Synthesizing Concurrent Graph Data Structures: A Case Study

R. Manevich, R. Kaleem, and K. Pingali

The University of Texas as Austin,
roman@ices.utexas.edu, {rashid,pingali}@cs.utexas.edu

Abstract. In this paper, we describe our experience with Autograph — a compiler for synthesizing concurrent graph data structure implementations in Java. The input to Autograph is a high-level declarative specification of an abstract data type (ADT), expressed in terms of relations and relational operations. The output is a concurrent implementation of that ADT with conflict detection and rollback baked in. Autograph also provides application programmers with a high-level constraint language for tuning the performance of the generated implementation.
We present a case study in which we use Autograph to synthesize a parallel graph data structure for a parallel implementation of Boruvka's minimal spanning tree algorithm taken from the Lonestar benchmark suite. In particular, we discuss a number of techniques used to specialize the graph data structure implementation for the application and the performance benefit from these specializations. Our results show that these techniques improve overall performance by 38% when compared to a library implementation in the Lonestar benchmark suite, and produce an implementation that runs as fast as a handwritten version of this benchmark that uses fine-grain locking.

1 Introduction

One of the more formidable challenges in parallel programming is the implementation of efficient parallel data structures. An easy way to build a parallel data structure is to use a single lock but this usually comes in the way of getting good parallel performance; conversely, fine-grain locking may give good performance but ensuring correctness of such implementations is usually very difficult. The implementation of efficient parallel data structures is even more complex in systems that use optimistic parallelization, such as the Galois system [5] and boosted transactional memory systems [8]. In these systems, the ADT implementation is responsible for detecting conflicts between concurrently executing computations, and for handling a conflict by rolling back some of the conflicting computations so that others can make progress. There are many approaches to detecting conflicts, such as commutativity conditions and abstract locks, and many ways of handling conflicts, such as undo actions, but all of them complicate the job of the data structure implementor substantially.

At present, these problems are addressed by hiring expert programmers to implement libraries of parallel data structures. These libraries usually have just a

small number of very generic data structure implementations; for example, both MTGL [1] and the parallel Boost graph library [6] provide only a compressed sparse row (CSR) representation for immutable graphs, and an adjacency list representation for mutable graphs. There are two disadvantages: not only are there very few choices but the few available implementations cannot be optimized for a particular application or platform. For example, if node or edge data in a graph is not shared, this data can be inlined into the node or edge objects, which gives better performance. Similarly, structural information that is not needed for a particular application can be eliminated from the graph, leading to a smaller data foot-print and better cache performance. These kinds of data structure optimizations can speed up the end-to-end performance of some applications substantially as we show in this paper, but they are not possible with libraries.

One attractive solution is to synthesize concurrent data structure implementations from high level abstract specifications. If application programmers can guide this synthesis, they can also produce concurrent data structures tuned for particular applications, without having to write explicitly parallel code.

To this purpose, we developed Autograph [12] — the first compiler that can synthesize efficient, safe, parallel implementations of a large class of complex pointer-based data structures implementations from high-level specifications. We have used Autograph in the context of the Galois system [5] to synthesize graph data structures for parallel graph benchmarks from the Lonestar benchmark suite [10]. The benchmarks using the synthesized graphs have usually achieved comparable performance to the original benchmarks that used graph data structures taken from the Galois data structures library; in some cases, the synthesized graphs yield substantially better performance [12].

In this paper we describe a case study that demonstrates the usage of Autograph for specializing a graph data structure to an algorithm, and demonstrate performance improvements on one specific graph benchmark, the Boruvka algorithm for computing the total weight of a minimal spanning tree (MST) [4]. This algorithm is interesting because (a) it morphs the graph (i.e., changes the set of nodes and edges), and (b) the data structure access patterns have certain characteristics that we exploit to improve performance, as we explain in this paper.

Outline. The rest of the paper is organized as follows. In Section 2, we describe the case study benchmark and provide background on parallelizing graph programs in Galois. In Section 3, we describe the specification of the graph data structure given to our tool for this benchmark. In Section 4, we study the properties of the algorithm with respect to its use of the graph data structure and suggest corresponding techniques to specialize the implementation. In Section 5, we empirically evaluate the suggested specialization techniques for their respective contributions in improving the running time of the benchmark. Our empirical evaluation also shows that using the synthesized graph data structure, we are able to compete with a hand-written implementation of the algorithm using fine-grain locking. Section 6 discusses related work and Section 7 concludes the paper.

```
1  Graph g = ... // Undirected graph with integer edge weights.
2  mstWeight = 0
3  Worklist wl = nodes of g
4  while (wl is not empty) { // In parallel.
5    a = remove arbitrary node from wl
6    if g does not contain a // Removed previously.
7      continue to next iteration
8    lt = null // Neighbor connected by lightest edge.
9    minWeight = infinity
10   foreach edge (a, b) {
11     wt = weight of edge (a, b)
12     if (wt < minWeight)
13       lt = b
14       minWeight = wt
15   }
16   contract edge (a, lt) // Removes lt from the graph.
17   mstWeight = mstWeight + minWeight
18   add node a to wl
19 }
20 return mstWeight
```

Fig. 1. Pseudo-code for computing the MST total weight using Boruvka's algorithm

2 Background

Unlike the more well-known MST algorithms of Prim and Kruskal, Boruvka's algorithm computes the MST bottom-up by iteratively coarsening the graph using edge contractions.

In Figure 1, we see the pseudo-code of a simplified implementation of the algorithm which computes the total weight of the MST in a connected graph. The code first stores each node in its own component (line 3). (Conceptually, each node is a degenerate sub-tree of the final result.) In each iteration, the algorithm selects one graph node, which we call an *active element* [5] out of a worklist (line 5). It then examines all edges incident on that node to find the lightest-weight edge (lines 10–15), and contracts that edge (line 16). The weight of the lightest edge is then added to the total weight of the MST (line 17). This computation is called an *activity*, and the set of graph nodes and edges touched by an activity is called the *neighborhood* of that activity [5].

Figure 2(a) shows an undirected graph (for now, ignore the two sub-figures on the right) with two depicted neighborhoods. For active node `a`, the lightest weight edge is the edge connecting it to node `c`. When this edge is contracted, all edges connected to nodes `a` and `c` must be examined to determine which edges must be kept and which ones must be deleted, so the neighborhood of the activity includes nodes `a`, `b`, `c` and `d`, and the edges between them. Similarly, if the active node is `f`, the neighborhood includes nodes `f` and `e`, as well as the edge between these nodes. The algorithm terminates when the graph has been contracted to a single node.

2.1 Parallelization of Graph Algorithms

Boruvka's algorithm exhibits the following computational, pattern which is also exhibited by many other graph algorithms: i) there are a large number of sites

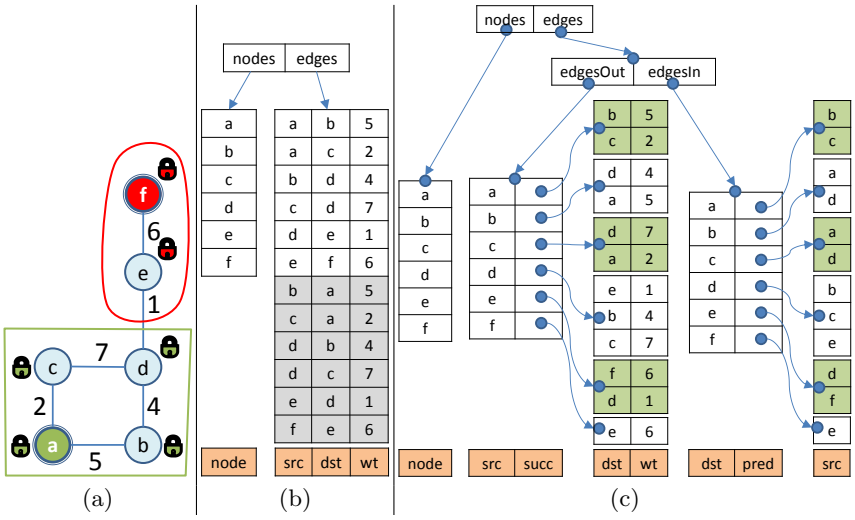

Fig. 2. Graph representations. (a) A weighted graph (with two neighborhoods) arising during a run of the benchmark in Figure 1; (b) A relational representation of the graph; and (c) A decomposed relational representation of the graph

in the graph (active elements) where computations (activities) need to be done; (ii) each activity reads and writes a small region of the overall graph (neighborhood), and (iii) activities can be performed in any order but activities may interfere with other activities since neighborhoods may overlap. Algorithms that exhibit this computational pattern include the preflow-push algorithm for maxflow computation, relaxation-based algorithms for single-source shortest-path (SSSP) computation, mesh generation, refinement and partitioning algorithms, and conservative discrete event simulation [5].

Since neighborhoods are usually small regions of the graph, it is possible to execute many activities concurrently provided the neighborhoods do not overlap. Since activities are created dynamically and the neighborhoods of activities are not known ahead of time in many algorithms, static parallelization is usually not useful for these algorithms. One solution is to execute activities speculatively, rolling back activities when they conflict. If each activity has transactional semantics, the overall parallel execution is correct.

The Galois system is an implementation of these ideas. The programming model enables programmers to write algorithms in *sequential* Java[1], using a construct called the Galois set iterator to iterate over a work-set of active elements. Iterations are executed in parallel by the runtime system. Concurrent data structures used by the application must be instances of library classes provided by the system. *All concurrency control is performed within these library classes.* For

[1] We have recently released a C++ version of Galois but in this paper, we report results from the previous Java release.

example, for graphs, an exclusive lock called an *abstract lock* is associated with each graph node and this lock is acquired by an activity when it touches that node by invoking a graph API method, as shown pictorially in Figure 2(a). If the lock has already been acquired by another activity, a conflict is reported to the runtime system, which rolls back offending activities. To permit rollback, methods that modify the state of the graph also record undo actions that are executed on rollback. Note that two iterations that do not conflict at the ADT level may still access the same memory locations in a concrete implementation of that ADT; therefore, these accesses must also be suitably synchronized.

3 Specifying a Graph Data Structure for Boruvka

Structure:
nodes : **rel**(*node*)
edges : **rel**(*src*, *dst*, *wt*)
FD {*src*, *dst*} → {*wt*}
FK *src* → *node*
FK *dst* → *node*

Methods:
```
edgeExists : contains(src, dst);
findMin : map(src, out dst, out wt){
  if (wt < minWeight) {
    lt = dst;
    minWeight = wt;
  }
};
```
... other methods ...

Decomposition:
root : **List**(*nodes* : **Set**(*node*),
  edges : **List**(
    edgesOut : **Map**(*src*, *succ* : **Map**(*dst*, *wt*))
    edgesIn : **Map**(*dst*, *pred* : **Set**(*src*))
)
)

Tiling:
```
     root tile ListTile
     edges tile ListTile
     nodes tile AttLinkedSet // variant
edgesOut tile AttMap
  edgesIn tile AttMap
      succ tile DualArrayMap // variant
      pred tile ArraySet     // variant
```

Fig. 3. A specification of the graph data type for the running example

In this section, we use the specification of the graph for the running example, shown in Figure 3, to describe the essential features of Autograph. There are four parts to the specification.

Structure A graph is a pair of relations called *nodes* and *edges*, in which the *nodes* relation has a single attribute (representing the node), and the *edges* relation has three attributes corresponding to the source node (*src*), destination node (*dst*), and weight of an edge (*wt*). Note that each undirected edge of the graph is represented in the edge relation by two tuples, one in either direction. Figure 2(b) shows how an instance graph (in Figure 2(a)) is represented in terms of these relations. Autograph generates an implementation of these relations.

A *functional dependency* constraint (**FD**) from *src* and *dst* to *wt* expresses the fact that a unique weight must exist for each edge (represented by a pair

of *src* and *dst*). Autograph generates code to perform a runtime check ensuring that tuples added to the *edges* relation do not violate the functional dependency.

A *foreign-key constraint* (**FK**) expresses the fact that the source and destination of an edge must be nodes, and must therefore be present in the *nodes* relation. Autograph generates code to maintain the foreign-key constraints by using cascading deletions: when a node is removed from the *nodes* relation, tuples in the *edges* relation containing the node value as either *src* or *dst* are removed, effectively removing all edges incident to the node.

Methods The graph methods used in the application program constitute the API for the synthesized data structure. Each of these API methods is described in terms of relational operations on the *nodes* and *edges* relations. These relations are assumed to be endowed with operations for adding and removing tuples, iterating over tuples, etc., and the API methods are described in terms of these operations. We discuss two of the methods: `edgeExists` accepts values for *src* and *dst* and checks whether *edges* contains a (unique) tuple with these values using the `contains` operation; `findMin` accepts a value for *src* and uses the `map` operation to iterate over tuples with that value, where in each iteration it passes the values of *dst* and *wt* (by use of the **out** keyword) to user-given code. Intuitively, this method scans the edges incident to a given node, implementing the loop in lines 10–15 in Figure 1.

Decomposition One disadvantage of the straightforward representation of the *edges* relation is that to find all the outgoing edges of a given node (e.g., in `findMin`), it is necessary to scan the entire relation. Although the code for doing this is generated automatically by the Autograph compiler, it can slow down graph applications like Boruvka that require efficient access to the neighbors of a given node. To address this issue, Autograph allows refining the relations by a *decomposition expression*, which defines how relations may be composed out of sub-relations. Thus, decomposition allows specializing the data structure implementation for the access patterns defined by the operations. Here, **root** represents the top-level data structure, **List** represents a (fixed) list of sub-relations, **Set** represents a relation by a set of tuples, and **Map** represents a relation with a functional dependency between its left sub-expression and right sub-expression.

The decomposition expression in Figure 3 does not decompose the *nodes* relation. The *edges* relation is broken down into two sub-relations: *edgesOut* and *edgesIn*, where *edgesOut* is further decomposed into a sub-relation over *src* and a sub-relation mapping each *dst* and *wt* given a value of *src*; and *edgesIn* is symmetric to *edgesOut* except that it does not keep *wt*. Figure 2(c) shows the result of decomposing the relational representation in Figure 2(b).

This representation provides efficient access to the neighbors of a given node *a*: since there is only one instance of each *src* in the *edgesOut* relation, we can just look up the first instance of *a* and use the associated *succ* relation to access

all of its neighbors. The relation *edgesIn* allows efficient removal of the edges incident to a node as part of a cascading deletion.

Tiling Autograph allows assigning concrete implementations to sub-relations from a repository of data structure implementations, which we call *tiles*.

In the example, `ListTile` implements a list of relations by a class that inlines the fields of the sub-relation implementations; `AttLinkedSet` implements a set by threading a singly-linked list through its members; `AttMap` implements a map relation by inlining the fields of the right-hand side sub-relation inside the object of the left-hand side attribute; `DualArrayMap` implements a map relation between two attributes by a pair of correlated arrays; and `ArraySet` implements a set relation by an array.

Tile assignments followed by `// variant` comments indicate cases where the choice of tiles led to performance improvements over the library data structures.

Automation We make two important notes on the guarantees that Autograph supplies for the generated implementation.

First, the description of the API methods in terms of relational operations does not change when the decomposition expression or tile assignments change. Autograph automatically adjusts the implementation of the operations and generates correct code. Therefore, the application programmer can generate a variety of implementations just by changing the decomposition and tiling portions of the specification.

Second, the specification does not contain any constructs related to parallelism. Autograph automatically generates implementations that ensure linearizability [9] and correct transactional behavior, including conflict detection and undo actions.

4 Specializing the Implementation

In this section, we describe the techniques used in Autograph to generate efficient code.

4.1 A Baseline Implementation

To establish a baseline, we start with a specification that mimics, as close as possible, the graph data structure found in the Galois library, as explained next.

Figure 4 shows only the relevant parts of the baseline specification, omitting parts that are unchanged relative to the specification in Figure 3.

Methods. The `findMin` method takes an extra argument `func` of type `Lambda`, which maintains auxiliary state needed for finding the weight of the lightest edge and the corresponding neighbor. The code for this type is shown in Figure 5.

Methods:	**Tiling:**
`findMin` : **map**(*src*, **out** *dst*, `Lambda func`) { `func.call(`*`src, dst`*`)` }; `getEdgeWeight` : **get**(*src*, *dst*, **out** *wt*);	*nodes* **tile** `AttArrayLinkedSet` *succ* **tile** `DualArrayListMap` *pred* **tile** `ArrayListSet`

Fig. 4. A baseline specification

```
class FindMinLambda extends Lambda {
    public int minWeight;
    public Node lt;
    public void call(Node src, Node dst) {
        int wt = getEdgeWeight(src, dst);
        if (wt < minWeight) {
            this.minWeight = wt;
            this.lt = dst;
        }
    }
}
```

Fig. 5. Code used by the baseline implementation of `findMin`

Since the API for Galois graph data structures does not enable simultaneous access to both the *dst* and *wt*, a call to `getEdgeWeight` is needed, which we have added to the specification in Figure 4.

Tiling. The `AttArrayLinkedSet` tile used in Figure 3 uses a clever concurrent implementation of a set of attributes, supporting constant time insertion, deletion, and membership testing. In particular, it does not use any explicit synchronization. The implementation uses an array where each index corresponds to a thread. At each index there is doubly-linked list of cells where each attribute is preceded by a dummy cell. In addition, each attribute contains a `contained` flag to indicate whether it is currently in the set. Inserting an attribute is performed by accessing the list corresponding to the executing thread and appending a two-element list containing a dummy cell and the attribute to the head of the list, and setting the `contained` flag of the attribute to `true`. This is thread-safe since only the executing thread can access the list. Deletion is performed by resetting the `contained` flag of the attribute and unlinking it from the list. This is thread-safe, since dummy cells separate attributes ensuring that deletions of different attributes access disjoint memory locations. In addition, Autograph acquires an exclusive abstract lock on the attribute, which ensures that simultaneous methods calls do not operate over a common node.

The `DualArratListMap` uses a pair of `java.util.ArrayList` instances to store a list of nodes (for *dst*) and a list of weights correlated by their indices. An `ArrayList` automatically resizes when new elements are added and uses autoboxing for storing the integer weights. Similarly, `ArratListSet` uses an `ArrayList` to implement a set of attributes.

4.2 Selecting Specialized Tiles

We observe that, after the graph is constructed, the code in the parallel loop accesses the *nodes* relation only for removing nodes (as part of edge contraction). Therefore, the sophistication in `AttArrayLinkedSet` for supporting concurrent insertion is not needed. We use `AttLinkedSet`, which uses a singly-linked list threaded across all members. Removing an attribute from the set is implemented by setting the Boolean (`contained`) field to false. This is safe in a concurrent setting, since the implementation of a containment operation acquires an abstract lock on the attribute, which ensures exclusive access to the attribute object.[2]

In our experiments the input graphs we have used are sparse and remain so throughout the run of the benchmark. Therefore, maintaining array-based implementations for *succ* and *pred* results in high performance as the constant time factors associated with array operations are small. We specialize the implementation by using arrays instead of `ArrayList`'s to avoid the overhead of encapsulating the arrays inside objects. Further, we specialize the implementation of *succ* by using an array of the primitive integer type for the weights, which avoids overheads due to autoboxing.

4.3 Specializing `map` Operations

```
findMin : map(src, out dst, out wt, LE{minWeight, lt} le) {
    if (wt < minWeight) { minWeight = wt; lt = dst; }
}
findMin : fuseLocks;
---
class LE { int minWeight; Node lt; }
```

Fig. 6. Code used in the optimized implementation of `findMin`

We specialize `map` operations in several ways, as we now describe using the `findMin` method as an example. The specialized specification and additional relevant code are shown in Figure 6.

Function Inlining. The default implementation of a `map` operation relies on a function object (`Lambda` types), which is invoked on each iteration (via the `call` method). We allow having code as part of the specification, which is inlined in the implementation of a `map` operation, thus eliminating the overheads associated with function objects.

[2] This attribute object is unlinked when possible by `map` operations, which iterate over the entire set.

Selecting Relevant Output Attributes. The Autograph language allows the user the flexibility to choose which attributes should be passed to body of a `map` operation. This is done by adding the **out** modifier before the attribute name, as shown in Figure 6 for *dst* and *wt*. This avoids the overhead associated with a call to `getEdgeWeight`.

Localizing Auxiliary State Objects. The `findMin` method in Figure 6 uses the `LE` type to maintain in its fields the weight of the minimal edge encountered so far and the corresponding neighbor. The body of the operation accesses the fields of the object `LE` object to update that information. Autographallows specifying a list of fields ({`minWeight, lt`}), which upon entering the `findMin` method are copied to local variables and upon. The body of the operation then accesses the local variables and upon exit these are copied back to the fields of the object. This technique reduces the cost of maintaining auxiliary information by reducing the number of memory accesses to a constant.

Loop Fusion. The default implementation of a `map` operation consists of two loops executed in sequence. The first loop acquires abstract locks on the nodes accessed by the operation and the second loop applies the body of the operation to the selected attributes. The directive `findMin : fuseLocks` instructs Autograph to fuse the two loops into a single one. The fused loop executes the lock acquisition followed by the body of the operation on each iteration, which increases memory locality. Fusing is possible in cases where the operation's body does not modify the graph, which is the case of `findMin`[3].

5 Evaluation

In this section, we present an empirical evaluation comparing the performance of the Boruvka benchmark with different data structure implementations and also to a hand-written version using fine-grain locking.

5.1 Experimental Setting

We use a machine with four quad-core 2.7GHz AMD Opteron 8384 (Shanghai) processors. Each core has a 64 KB L1 cache and a 512 KB L2 cache. Each processor has a 6 MB L3 cache that is shared among the cores. The system has 80GB of shared main memory. The software environment is Ubuntu Linux version 10.04.1 LTS, with Java HotSpot 64-bit Server virtual machine version 1.6.0. Each variant of the benchmark was run 4 times in the same instance of the JVM, and executed with the maximum heap size to avoid garbage collection. The first run is dropped to account for the overhead of JIT compilation. We report a mean of the remaining three runs.

[3] Fusing is possible for operations that modify the graph as well, but at the cost of unnecessary increases to the number of undo actions.

We use as input the DIMACS map for the road network of the Western USA (with 6,262,104 nodes and 15,248,146 edges). The graph is quite sparse, (the average degree of a node is 2.4). During the run of the benchmark, the average degree remains low (4.9 and 15 at most).

5.2 Performance Results

Figure 7 compares the running times obtained with:

Lib The generic (library) graph data structure supplied with Galois.
Base The graph synthesized from the baseline specification (Section 4.1).
Opt The graph synthesized using all specializations described earlier (Section 4.2 and Section 4.3).
FGL A hand-written implementation of the benchmark using fine-grain locking. For each node, the edges are stored in increasing order. This makes accessing the lightest weight edge efficient.

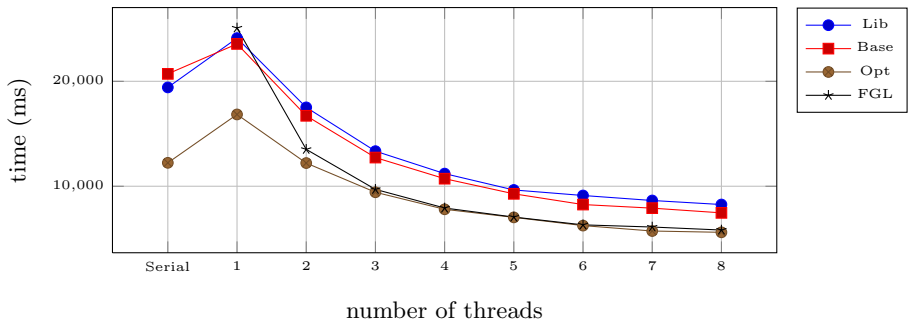

Fig. 7. Absolute running times for variants of the Boruvka benchmark

The plot shows that the Lonestar takes approximately 24s on one thread and 8.2s on 8 threads. The optimized version takes 16s on one thread and 5.6s on 8 threads. The fine-grain locking version takes 25s on one thread and 5.8s on 8 threads.

Figure 8 shows the abort ratios yielded with the different graph data structures.

In this application, the number of committed iterations is equal to the number of nodes in the graph, so it is the same for all versions. Abort ratios increase slightly as the number of threads is increased but they are quite small for all versions. Therefore, the differences in the end-to-end running times can be attributed mainly to the differences in the execution times per iteration.

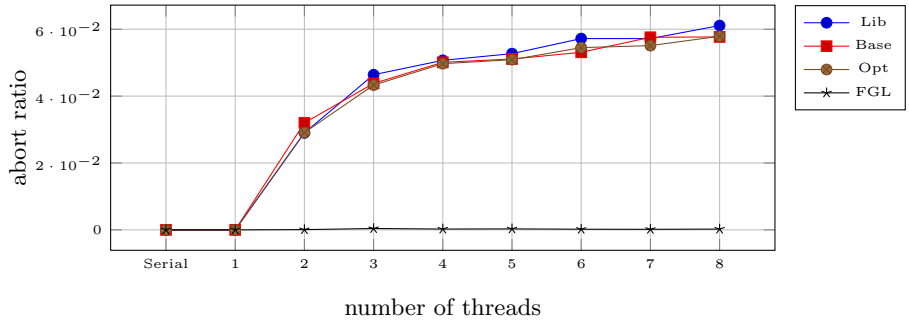

Fig. 8. Abort ratios for Boruvka variants

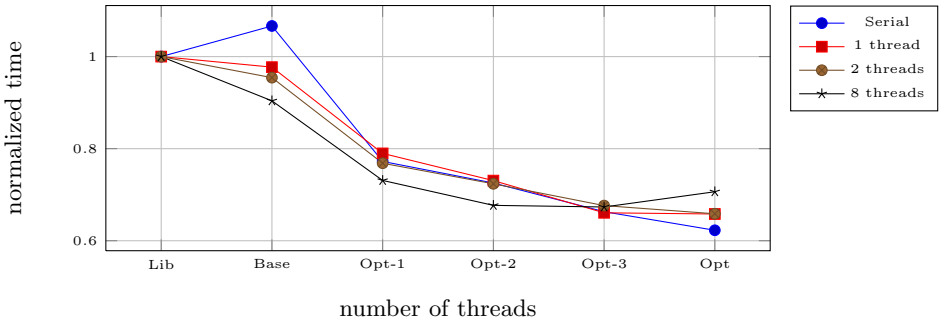

Fig. 9. Running times relative to 'Lib', per thread execution

5.3 Impact of Optimizations

Figure 9 shows normalized running times for variants of the data structure:

Lib The generic (library) graph data structure supplied with Galois.
Base The graph synthesized from the baseline specification, as described in the previous section.
Opt-1 The graph synthesized by starting with the baseline specification and selecting specialized tiles (Section 4.2).
Opt-2 The graph synthesized by starting with Opt-1 and applying function inlining and selection of relevant attributes (described in Section 4.3).
Opt-3 The graph synthesized by starting with Opt-2 and applying localization of auxiliary state objects (described in Section 4.3)
Opt The graph synthesized by starting with Opt-3 and applying loop fusion (described in Section 4.3).

The most significant improvement is obtained from selecting specialized tiles. Next in significance is the application of function inlining and selecting relevant attributes. Auxiliary object localization and loop fusion have positive contribution for the serial run and for a low number of threads, but a slight negative effect for 8 threads. This is primarily because of increased abort ratios (0.0578 for `Opt-3` and 0.0615 for `AGOpt`).

6 Related Work

An important step in data structure synthesis was taken in 1993 by Cohen and Campbell, who built a compiler that synthesized sequential data structure implementations from relational specifications [3]. Their language allows vertical decompositions into sets, maps, and tuples, and uses annotations to assign implementations to each sub-expression.

Beyer [2] developed Crocopat, which is a relational language built around binary decision diagrams (BDDs). Lhoták and Hendren [11] extended the Java programming language by adding direct support for relations and relational operations represented by BDDs. These tools are geared toward representation and manipulation of very large relations.

More recently, Hawkins et al. [7] extended the work of Cohen and Campbell to allow more sophisticated decompositions including sharing of sub-relations. Their compiler also auto-tunes implementations by enumerating decompositions and evaluating the performance on sample inputs.

None of the works mentioned above produces concurrent data structure implementations.

7 Conclusions

The Autograph system described in this paper permits the synthesis of efficient concurrent data structures that are competitive in performance with handwritten structures for irregular graph algorithms such as Boruvka. We have also shown that data structures optimized for particular algorithms can be obtained using the Autograph system. The flexibility provided by Autograph enables developers to quickly experiment with a wide variety of variants of a data structure, an activity that would otherwise require substantial manual effort.

Acknowledgements. We thank Donald Nguyen and Andrew Lenharth for useful discussions and for providing the `AttArrayLinkedSet` data structure. We thank Roberto Lublinerman and Swarat Chaudhuri for supplying us the implementation of Boruvka with fine-grained locking.

References

1. J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. *IPDPS*, 0:495, 2007.
2. D. Beyer. Relational programming with crocopat. In *ICSE*, 2006.
3. D. Cohen and N. Campbell. Automating relational operations on data structures. *IEEE Software*, 10:53–60, 1993.
4. D. Eppstein. *Spanning trees and spanners*, pages 425–461. Elsevier, 1999.
5. K. Pingali et al. The TAO of parallelism in algorithms. In *PLDI*, 2011.
6. D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *POOSC*, 2005.
7. P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, pages 38–49, 2011.
8. M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*. ACM, 2008.
9. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
10. M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.
11. O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *PLDI*, 2004.
12. R. Manevich, R. Kaleem, and K. Pingali. Synthesizing concurrent relational data structures. Technical Report TR-2012-17, The University of Texas at Austin, 2012. Available from http://users.ices.utexas.edu/∼roman/TR-2012-17.pdf.