhoods of those active nodes. In some algorithms, activities may modify the graph structure of the neighborhood by adding or removing graph elements. To keep the discussion simple, we assume from here on that active elements are nodes, unless otherwise stated.

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 unordered algorithms, such as the BFS algorithm in Figure 3, the implementation can pick *any* active element for execution. In ordered algorithms, such as the BFS algorithm in Figure 2, there are ordering constraints on the processing of active nodes.

Opportunities for exploiting parallelism in irregular algorithms arise as follows. If there are many active nodes at some point during the computation, each one is a site where a thread 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 algorithms, this order must be the same as the ordering on active nodes. Given a set of active nodes and an ordering on them, amorphous data-parallelism is the parallelism that arises from simultaneously processing active nodes, subject to these constraints.

2.1 The Galois system

The Galois system is an implementation of these abstractions that enables the exploitation of amorphous data-parallelism on multicore processors [20]. We summarize the main points of this system to make the paper self-contained.

Irregular algorithms can be formulated as iteration over a workset of active nodes. Galois introduces two constructs called *Galois set iterators* for specifying iteration over unordered and ordered sets. These iterators are similar to conventional set iterators, but *they permit new elements to be added to a set while it is being iterated over*.

- **Unordered-set iterator: foreach (e: Set S) do B(e)** The loop body B(e) is executed for each element e of set S in some arbitrary order. The execution may add new elements to S.
- **Ordered-set iterator: foreach (e: OrderedSet S) do B(e)** Similar to the unordered set iterator, except a sequential implementation must choose a minimal element from S in every iteration and the ordering is specified by a user provided comparator.

Note that these iterators have well-defined sequential semantics. In addition, the unordered set iterator expresses the fact that active elements can be processed in any order.

All concurrency control is performed in a library of concurrent data structures similar to the Java collections library, containing concurrent implementations of common data structures like graphs. The library also provides implementations of parallel unordered and ordered sets for implementing Galois set iteration in parallel.

In the baseline execution model, the graph is stored in shared-memory, and active nodes are processed by some number of threads. A free thread picks an arbitrary active element from the workset and speculatively applies the operator to that element, making calls to the graph class 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 element and spreads incrementally whenever a graph API call is made that touches new nodes or edges. To ensure that neighborhood constraints are respected, mechanisms similar to those used in thread-level speculation or software transactional memory can be used in principle [7, 9, 16, 29]. Each graph element has an associated logical lock that must be acquired before a thread can access that element. Locks are held until the activity terminates. Lock manipulation is performed by the code in the graph API call, not in the application code. 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. To enable rollback, each graph API method that modifies the graph records *undo* information, *i.e.*, copies of the old data. Like lock manipulation, rollbacks are a service provided by the library and runtime system.

If active nodes are not ordered, the activity terminates when the application of the operator is complete and all acquired locks are released. If active nodes 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 [20]. In this case, locks are released when the activity commits.

2.2 ParaMeter

One measure of amorphous data-parallelism in the execution of an irregular algorithm for a given input is the number of active nodes that can be processed in parallel at each point during the execution *if one had perfect knowledge of neighborhood and ordering constraints*. This is called the *available parallelism* at each step of computation, and a plot showing the available parallelism of an irregular algorithm for a given input is called a *parallelism profile*. Intuitively, this is the parallelism in the execution of the program, assuming that (i) each activity takes one time unit, (ii) there is an unbounded number of processors, and (iii) active nodes are scheduled greedily (neighborhood conflicts between activities of the same priority are resolved by rolling back one of the conflicting activities, chosen at random). In principle, the precise shape of the parallelism profile may depend on how neighborhood conflicts are resolved, but

in practice, the parallelism profiles are fairly robust with respect to this choice. The total number of computation steps needed to execute an algorithm for a given input reflects the critical path length in this model.

In this paper, we present parallelism profiles generated by the ParaMeter tool [21]. In studying these profiles, it is important to note that ParaMeter considers conflicts only the abstract data type level; in particular, if the neighborhoods of two activities are disjoint, ParaMeter assumes that there are no conflicts between them. Therefore, these profiles are independent of the concrete representation of the graph.

3. Case studies

This section presents ordered and unordered algorithms for four irregular problems: breadth-first search, single-source shortest-path computation, discrete-event simulation, and minimal spanning tree computation.

3.1 Breadth-First Search

Breadth-first search (BFS) of undirected graphs starts from a *root*, and explores all the neighbors of this node before exploring their neighbors, *etc*. One formulation of BFS is that it computes a value called $level$ for each node n such that $level(n)$ is the number of edges on the shortest path from the root to n.

```
1  Graph g = /* read in graph */
2  OrderedWorkSet ows; //ordered by level
3  ows.add(root); //root has level 0
4  foreach (Node n: ows) {
5    int level = n.getLevel() + 1;
6    for (Node m: g.neighbors(n)) {
7      if (m.getLevel() == INF) { //not visited
8        m.setLevel(level);
9        ows.add(m); //add to ows to update neighbors
10 } } }
```

Figure 2. Ordered BFS algorithm

Figure 2 shows pseudocode for the standard BFS algorithm. Initially, the $level$ of the root is set to zero, and the $level$ of all other nodes is set to ∞ (this can be done while reading in the graph). The algorithm maintains a workset of nodes, ows, ordered by $level$. Initially, only the root node $root$ is in the workset. Each iteration of the loop removes a node n from the workset and visits the neighbors of n. If a neighbor has not been visited before, *i.e.*, its $level$ is infinity, its $level$ is set to $level(n)+1$, and it is added to the workset. The algorithm terminates when the workset is empty. This is an *ordered* algorithm because the workset must be processed in a particular order (in this case, in level order).

```
1  Graph g = /* read in graph */
2  WorkSet ws;
3  ws.add(root);
4  foreach (Node n: ws) {
5    for (Node m: g.neighbors(n)) {
6      int level = n.getLevel() + 1;
7      if (level < m.getLevel()) { //can be relaxed
8        m.setLevel(level);
9        ws.add(m); //add to the ws to update neighbors
10 } } }
```

Figure 3. Unordered BFS algorithm

A different algorithm can be obtained by noticing that the $level$ of a node is a local minimum in the graph, *i.e.*, the $level$ of a node (except for the root) is one higher than the minimum $level$ of its neighbors. Hence, the computation of $level$ for node n can be described by the following fixpoint system.
Initialization:
$level(root) = 0; level(k) = \infty, \forall k\ other\ than\ root$
Fixpoint computation:
$level(n) = min(level(m) + 1,\ \forall m\ \in\ neighbors\ of\ n)$

Because of this property, it is possible to process the nodes from the workset in any order, thus yielding an unordered algorithm in which multiple nodes can be processed in parallel. The pseudocode is given in Figure 3. The ordered workset is replaced with an unordered set from which an arbitrary element can be selected. In the unordered algorithm, it may happen that a node is temporarily assigned a $level$ that is higher than the final value. However, the $level$ will monotonically decrease until it reaches the correct value. Whenever a thread updates the $level$ of a node, it adds that node to the workset so that its neighbors can be updated if necessary. The algorithm finishes when the workset is empty. The unordered algorithm may execute more iterations than the ordered algorithm; exactly how many more depends on the scheduling policy implemented in the workset. We discuss this in more detail in Section 5.

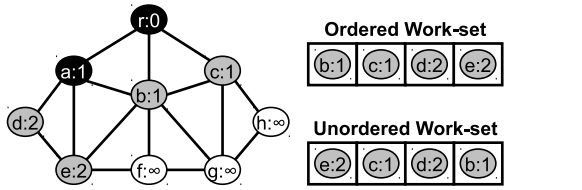


Figure 4. BFS example. The root node r and node a have been processed from the workset (colored black). The nodes in grey are currently active and in the workset. The nodes in the ordered workset are ordered by their level.

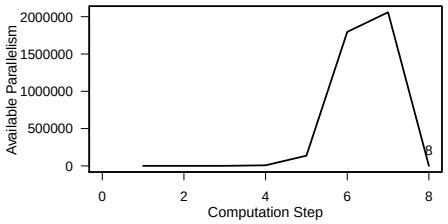
The monotonicity of updates in the unordered BFS algorithm enables an important optimization: there is no need to keep undo information for the updates performed by multiple threads on a node as long as the updates are atomic and visible to all the threads. Since only a single value is updated, this update can be implemented efficiently using an atomic primitive like compare-and-swap (CAS), and there is no need to use abstract locks for conflict detection.

3.1.1 BFS: Available parallelism

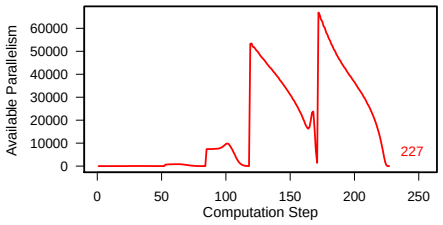
For our BFS experiments, we used a randomly connected undirected graph containing 4 million nodes and approximately 36.8 million edges. The degree of each node is chosen randomly in the range $[10, 20]$. Figure 5 shows the parallelism profiles produced by ParaMeter for the unordered and ordered BFS algorithms. The unordered algorithm has a critical path of just eight computation steps, whereas the ordered algorithm has a critical path that is more than 25 times as long. Our implementation of unordered algorithm uses CAS operations instead of abstract locks, which is why ParaMeter does not report any neighborhood conflicts. At every computation step i, all nodes at level i are processed and the nodes at level $i+1$ become active. Thus, in a graph with average degree d, the available parallelism increases roughly as d^i. In the ordered algorithm, the available parallelism also increases but more slowly because of the ordering constraints and neighborhood conflicts. Nevertheless, there is substantial parallelism even in the ordered version. Both versions perform 4 million iterations (one iteration per node). There is no wasted work performed by the unordered version because the schedule used by ParaMeter ends up being the same as the schedule used by the ordered version; other schedules may perform wasted work.

3.2 Single-Source Shortest Path

The single-source shortest-path problem (SSSP) finds the shortest paths from a given source node to all other nodes in a graph with weighted edges. The edge weights represent the distance between adjacent nodes. SSSP computes a distance $d(n)$ for every node n in



(a) Unordered BFS algorithm



(b) Ordered BFS algorithm

Figure 5. BFS parallelism profiles of a random graph with 4M nodes and approximately 36.8M edges

the graph such that $d(n)$ equals the distance from the source node to n along the shortest path. The distances of all nodes other than the source node are initialized to ∞, and then iteratively reduced to the correct value. BFS can be seen as a special case of SSSP in which all edge weights are 1.

A well-known ordered algorithm for computing SSSP is Dijkstra's algorithm, presented in Figure 6. Conceptually, the algorithm grows a shortest-path subtree of nodes and examines the edges crossing the cut between the nodes in the tree and the rest of the graph. For each edge (n, m) crossing the cut, it adds a pair $< m, distance >$ to the workset, where $distance = d(n) + weight(n, m)$. The workset is ordered by the $distance$ of each pair. Initially, the tree contains only the source node as the root, and the workset contains tuples corresponding to the neighbors of the source node. The algorithm repeatedly removes the pair with the minimum distance from the workset and, if the corresponding node has not already been visited, updates the node's distance and adds $< neighbor, distance >$ pairs to the workset for each of its neighbors. Note that the distance of each node is updated exactly once and that $< node, distance >$ pairs that refer to different nodes can be processed in parallel. However, any change made to the distance of a node must remain speculative until the distance becomes the minimum in the workset because a pair that is processed later may produce a shorter distance for a node than an earlier pair did.

```
1  Graph g = /* read in graph */
2  OrderedWorkSet ows; //ordered by distance
3  for (Node m: g.neighbors(root)) {
4    ows.add(<m, g.edgeWeight(root, m)>);
5  }
6  foreach (Pair <node, dist>: ows) {
7    if (node.getDist() == INF) { // not visited before
8      node.setDist(dist);
9      for (Node m: g.neighbors(node)) { //create tuples
10       if (m.getDist == INF) { //for unvisited neighbors
11         ows.add(<m, dist + g.edgeWeight(node, m)>);
12 } } } }
```

Figure 6. Dijkstra's ordered SSSP algorithm

To derive the unordered algorithm, we note that shortest distance from the source to each node in the graph can be computed using the following fixpoint system.
Initialization:
$d(root) = 0; d(k) = \infty,\ \forall k\ other\ than\ root$

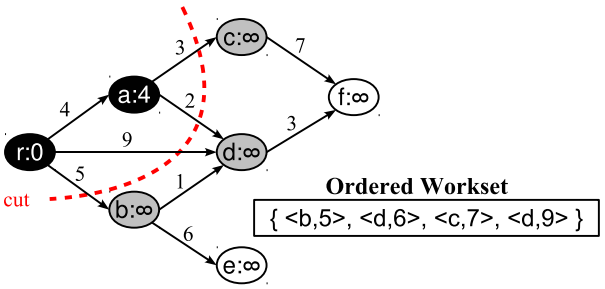


Figure 7. Dijkstra on sample graph. Tuple $< a, 4 >$ has been processed from the workset (shown in black). Tuples corresponding to the edges crossing the cut are in the workset and shown in grey.

Fixpoint computation:
$d(n) = min(d(m)+w(m,n),\ \forall m \in incoming\ neighbors\ of\ n)$

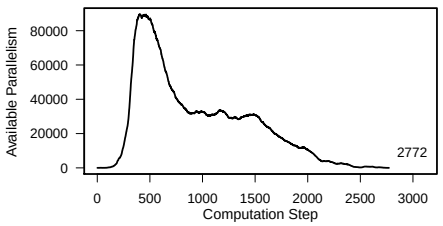
where $w(m, n)$ is the edge weight, *i.e.*, the distance between m and n. This observation yields an unordered algorithm for SSSP that is essentially the Bellman-Ford algorithm [4, 12, 15]. A node may be updated multiple times, but its distance is guaranteed to decrease monotonically to the correct value. Figure 8 shows the pseudocode for the unordered algorithm based on the fixpoint equation. Initially, the workset contains only the source node. In every step, the algorithm removes a node from the workset and tries to reduce the distance of each outgoing neighbor. If the distance of a neighbor is reduced, this neighbor is added to the workset. Hence, the workset always contains the nodes whose neighbors may need to be updated. The algorithm terminates when no more updates have to be performed. In both the ordered and the unordered algorithms, the neighborhood consists of the neighbors of the node being updated and the node itself.

```
1  Graph g = /* read in graph */
2  WorkSet ws;
3  ws.add(root);
4  foreach (Node n: ws) {
5    for (Node m: g.neighbors(n)) {
6      int dist = n.getDist() + g.edgeWeight(n, m);
7      if (dist < m.getDist()) { // can be relaxed
8        m.setDist(dist);
9        ws.add(m); // add to ws to update neighbors
10 } } }
```

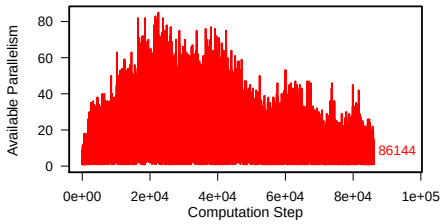
Figure 8. Unordered SSSP algorithm

3.2.1 SSSP: Available parallelism

Figure 9 shows the parallelism profile for the ordered and unordered SSSP algorithms. The input is a directed graph representing the road network of Florida [1]. The graph contains 1.07 million nodes and 2.68 million edges. The edge weights reflect the distances between adjacent locations. Like the unordered BFS implementation, the implementation of the unordered algorithm does not use logical locks. The available parallelism initially increases exponentially because, at each step, all nodes in the workset can be processed in parallel without conflicts. This results in a shorter critical path for the unordered algorithm. The available parallelism drops when the algorithm runs out of nodes to process. Dijkstra's ordered algorithm also shows significant parallelism, almost 20 parallel activities on average, but it has a critical path that is over 30 times longer. Note that it executes roughly 50 times fewer activities than the unordered algorithm (1.34 million compared to 68.06 million). Because active nodes cannot be committed until they become the highest priority nodes in the system, the available parallelism in Dijkstra's algorithm increases slowly and intermittently drops to low values.



(a) Unordered SSSP algorithm



(b) Dijkstra's ordered SSSP algorithm

Figure 9. SSSP parallelism profiles of the road network of Florida containing 1.07M nodes and 2.68M edges

3.3 Discrete-Event Simulation

In discrete-event simulation (DES), 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[1] can be sent. In most applications such as circuit or network simulation, a processing station interacts directly with only a small number of other processing stations, so the graph is sparse.

```
1 Graph g = /* read in graph */
2 OrderedWorkSet ows;
3 ows.add(initialEvents); //ordered by timestamp
4 foreach (Event e: ows) {
5   Station s = e.getStation(); // target station
6   List newEvents = s.simulate(e); //newly created
7   ows.add(newEvents); //events are added to the ows
8 }
```

Figure 10. Ordered DES algorithm

Figure 10 shows the pseudocode for the ordered discrete-event simulation algorithm. An ordered workset holds the pending events, ordered by their timestamps. In each step, the earliest message in the workset 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 messages are added to the workset. In principle, parallelism can be exploited in discrete-event simulation by processing multiple events from the workset concurrently. However, the conditions for non-interference of these activities are fairly complex. In Figure 11, there are two events in the workset for stations A and C, but it may or may not be legal to process them in parallel. If station A processes the event at its input and produces an event with timestamp 3, which is then processed by station B and B produces an event with timestamp 4, that event must be processed by station C before it processes the event with timestamp 5 and there are no opportunities for parallel execution. However, if the event produced by station B has a timestamp greater than 5, it is legal to process the two events, arriving at A and C, in parallel. Whether two events can be processed in parallel at a given point depends on what happens later in the execution.

[1] We use the terms *Message* and *Event* interchangeably in this discussion

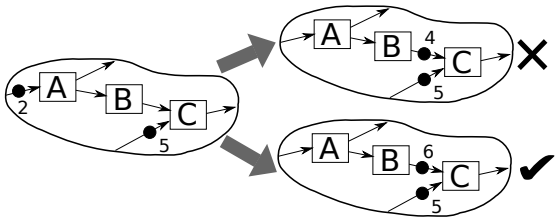


Figure 11. Discrete-event simulation. A, B and C are processing stations. A and C have incoming events with timestamps 2 and 5, respectively. Whether A and C can process events in parallel depends on the events generated by A (and B)

Chandy and Misra [8] proposed an unordered algorithm for DES in which additional messages called NULL messages are sent to permit stations to decide locally whether it is safe to process a given event. In this algorithm, a station can process events only after it has received events on all of its inputs, at which point it determines a set of *Ready Events*. Let T_i be the timestamp of the latest event received on input i of a station. Then the set of Ready Events is $\{Event\ e \mid timestamp(e) \leq min(T_i\ \forall i)\}$. The key observation is that the events received on an input always have monotonically increasing timestamps. A station can therefore safely process all events in the Ready Events set because it will never receive an event with a timestamp lower than $min(T_i)$. A station with a non-empty Ready Events set is called an *active* station. When a station sends a message with timestamp T to one of its neighbors, it also sends a NULL message with timestamp T to its other neighbors. When a station processes an incoming NULL message, it sends out NULL messages (with timestamps incremented by its processing time) to all of its neighbors.

Figure 12 shows the pseudocode for the unordered DES algorithm. The nodes corresponding to active stations are stored in an unordered workset. The call to `simulate()` computes and processes the Ready Events. The outgoing neighbors that have become *active* are added to the workset. The algorithm terminates when there are no active stations left.

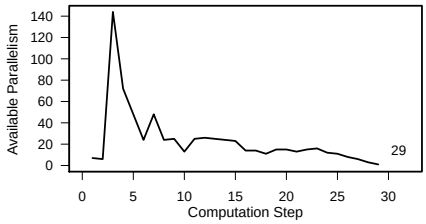
```
1 Graph g = /* read in graph */
2 WorkSet ws;
3 ws.addAll(g.readyNodes()); // nodes with initial events
4 foreach (Node n: ws) {
5   Station s = n.getStation();
6   s.simulate(); //process ready events
7   for (Node m: g.neighbors(n)) {
8     if (m.isActive()) //has non-empty ready-events set
9       ws.add(m);
10 } }
```

Figure 12. Chandy-Misra's unordered DES algorithm

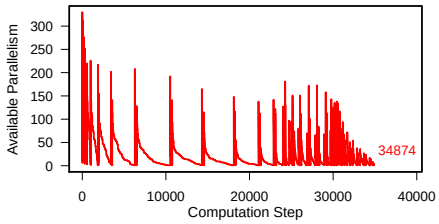
3.3.1 DES: Available parallelism

We show the parallelism profile for Chandy-Misra's unordered and the ordered DES algorithms in Figure 13. The input is a 6×6 tree multiplier circuit represented as a directed graph with 688 nodes and 1266 edges; its simulation involves processing 699,621 events. The active elements in the ordered and the unordered algorithms are very different. When a node executes in the unordered algorithm, it processes all the events in its Ready Set, and this is counted as a *single* activity. In the ordered algorithm, the processing of each event is an activity. The unordered algorithm has a critical path that is more than 1000 times shorter than that of the ordered algorithm, and it executes fewer activities (688 compared to 699,621); however, this difference is an artifact of the different notions of activity in the two algorithms. The shape of the available parallelism curve depends on the input. Here, the graph represents a circuit with a small number of inputs and outputs but a substantial amount of circuitry in the

"middle". Thus, the available parallelism increases rapidly and then gradually drops off.



(a) Chandy-Misra's unordered DES algorithm



(b) Ordered DES algorithm

Figure 13. DES parallelism profiles of a 6×6 tree multiplier with 688 nodes and 699,621 events

3.4 Minimal Spanning Tree

MST algorithms compute a minimal-weight spanning tree of a weighted, undirected graph. We study Prim's and Kruskal's algorithms, which are ordered, and Boruvka's algorithm, which is unordered.

```
1  Graph g = /* read in graph */
2  OrderedWorkSet ows; //ordered by weight
3  for (Node m: g.neighbors(root)) {
4    //tuple <node, parent, weight>
5    ows.add(<m, root, g.edgeWeight(root, m)>);
6  }
7  foreach (Tuple <n, parent, weight>: ows) {
8    if (n.getWeight() == INF) { // not visited
9      n.setWeight(weight);
10     n.setParent(parent);
11     for (Node m: g.neighbors(n)) { //create tuples for
12       if(m.getWeight() == INF) { //unvisited neighbors
13         ows.add(<m, n, g.edgeWeight(n, m)>);
14 } } } }
```

Figure 14. Prim's ordered MST algorithm

Prim's algorithm uses an approach similar to Dijkstra's SSSP algorithm. Figure 14 shows the pseudocode. The algorithm starts with a component containing only the root and examines the edges crossing the cut between the component and the rest of the graph. For each examined edge, it adds a tuple $< node, parent, weight >$ to an ordered workset, where the weight of a tuple is the weight of the edge under consideration. (In Dijkstra's algorithm, the weight is the total path weight from the source.) The parallelism in this algorithm is similar to the parallelism in Dijkstra's algorithm. The neighborhood comprises the node of the tuple being processed and its neighbors. Conflicts are resolved based on the weight of the tuple.

Kruskal's algorithm is another ordered MST algorithm. Figure 15 lists the pseudocode. This algorithm begins with an ordered workset containing all edges of the graph sorted by weight. It then removes the edges from the workset and adds them to either an existing MST component or a newly created component, but only if adding the edge does not create a cycle. A cycle would be introduced if the two nodes connected by the edge belong to the same component. The components are managed through a Union-Find data structure, where each node has a link to the representative of its component. Kruskal's algorithm can process multiple edges in parallel if the nodes comprising each edge belong to different components. The neighborhood of an edge includes its two nodes and the nodes visited during the `findSet()` operation, which follows the links to the representative node.

```
1  Graph g = /* read in graph */
2  OrderedWorkSet ows; //ordered by weight
3  ows.addAll(g.edges());
4  Set mst;
5  foreach (Edge e: ows) {
6    Node rep1 = findSet(e.firstNode())  //find the rep.
7    Node rep2 = findSet(e.secondNode()) //of my component
8    if (rep1 != rep2) { //must be diff. components
9      mst.add(e);
10     union(rep1,rep2); //merge the two components
11 } }
```

Figure 15. Kruskal's ordered MST algorithm

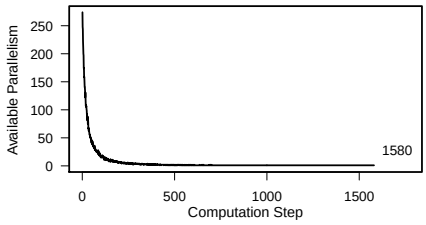
Boruvka's algorithm is an unordered MST algorithm. It is similar to Kruskal's algorithm in that it also creates independent components in parallel. The key insight that allows Boruvka's algorithm to be unordered is that the lightest-weight edge incident upon a node must always be a part of the MST. Figure 16 presents the pseudocode. Initially, each node forms its own MST component and is added to an unordered workset. Then, the algorithm finds the lightest edge leaving a component that does not introduce a cycle and contracts this edge. Contracting involves combining the nodes at either end of the edge into a single component; therefore the neighborhood consists of the these two nodes and their neighbors. The algorithm terminates when there is only one component left.

```
1  Graph g = /* read in graph */
2  WorkSet ws;
3  ws.addAll(g.nodes());
4  Set mst; //will contain the mst edges
5  foreach (Node n: ws) {
6    Edge e = n.lightestEdge();
7    if (e != null) {
8      mst.add(e);
9      Node m = graph.contract(e); //removes cycles
10     ws.add(m); //add the super-node back
11 } }
```

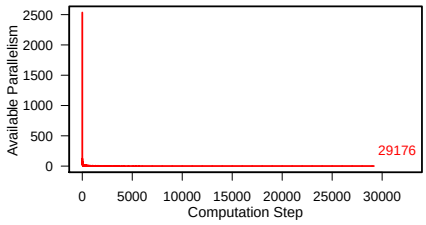
Figure 16. Boruvka's unordered MST algorithm

3.4.1 MST: Available parallelism

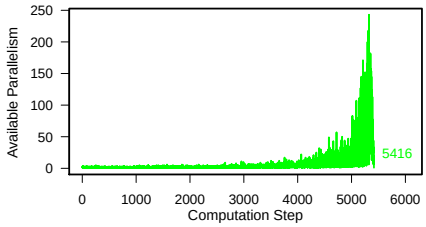
Figure 17 shows the parallelism profiles for Boruvka's, Kruskal's, and Prim's MST algorithms. The input is a random undirected graph with 10,000 nodes and 35,605 edges; the edges have random, uniformly distributed integer weights between 1 and 1000. The active elements in the three algorithms are different; they are nodes in Boruvka's algorithm, edges in Kruskal's algorithm, and 3-tuples, corresponding to edges, in Prim's algorithm. The main source of wasted work in Boruvka's algorithm is the nodes left in the workset that have already been merged into other components and are no longer a part of the graph. This is why Boruvka's algorithm executes more than 10,000 activities (19,999 in this particular case). Kruskal's algorithm has the longest critical path, while Boruvka's has the shortest. Both Kruskal's and Boruvka's algorithm are *coarsening* algorithms because they eliminate edges from the graph, which explains why they have similar looking parallelism profiles. This shape is characteristic of other coarsening algorithms as well [27]. Prim's algorithm, on the other hand, is a refinement algorithm because it adds new nodes to the tree, which is why it has a very different parallelism profile.



(a) Boruvka's unordered MST algorithm



(b) Kruskal's ordered MST algorithm



(c) Prim's ordered MST algorithm

Figure 17. MST parallelism profiles of a random graph with 10,000 nodes and 35,605 edges

4. Implementation

The parallelism profiles in Section 3 demonstrate that there is a significant amount of parallelism in both ordered and unordered irregular algorithms. To study how well this parallelism can be exploited in practice, we implemented these algorithms in the Galois system [20]. In this section, we describe a number of important optimizations for performance.

4.1 Unordered algorithms

We applied the following optimizations to our implementations of unordered algorithms [24].

Work chunking: To reduce the overhead of accessing the workset, we combine the active elements into chunks. This optimization is similar to Chunked Self Scheduling (CSS) [28, 31] and Iteration Coalescing [24]. When a thread accesses the workset, it removes a chunk of active elements instead of just one element. It then iterates over these elements without accessing the workset again. Newly created work is cached locally, and after the entire chunk is processed, this work is added to the global workset. The chunk size can be fixed statically or determined dynamically.

Wasted work reduction: The amount of wasted (and aborted) work depends on the order in which the active elements are processed. However, there is a tradeoff between the cost of reducing such useless work and the overhead incurred by enforcing a processing order. For example, FIFO order eliminates the wasted work on one thread for BFS, but for multiple threads it is costly to enforce strict FIFO order. Therefore, a good tradeoff is to maintain a loose FIFO order in which all threads remove active elements from one end and add to the other. In case of SSSP, we should process the nodes in order of increasing weight to eliminate the wasted work. However, using a priority queue to implement the workset increases the access time of the workset beyond the benefit of reducing the amount of wasted work.

Removing undo information and conflict detection: SSSP and BFS do not require a conflict detection mechanism because node values decrease monotonically. Conflict detection is needed in DES, but we can apply the *One-Shot* optimization [24] because the neighborhood of every station is statically known to be its outgoing neighbors. A One-Shot implementation first locks the entire neighborhood before making any updates. Thus, an activity will either abort before it made any changes or it is guaranteed to complete successfully. This technique reduces the overhead of the runtime by eliminating the need to keep undo information for the changes made to the graph.

4.2 Ordered algorithms

For ordered algorithms, the main overheads arise from the implementation of ordering constraints.

Commit queue: The executed activities cannot commit until they become the highest priority activities in the system. Therefore, the runtime must keep a record of the executed activities until they commit. A data structure called *Commit Queue* is one way to maintain this information. The Commit Queue must be ordered by the priority of the activities, *i.e.*, the priority of the active element corresponding to an activity. Newly scheduled activities may conflict with already executed activities. Thus, ordered implementations tend to have higher abort ratios.

Sequential commit: Ordered activities can execute in parallel, but they must commit sequentially to ensure correctness. An activity can only commit if it has the highest priority among the executed activities in the commit queue and the pending activities in the workset. If the amount of work per activity is small, as is the case with the ordered algorithms we consider, committing sequentially becomes the main bottleneck. The time to execute an activity needs to be at least $P - 1$ times longer than the time taken by a commit action to keep P threads busy. Note that in unordered algorithms commits can occur in parallel.

Priority-ordered workset: The runtime system needs to know the highest priority element among the pending active elements for two reasons: (i) an executed activity can be committed only if it has highest priority among pending and executing/executed activities, and (ii) to ensure global progress, the highest priority element in the ordered workset needs to be scheduled for execution. Accessing priority queues is usually more expensive than accessing unordered worksets.

In addition, many optimizations that are legal for unordered algorithms are not legal for ordered algorithms. For example, One-Shot optimization cannot be used with ordered algorithms because an activity can be aborted at any point before it is committed, even if it has completed execution and is merely waiting to get to the head of the commit queue.

5. Performance Results

We studied the performance and scaling of our implementations on a Sun UltraSPARC T2 (Niagara 2) Server. This machine has an 8-core processor and 32 GB of main memory. Each core supports 8-way multithreading and has an 8 kB L1 data cache and a 16 kB L1 instruction cache. The 8 cores share a 4 MB L2 cache. We used Sun's 64-bit Hot Spot Java Runtime Environment version 1.6 for our tests. The runtimes reported are the median of five runs in the same JVM instance.

We were unable to obtain a speedup for any of the ordered algorithms discussed in Section 3, so we do not present results for them.

5.1 Breadth-First Search

Figure 18 shows the speedups for unordered BFS. We used two randomly generated undirected graphs as input. The large input contains 8 million nodes and 73.6 million edges. The small input contains 4 million nodes and 36.8 million edges. The reported speedups are relative to the ordered sequential algorithm, which uses a FIFO queue. The figure presents results for two implementations:

Fixpoint: The unordered fixpoint code implements the algorithm discussed in Section 3.1 and includes work chunking with a chunk size of 1024 elements. The workset is implemented using a concurrent FIFO queue.

Wavefront: The wavefront code implements a commonly used algorithm that exploits parallelism within a level [3, 22, 34]. It uses two worksets, a current workset containing all nodes at the current level in the graph and a next workset into which all the nodes from the next level are added. Threads only remove nodes from the current workset and update the levels of the unvisited neighbors, which are added to the next workset. After processing an entire level, the threads synchronize at a barrier. We found that work chunking is also necessary for this algorithm to perform well. We used 1024-element chunks.

Both implementations scale well. The fixpoint implementation reaches a speedup of 22 with 32 threads. The wavefront implementation scales slightly better and reaches a speedup of 27 with 32 threads. Scheduling is successful in reducing the amount of wasted work. The fixpoint implementation performs slightly worse because of the cost of the compare-and-swap (CAS) instructions and the complex termination detection. In contrast, termination detection in wavefront is trivial; the parallel computation is terminated when the workset becomes empty.

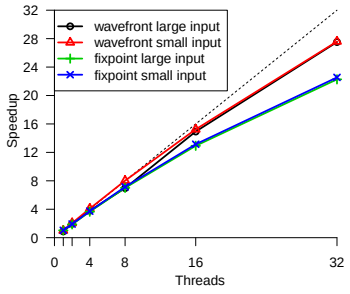


Figure 18. Speedup of the unordered BFS fixpoint and wavefront algorithms; small input = random graph with 4M nodes, 36.8M edges, serial runtime = 21.31 sec.; large input = random graph with 8M nodes 73.6M edges, serial runtime = 45.33 sec.

5.2 Single-Source Shortest Path

Figure 19 shows the speedup results for unordered SSSP. The reported speedups are relative to our sequential implementation of Dijkstra's ordered algorithm, which uses a heap-based priority queue. We used two road networks as inputs [1]. In these inputs, the nodes represent locations and the edge weights represent the distance between adjacent locations. The small input contains the road network of the Western USA and has 6.26 million nodes and 15.1 million edges. The large input contains the road network of the entire USA and has 23.9 million nodes and 57.7 million edges.

Using the FIFO schedule, our Fixpoint SSSP algorithm may perform orders of magnitude of wasted work and is therefore much slower than Dijkstra's algorithm. The Delta Stepping algorithm by Meyer *et al.* [25] essentially offers an efficient schedule for the Fixpoint algorithm. It hashes the nodes into buckets based on their distance label and processes each bucket in parallel in increasing order. The width of a bucket is controlled by a parameter Δ, which governs the amount of parallelism in each bucket and the number of extra updates. For example, a larger Δ value results in more wasted work but also in increased parallelism per bucket. We found that a Δ value of 400,000 and a chunk size of 128 within a bucket works best for our inputs. We obtain a speedup of 4.4 and 4.7 over Dijkstra's algorithm for the small and large inputs, respectively. The limited scaling arises from the fact that these graphs are very sparse and take little time to process; Madduri *et al.* [23] also noticed poor scalability of these inputs on their implementation of the Delta Stepping algorithm.

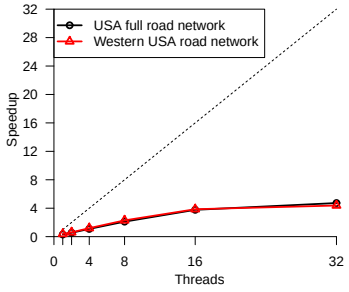


Figure 19. Speedup of the unordered SSSP algorithm; small input = Western USA road network with 6.26M nodes, 15.1M edges, serial runtime = 10.53 sec.; large input = entire USA road network with 23.9M nodes 57.7M edges, serial runtime = 45 sec.

5.3 Discrete-Event Simulation

We implemented the unordered version of the DES algorithm to model logic circuits. We again used two inputs. The small input comprises a 10×10 tree multiplier circuit, which results in 11.68 million events. The large input comprises a 64-bit Kogge-Stone adder circuit, which results in 89.6 million events. The speedup relative to the ordered sequential algorithm is shown in Figure 20. Note that we included the One-Shot optimization described above in our implementation of the unordered algorithm. The small input results in poor scaling, reaching a speedup of 4.5, but the large input scales reasonably and reaches a speedup of 11.3 with 32 threads.

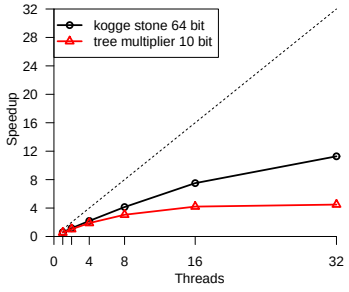


Figure 20. Speedup of the unordered DES algorithm; small input = 10×10 bit tree multiplier, serial runtime = 77.3 sec.; large input = 64-bit Kogge-Stone adder, serial runtime = 409 sec.

5.4 MST Boruvka

Figure 21 shows the speedup results for Boruvka's MST algorithm compared to a sequential implementation of Kruskal's algorithm. We used two inputs for our experiments: The small input is a two dimensional grid of size 1000×1000, where the edges are assigned uniformly random integer weights between 0 and 100. The large input is the road network for Western USA, with 6.26 million nodes and 15.1 million edges. The parallel implementation of Boruvka is slightly different from the pseudocode described in Figure 16. The foreach loop is divided into two phases: (i) *Match* and (ii) *Merge*.

In the Match phase, each node tries to match itself to an unmatched neighbor node on its lightest outgoing edge, thus forming a pair. In the Merge phase, the pair of matched nodes is merged into one bigger component by contracting the selected edge. All the threads either execute the match or the merge phase simultaneously, which eliminates the need for conflict detection and enables the work-chunking optimization. Initially, a large chunk size is chosen; as the execution proceeds, the chunk size shrinks because the number of participating nodes decreases due to edge contractions. The edge contractions have two kinds of effects on the execution:(i) the number of outgoing neighbors of a node increases rapidly, which makes Match and Merge expensive, and (ii) the number of active nodes decreases, limiting the available parallelism towards the end and hence the speedups (*cf.* Figure 17). With 32 threads, we achieve a speedup of 2.7 on the small input and 11.9 on the large input.

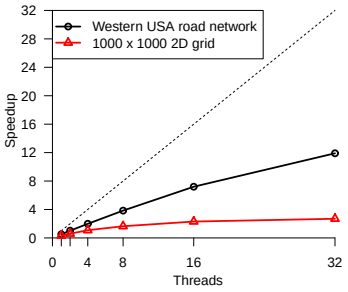


Figure 21. Speedup of Boruvka's MST algorithm; small input = 1000 × 1000 2D grid, serial runtime = 10.93 sec.; large input = Western USA road network with 6.26M nodes, 15.1M edges, serial runtime = 47.99 sec.

5.5 Wasted work results

In this subsection, we investigate the amount of wasted work executed by our implementations of the unordered algorithms. We used the iteration count of the *foreach* loop as a measure of the work performed. While algorithm specific metrics may be more meaningful, *e.g.*, the number of updates to a node label in BFS and SSSP, the number of iterations correlates with such metrics and applies uniformly to all algorithms considered. Figure 22 shows the number of iterations executed by the unordered implementation relative to the sequential ordered implementation, which does not perform any wasted work. We used the large input for each benchmark to obtain these results. In the BFS Fixpoint and wavefront algorithms, the amount of wasted work is negligible. The one-thread runs of the SSSP Fixpoint algorithm with the Delta Stepping schedule already perform more than 5 times the amount of required work, but the value of the Δ chosen delivers the best scalability for these inputs. Interestingly, the wasted work decreases slightly with the number of threads; due to non-determinism of the parallel execution, the threads deviate slightly from the suggested schedule, which coincidentally results in a more efficient schedule.

In case of DES, the ordered algorithm uses different active elements than the unordered algorithm. Therefore, we plotted the ratio of the number of iterations executed over the number of iterations executed by the one-thread run of the unordered algorithm. Hence, the extra iterations are the iterations aborted due to conflicts, which increase with the number of threads. However, due to the One-Shot optimization, these aborted iterations are cheap and do not significantly hurt parallel performance. Note that the serial unordered implementation performs some wasted work equal to the number of NULL messages processed, which are absent in the ordered algorithm. For Boruvka, we plot the ratio of iterations performed by our parallel implementation to the number of iterations of Kruskal's algorithm. Again, the two algorithms use different types of active elements in the workset. The wasted work represents the number of unsuccessful attempts by a thread to match a node with its neighbor on the lightest outgoing edge (*cf.* Section 5.4). Here, the wasted work increases slightly with the number of threads.

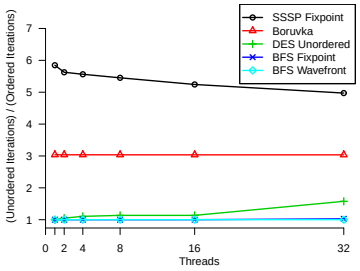


Figure 22. Ratio of iterations performed by the parallel unordered over the sequential ordered implementations for the small inputs

6. Related and Future Work

In the theory community, parallel algorithms for graph problems have been studied using the PRAM model [17, 19]. With the exception of one variant – the so-called Arbitrary CRCW PRAM [30] the PRAM model follows serial semantics, provided (in some PRAM variants) that programmers ensure that multiple processors do not write to the same memory location in a given step. The PRAM model is synchronous and all processors execute instructions in lock-step, so synchronization is inexpensive. In contrast, the Galois execution model is asynchronous because we do not assume anything about the relative speeds of processors. The Explicit Multi-threading (XMT) model [32] is an asynchronous extension of the Arbitrary CRCW PRAM variant that exploits independence of order (IOS) semantics in the explicitly parallel sections of the program. Specialized hardware is used to reduce the overhead of creating and managing small threads [33]. In contrast to PRAM and XMT, there are no explicitly parallel constructs in Galois programs. However, the Galois execution model has to contend with small threads as well, and for reducing the overhead of these, the XMT hardware assist would be useful.

There are many parallel implementations of the four irregular problems considered in this paper, but to the best of our knowledge, no previous work has tried to extract common abstractions and lessons from them. In addition to the work cited earlier, we mention some of the most relevant work here. Edmonds *et al.* [13] describe a single-source shortest-path implementation that partitions the graph and employs a separate priority queue for each partition. Their implementation relies on the monotonically decreasing nature of the node values. For SSSP, the issue of wasted work has also been studied. Meyer *et al.* [25] proposed the unordered *Delta Stepping* algorithm, which allows trading off work efficiency for parallelism (and hence scalability) by controlling the Δ parameter *cf.* Section 5.2. Madduri *et al.* [23] describe an efficient implementation of the Delta Stepping algorithm. Bader and Cong [2] and Chung and Condon [11] have discussed parallel implementations of Boruvka's algorithm. Our implementation of Boruvka borrows ideas from this work. *Chaotic relaxation* is an unordered algorithm for iteratively solving linear systems [10].

A common parallelization approach for breadth-first search is a level-based implementation [3, 22, 35], in which all the nodes at a specific level in the graph are processed in parallel. As the neighbors of the nodes in the next level are all assigned the same value, the concurrent updates do not have to be atomic. It would be interesting to design a static analysis that could determine this automatically.

While tradeoffs between unordered and ordered algorithms have been studied in the past, it is not the case that ordered irregular

algorithms are necessarily inferior to their unordered counterparts. For example, in the discrete event simulation community, many people use the Time Warp algorithm [18], which is ordered, instead of the Chandy-Misra algorithm [8], which is unordered. Since most ordered algorithms do less work than their unordered counterparts, they may become competitive if we can reduce the overheads of implementing ordering, thus highlighting the need for much better system support for speculative execution of ordered algorithms.

7. Conclusions

This paper studied the theoretically available and practically exploitable amorphous data-parallelism in several ordered and unordered graph algorithms. We found that ordered algorithms had significant available parallelism, with at least ten parallel activities on average even in the worst case. However, this parallelism is hard to exploit because of short loop bodies and the practical difficulties of implementing ordering efficiently. Unordered algorithms have more amorphous data-parallelism and shorter critical paths than ordered algorithms, but they often perform wasted work. Hence, there is a tradeoff between work efficiency and scalability when choosing between ordered and unordered algorithms. Nevertheless, the unordered algorithms are easier to optimize and schedule, which leads to efficient parallel implementations. Hence, our recommendation for parallelization is to convert ordered into unordered algorithms when possible, and control the amount of wasted work by proper scheduling.

References

[1] 9th dimacs implementation challenge - shortest paths. `http://www.dis.uniroma1.it/~challenge9/links.shtml`.

[2] 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.

[3] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics, 16(1)*, pages 87–90, 1958.

[5] Gianfranco Bilardi, Paolo D'Alberto, and Alexandru Nicolau. Fractal matrix multiplication: A case study on portability of cache performance. In *Algorithm Engineering*, pages 26–38, 2001.

[6] Gianfranco Bilardi, Afonso Ferreira, Reinhard Lüling, and José D. P. Rolim, editors. *Solving Irregularly Structured Problems in Parallel, 4th International Symposium, IRREGULAR '97, Paderborn, Germany, June 12-13, 1997, Proceedings*, volume 1253 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] Brian D. Carlstrom, Austen McDonald, Michael Carbin, Christos Kozyrakis, and Kunle Olukotun. Transactional collection classes. In *PPOPP*, pages 56–67, 2007.

[8] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Eng.*, 5(5):440–452, 1979.

[9] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[10] D. Chazan and W. Miranker. Chaotic relaxation. *Linear algebra*, 2(199-222), 1969.

[11] Sun Chung and Anne Condon. Parallel implementation of boruvka's minimum spanning tree algorithm. *Parallel Processing Symposium, International*, 0:302, 1996.

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

[13] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-source shortest paths with the parallel boost graph library. In *9th DIMACS Implementation Challenge - Shortest Paths*, November, 2006.

[14] Paul Feautrier. Some efficient solutions to the affine scheduling problem: One dimensional time. *International Journal of Parallel Programming*, October 1992.

[15] Jr. Ford, L. R. Network flow theory. Technical Report Report P-923, The Rand Corporation, Santa Monica, Cal, 1956.

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

[17] Joseph Ja´ Ja´ . *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1992.

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

[19] Jorg Keller, Cristoph Kessler, and Jesper Traff. *Practical PRAM Programming*. Wiley-Interscience, New York, 2001.

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

[21] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *PPoPP*, pages 3–14, 2009.

[22] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, pages 303–314, 2010.

[23] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *9th DIMACS Implementation Challenge - Shortest Paths*, November, 2006.

[24] Mario Mendez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP*, 2010.

[25] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *J. Algs. 49(1)*, pages 114–152, 2003.

[26] http://www.openmp.org/.

[27] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[28] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[29] Lawrence Rauchwerger and David 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.

[30] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[31] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers, 1993.

[32] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA*, pages 140–151, 1998.

[33] Xingzhi Wen and Uzi Vishkin. Fpga-based prototype of a pram-on-chip processor. In *Conf. Computing Frontiers*, pages 55–66, 2008.

[34] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[35] Yang Zhang and Eric A. Hansen. Parallel breadth-first heuristic search on a shared memory architecture. In *AAAI-06 Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, July 2006.