A Parallel Version of Tarjan's Algorithm

Alexander Zabrodskiy

A dissertation submitted for the degree of
MSc. in Computer Science

Pembroke College
Oxford University
United Kingdom
Trinity Term 2017

Abstract

We build upon the work of [3] to produce a parallelization of Tarjan's algorithm that is fully lock-free while there are more tasks than threads. The majority of the implementation uses wait-free protocals such as our deadlock detection system. Further, we contribute memory recycling techniques which approximately halve the memory footprint and reduce execution time by about a third. Overall, the implementation features about a 4.89 fold speed-up over the sequential algorithm on our eight-core Mac Pro with a 6.28 fold speed-up on large graphs.

Acknowledgments

I would like to sincerely thank Gavin Lowe for suggesting and supervising this exciting project. I also want to thank Gavin Lowe for getting me interested in concurrent programming and giving me the skills to tackle this project with his excellent Concurrent Algorithms and Data Structures course. I would also like to thank my parents and girlfriend for their support and encouragement throughout the year.

Contents

Chapter 1

Introduction

This chapter introduces our motivation and main contributions in the first two sections. In the third section, we provide some definitions and concepts that will be important later in the dissertation. The final section overviews past literature on parallel strongly connected component algorithms.

1.1 Motivation

This dissertation expands on the work done in Lowe's paper *Concurrent Depth-First Search Algorithms Based on Tarjan's Algorithm* [3]. Both this dissertation and [3] are primarily concerned with parallelizing Tarjan's algorithm, a depth-first method of partitioning a graph into strongly connected components (i.e. the maximal subsets S of a graph such that $\forall v_1, v_2 \in S$ there is a directed path from v_1 to v_2).

Finding strongly connected components (SCCs) in a graph is a rather common and fundamental problem in graph theory. For instance, scientists study ecological systems using predator-prey relationship graphs. [1] describes how strongly connected components help ecologists identify energy flow cycles (food chains). The advent of social networks, often modeled by graphs, has introduced another important application of SCCs. Strongly Connected Components can be used to identify which communities and friend groups someone belongs to. Further, SCCs can illustrate the development of those communities [4]. As [4] explains, the SCCs of social network graphs can also predict a person's interests based on the other members of the SCC and be used to study the evolution of communities. SCCs are also useful for

studying internet web crawls using web graphs [11].

Given that Tarjan's algorithm is one of the most well-known and arguably the most efficient approach to finding SCCs, it is not difficult to find an impetus for parallelizing the algorithm. Further, society's growing predilection for big data, the prevalence of multi-core machines, and an increased availability of memory suggest interest in paralyzing popular search algorithms will only increase.

A chief motivation for [3] and this dissertation is developing an improved FDR (Failures Divergences Refinement) model checker for the CSP language. The process algebra CSP is a formal language used to define and analyze concurrent systems composed of independent sequential processes. The FDR model checker is used to verify various properties and assertions about processes written in the CSP language. It turns out a number of calculations performed by FDR closely relate to Tarjan's algorithm. For instance, a key functionality of more recent versions of FDR is the compression of transition graphs of processes. One of these compressions, tau-loop-factor, identifies vertices of a SCC meeting certain conditions and compresses it into one state. Even more important to the model is identifying which vertices are divergent (can lead to unbounded number of tau events). Divergence is important in CSP as it describes livelock, a situation where a system may never terminate. Finding whether a vertex is divergent is equivalent to determining whether it is part of a lasso in the graph restricted of tau transitions. A vertex is part of a lasso if there is a path from it to a member of a cycle. Luckily, a minor alteration on Tarjan's algorithm can be used to find lassos [3].

As variants of Tarjan's algorithm can solve problems relevant to the FDR, parallelizing the algorithm can help the FDR handle large graphs and scale well on multi-core machines. With this in mind, Lowe [3] presents a concurrent version of Tarjan's algorithm along with variants to find loops and lassos. Additionally, [3] provides a prototype implementation in Scala, a strong statically typed language that is typically compiled to Java Bytecode and run on the JVM.

1.2 Contributions

Next, we describe the dissertation's aims and contributions. We build on the work of [3] and propose an implementation with several improvements.

Our first contribution is developing an implementation that does not use any locking/mutexes. This includes introducing a wait-free protocol to suspend searches in the event of collision and a wait-free protocal to detect and resolve suspension cycles (deadlock). We also implemented a wait-free `BlockedList` data structure to store searches suspended on vertices.

There are several reasons why eliminating locking may help increase scalability. If several threads want to enter a critical section, only the thread holding a lock can make progress. If the thread is descheduled before forfeiting the lock, no thread makes progress until the thread resumes. Thus, locking acts as a sequential bottleneck. Under high contention a CPU utilization of a concurrent program using locking can devolve into that of a sequential one. Using operating system locks could be inefficient due to expensive system/OS calls and context switches. Meanwhile, spin locks naturally lead to wasted CPU resources. Both types of lock may also necessitate a significant amount of memory.

To our knowledge, no lock-free parallelization of Tarjan's algorithm currently exists. [16] defines a method as lock-free "if it guarantees that infinitely often some method call finishes in a finite number of steps". In other words, lock-freedom ensures that at least one thread makes progress regardless of how the threads are scheduled. A contribution of this paper is that each component of the algorithm is lock-free. The algorithm as a whole is lock-free if we assume an infinite execution (executing on an infinitely large graph). We must assume infinite execution because toward the end of the algorithm, there will be less jobs remaining than threads and technically the scheduler could foil progress by scheduling the idle threads. The algorithm would be fully lock-free on finite graphs if we had the idle threads assist the working threads if they stalled, but it is more efficient in practice to just use a `sleep()` system call on the threads.

We offer other improvements as well. For instance, our algorithm employs techniques to efficiently recycle and reuse memory. Memory recycling alone reduces the run-time anywhere by about 25-35% and reduces memory usage by 2 to 4 times for most graphs. Also, we deliver our implementations in C++ rather than Scala. An obvious motivation for this is the desire to eventually incorporate our work into the FDR model checker which is written in C++. Additionally, we benefit from C++ being a faster language than Scala, though the difference is by no means astronomical (see [15] for a benchmarking done by Google). More importantly, C++ offers finer grained control of both memory management and memory barriers which gives lock-

free algorithms more potential to outperform their lock-based counterparts.

Overall, our implementation offers a 4.89 times speedup over the sequential algorithm on our 8-core Mac OS machine including a 6.28 times speedup on large graphs. Running the algorithm on a 16 core Linux machine gives similar average speedups of 4.79 for all graphs and 5.99 for large graphs.

The remainder of this chapter covers related work and introduces some important terms and definitions. Chapter 2 details the sequential version of Tarjan's algorithm while chapter 3 introduces our parallelization and explains how synchronization is achieved in the algorithm's main components. Our memory recycling techniques are presented in chapter 4. The original algorithm is parallelized by running multiple searches concurrently. A significant challenge that arises is handling search collisions; i.e. when multiple searches try to explore the same vertex. Chapter 5 draws on [3] to design a wait-free a protocol for detecting collisions and suspending searches visiting a vertex already owned by another search. Chapter 5 also presents lock-free protocols for detecting and resolving suspension cycles. Chapter 6 gives a lock-free implementation of the blockedList data-type we use to record the searches are blocked on specific vertices. Chapter 7 provides empirical results and benchmarks the implementation.

1.3 Important Definitions and Concepts

Before describing our concurrent Tarjan's algorithm, we will briefly define relevant terminology and describe the standard sequential algorithm. We define the term strongly connected as in the popular algorithms textbook, [1].

Definition 1.1. *Two vertices x and y are **strongly connected** if there are directed paths from x to y and from y to x. A graph is strongly connected if all its vertices are strongly connected.*

Definition 1.2. *A **strongly connected component** (SCC) is a maximal strongly connected subset of a graph. That is, the SCCs of a graph are exactly the equivalence classes formed by partitioning a graph by strong connectivity.*

Example 1.1. *Consider a graph with three vertices: rock, paper, and scissors. Suppose there are edges from rock to scissors, from scissors to paper, and from paper to rock. The set containing all the graph's three vertices is*

its only strongly connected component as there exists a path between any of the graph's vertices. The individual vertices are not SCCs because they are not maximal strongly connected subsets of the graph.

Example 1.2. *Consider a graph $G = \{a, b, c\}$ with edges from a to b, from b to c and from b to a. The strongly connected components of the graph are $\{a, b\}$ and $\{c\}$.*

Example 1.3. *All maximal connected components of an undirected graph are strongly connected components. A connected undirected graph has only one SCC. These two facts are easy to prove from the definitions we provided, and demonstrate that strong connectivity is only an interesting property of directed graphs.*

Strongly connected components can be better understood by considering their relationship with cycles. In fact, a cycle containing all the vertices of an SCC can always be obtained from a strongly connected component provided we permit cycles to contain repeated edges and vertices.

In the previous section, we defined a method to be lock-free if finitely often some method call finishes in a finite number of steps. A stronger non-blocking criteria is wait-freedom, the definition is adapted from [16].

Definition 1.3. *A method is **wait-free** if it guarantees that every call finishes its execution in a finite number of steps. Wait-free data structures and algorithms are composed of only wait-free methods.*

Wait-free is a stronger condition than lock-free as the later guarantees that the system as a whole makes progress irregardless of scheduling while the former guarantees progress for each *thread*.

There is an important relationship between locking (i.e. mutexes, spin-locks) and the lock-free and wait-free progress conditions. If each thread attempts to enter a critical section, one can successfully acquire the lock and become descheduled. No guarantee can be made about system progress since the thread can be descheduled indefinitely. Hence, lock-free and wait-free algorithms cannot rely on locks for synchronization. Instead, they prevent data-races using *atomic* operations provided by the hardware which are guaranteed to occur in one step relative to other threads. For instance, C++ provides the `std::atomic` template to wrap variables/objects to direct the compiler to perform atomic operations on the variable. An atomic integer

can be have its value read, written and incremented in one atomic step, preventing a data-race with other threads. Atomic operations are often slower than their regular counterparts, but usually are faster than the regular counterparts wrapped with a mutex. Many lock-free algorithms rely on this very powerful atomic operation which appears throughout the paper:

Code 1.1

```
1 bool compare_and_exchange(var, expectVal, newVal) : atomic{
2   if(var == expectVal) {
3     var = newVal;
4     return true;
5   }
6   return false;
7 }
```

The function changes a variable's value to `newVal` if and only if its current value is `expectedVal`. The entire function is completed in one step (atomically). The function is useful for synchronization because it only writes to the memory location if the location previously held the expected value (no thread modified it in the meantime). Therefore, it provides a guarantee that a successful write will reflect the up-to-date value. In the event of failure, a thread can reread the value and develop a response based on the new value and perhaps try another `compare_and_exchange()`.

1.4 Related Work

The most efficient sequential SCC algorithms, such as Tarjan's algorithm [2] and the path-based strong component algorithm [6], are based on the depth-first search ordering of vertices. In contrast, most parallel SCC algorithms have strayed from this approach as the computation of depth-first search orderings has been shown to be P-Complete and hence inherently difficult to parallelize [7]. Instead, parallel SCC algorithms tend to use a divide and conquer approach, splitting the problem into components that can be computed in parallel. The Forward-Backward (FB) divide and conquer algorithm given by [8] has been built upon in many subsequent parallel SCC algorithms (e.g. [9][10][11][13]).

The FB can be summarized as follows. Let G be a graph. Define vertex v to be reachable from vertex u if there is a directed path from u to v following the graph's edges. The algorithm begins by selecting a pivot vertex v and

identifying the sets $F(G, v)$ and $B(G, v)$ where $F(G, v)$ is the set of vertices in G reachable from v and $B(G, v)$ is the set of vertices that v is reachable from. By definition of SCC, the intersection of $F(G, v)$ and $B(G, v)$ is exactly the SCC containing v. The remainder of the graph is partitioned by the three sets $F(G, v) \setminus SCC(v)$, $B(G, v) \setminus SCC(v)$ and $R(G, v) = G(F(G, v) \cup B(G, v))$. [8] demonstrates that each remaining SCC is contained exclusively in one of the three sets. The algorithm can recursively be applied in parallel to each of the three sets since they can be treated independently. In turn, each of those sets will be partitioned into three independent sets. The work is quickly divided into a sufficient amount of components that can be evaluated concurrently.

As explained by [11], FB's performance suffers on sparse graphs or graphs with many small SCCs partly because for small SCCs the majority of the graph is in the remainder set $R(G, v)$, resulting in a poor division of work. Further, the algorithm can have a high recursion depth on graphs with many SCCs resulting in overhead such as new tasks repeatedly being assigned to threads. Many subsequent papers aim to address these weaknesses. McLendon et al. enhanced the algorithm by adding a trimming step to remove trivial SCCs [10]. After each FB iteration, [10] trims out the vertices with an in-degree or out-degree of zero (trivial SCCs by definition) from the three result sets. The trimming step in turn causes more vertices to have an in-degree/out-degree of zero, so the process can be repeated iteratively.

S. Orzan presents a different parallel SCC algorithm [12]. Orzan's coloring algorithm begins by assigning each vertex a unique integer label (color). The colors are then propagated along the edges from every vertex. Each vertex keeps the maximum of its initial color and the colors propagated to it. Once a steady state is reached, the graph is partitioned by color into multi-components. Clearly, each vertex in one of these components is reachable from the vertex v originating the subgraph's color. v must be the root of an SCC. The algorithm then identifies the vertices in the subgraph from which v is reachable; these must be in the same SCC as v. Afterward, the vertices assigned to SCCs are removed and the process is repeated. The steps propagating colors and finding vertices from which v is reachable are easily parallelized. Contrasting the performance of FB, the coloring algorithm performs well (poorly) on graphs with smaller (larger) SCCs. This is because the algorithm can identify multiple SCCs in each step, but it can take a very long time for the correct color to propagate through a large SCC.

The OBF algorithm [13] aims to improve on FB by partitioning the graph into more than three components while incorporating techniques such

as FB and trimming to identify SCCs. [5] implements many of the above parallel algorithms on GPUs and shows that they perform better than their CPU counterparts.

Hong et al. [9] studies the common characteristics of real-world graphs and designs an algorithm optimized to perform in those situations. For instance, [9] notes that many real-world graphs reflect the small world property; they have one very large SCC and many small ones. Thus, [9] splits up the algorithm into two phases. The first phase is optimized for identifying very large SCCs in parallel. After this phase, the graph is left with many small often disconnected SCCs. The second phase partitions the graph into weakly connected components and identifies the SCCs of each component in parallel. [9], equipped with other improvements, such as parallelized trimming of SCCs up to size two and an efficient work queue, sees significant improvements over the previously discussed methods.

[11] takes a similar approach to [9] in designing a hybrid algorithm that takes advantage of the previous algorithms' strengths at different stages. [11] starts with simple trimming and an iteration of FB to identify the enormous SCC. Afterward the algorithm uses coloring until a small number of vertices remain before switching to the serial Tarjan's algorithm.

Some recent implementations have reexamined the depth-first search approach by parallelizing Tarjan's algorithm, arguably the most efficient serial SCC algorithm. [3], which this paper builds upon, parallelizes Tarjan's algorithm by running several concurrent searches. [3]'s searches do not repeat work; if a search encounters a vertex owned by another search, it suspends until the vertex is finished. A `Suspended` map logs the suspended searches so that deadlocks can be resolved. Renault proposes another parallelization of Tarjan's algorithm [14] that uses a union find structure to store vertices that have been assigned to SCCs. A significant difference between the two Tarjan parallelizations is that searches can repeat work in [14] if they both start working on the same SCC before either completes it and stores it on the union-find data structure. Repeating work may waste resources, but it avoids the overhead of deadlock detection. These two algorithms do not perform much better than the serial algorithm on graphs where a significant portion of the vertices belong to one giant SCC, but achieve significant speed-ups on graphs with many small or medium sized SCCs. Unlike the algorithms based on FB, these algorithms can be used while constructing the graph on the fly, which is useful in applications like FDR.

Chapter 2

The Original Tarjan's Algorithm

2.1 Overview of Tarjan's Algorithm

In this chapter, we give an overview of the sequential Tarjan's algorithm. The original algorithm was introduced in Robert Tarjan's 1972 paper *Depth-First Search and Linear Graph Algorithms* [2]. It has the same asymptotic runtime, $O(V + E)$, as the other two popular approaches for finding SCCs, Kosaraju's algorithm and the path-based strong component algorithm. This bound is asymptotically optimal as all vertices and edges must be considered at least once in order to partition a graph into SCCs.

Robert Tarjan's idea was to label vertices in the order in which they are first visited by the depth-first search. The root of the search is labeled `0`, the next vertex visited is labeled `1` and so on. This integer acts as a sequence number for a vertex and we refer to it as the *index* of a vertex. For each vertex v, we also record the index of the earliest discovered vertex (the one with the lowest index) that was reached via depth first search on v. We refer to this integer as the *rank* of the vertex. Intuitively, the index and rank help identify cycles and SCCs by allowing us to record how far "back" depth-first search on a given vertex can take us.

All the members of a SCC are assigned once they are all identified. We refer to a cell assigned to an SCC as *complete*. A search ignores any complete cells it encounters since the cells the search is currently exploring cannot be in the same SCC since all the cells are assigned simultaneously.

Suppose our search is at vertex v with index i_v and finds an incomplete neighbor of v, w, with index $i_w < i_v$. Since the vertex was discovered earlier than v and is incomplete, w must either be on the current depth-first path to v or there must be a path from w to some vertex c on the depth-first path.

Thus, there must be a path from w to v. Since we just discovered a path from v to w, it must be that v and w are in the same strongly connected component. We record this by updating the rank of v to be at most j. Note that any vertex on the current depth first path to v now has a path to w; thus, the ranks of those vertices are updated to be at most j during backtracking.

2.2 Presenting the Algorithm

For both the sequential and concurrent implementations, each vertex is associated with a corresponding `Cell` object storing its index, rank, status, and other information related to the vertex. When a vertex is first explored, it is assigned an index and a `Cell` object is created for the vertex and pushed to a stack data structure we refer to as the `tarjanStack`. The `tarjanStack` holds the vertices that have been visited, but not yet assigned to a SCC and is naturally sorted by index.

Code 2.1

```
1 struct Cell{
2   Vertex vertex;
3   int index, rank;
4   Status status = NEW_CELL;
5   list unassignedNeighbors;
6   void updateRank(int update) {
7     rank = min(update, rank);
8   }
9 };
```

A cell's status variable is set to `NEW_CELL` when it is first created, `ON_STACK` when it is first visited by the search and placed on the `tarjanStack`, and `COMPLETE` when its vertex has been assigned to a SCC. Each cell object also contains a list of the vertex's neighbors that have not yet been explored from this cell. Given a vertex, we must be able to retrieve the corresponding cell object efficiently. We use a standard library hash table in our implementations to achieve this.

The depth-first nature of the algorithm is typically implemented using the function call stack (i.e. recursion) for backtracking. As in [3]; however, we present an iterative version of the algorithm as it easier to parallelize. For instance, in the concurrent version, a thread suspends a search that tries to explore a cell on another search's `tarjanStack` and works on another search

in the meantime. Suspending a search would require saving the call stack somewhere and perhaps later transferring it to another thread, which could get messy. We use the `controlStack` stack object to emulate the call stack.

The code for the algorithm is presented below. The function `tarjan()` calls `search()` to explore every vertex that has not yet been encountered by a search. A `search(root)` call starts a depth-first search on the root vertex.

Each iteration of `search()`'s main while loop explores a new edge or backtracks when no edges remain from the current vertex. Consider an iteration of the loop exploring an edge from v to w. If w has never been encountered, `claim()` is called (line 19) to add the `Cell` object representing the vertex to the stacks with index and rank equal to `cellCount`. Afterward, `cellCount` is incremented and `initilizeNeighbors()` is called to probe the graph to find the neighboring vertices of the w and assign `Cell` objects to the vertices if necessary. Lines 39-46 implement `claim()`.

If w is already on the stack, the rank of v is updated to be at most that of w. Vertex w can be ignored if it already belongs to a SCC since all the SCC's members have already been identified. When backtracking from v, we update the predecessor's rank to be at most that of all its depth-first descendants to log how far "back" the edge from v to the predecessor takes us (line 26). We define the *root* of an SCC to be the cell/vertex first added to the `TarjanStack`, i.e. the one with the smallest index. When backtracking from a cell c, if `c.rank == c.index` (line 27), c is the root of the SCC containing c and the cells above it (closer to the top) on the `TarjanStack`. The cells are marked complete, removed from the `TarjanStack`, and assigned to the SCC on lines 28-33. Otherwise, if `v.rank` $\neq$ `v.index`, the vertex remains on the `TarjanStack`. This maintains the key invariant that a cell c remains on the stack if there is path from c to a cell earlier on the stack.

Code 2.2

```
1  int cellCount = 0; //Total of cells explored
2  Stack<Cell> tarjanStack;
3  Stack<Cell> controlStack;
4  void tarjan(Graph graph){
5    //Start a search rooted at each vertex that has not yet
       been considered
6    for(vertex in graph)
7      if(vertex not in SCC) search(new Cell(vertex));
8  }
9  void search(Node root){
10   claim(root);
```

```
11    while(controlStack.nonEmpty){
12      Cell curr = controlStack.top;
13      if(curr.hasUnassignedNeighbors()){
14        Cell child = curr->unassignedNeighbors.pop();
15        if(child.status == ON_STACK)
16          curr->updateRank(child->index);
17        else if(!child.status == COMPLETE)
18          //Not yet on stack nor assigned to an SCC
19            claim(child);
20            //so add it to the tarjan stack
21      }
22      else{ //No more neighbors to explore, so backtrack
23        controlStack.pop();
24        //Update the node before curr rank reflecting curr rank
25        if(!controlStack.empty())
26          controlStack.top()->updateRank(curr->rank);
27        if(curr.index == curr.rank)
28          SCC = new Set<Vertex>()
29          do{
30               cell = tarjanStack.pop();
31               cell.status = COMPLETE;
32              SCC.add(cell.vertex);
33          }while(cell != curr);
34      }
35    }
36  }
37  // explores a cell and adds the cell
38  // to the search's tarjanStack
39  void claim(Cell cell){
40    controlStack.push(cell);
41    tarjanStack.push(cell);
42    cell.rank = cell.index = cellCount;
43    cellCount += 1;
44    cell.status = ON_STACK;
45    cell.unassignedNeighbors = initilizeNeighbors(cell);
46  }
```

Definition 2.1. *A cell d is a* ***depth-first descendent*** *of cell c if the depth first search on c reached d and added it to the* `tarjanStack`*.*

The following theorem helps elucidate the reasoning behind how the algorithm assigns cells to SCCs.

Theorem 2.1. *Let c be a cell with* `c.index == c.rank` *after all its neighbors have been considered. c is the root of the SCC containing itself and any cells*

currently above (closer to the stack's top) it on the ***tarjanStack****.*

Proof. Suppose we have such a cell c with c.`index == ` c.`rank`. Cell c is root of a SCC because depth-first search on c did not reach any earlier vertices. Suppose, for contradiction, there exist cells higher in the `TarjanStack`, but in a different SCC than c. Let d be the cell with minimum index among this group of vertices. Since d is higher on the `TarjanStack` than c (d added after c), d must have been discovered in the depth-first search on c. Therefore, there is a path from c to d. By definition of SCC, a path from d to c would yield a contradiction. We have d.`index` $\neq$ d.`rank`, otherwise the cell would have been completed by lines 27-33 when we backtracked from d. This means there must be a path from d to some cell e earlier in the `TarjanStack`. The cell e must be higher in the stack than c, otherwise we would have the impossible relation c.`index` $>$ e.`index` $>=$ d.`rank` $>=$ c.`rank`, where the middle inequality holds because there is a path from d to e, and the rightmost inequality holds because c.`rank` is set to at most d.`rank` sometime after backtracking from d. If there is a path from e to c, the path from d to e to c would result in d being in the same SCC as c, a contradiction. Thus, no path from e to c exists and hence they are in different SCCs. But since e.`index` $<$ d.`index`, this contradicts the assumption that d is the cell with minimum index among the cells higher than c not not in the same SCC as c. □

Next, we will discuss the observations mentioned in [3] because they are useful later in the paper. We will also outline the reasoning behind some of the observations to help the reader grasp the algorithm.

Theorem 2.2. *The following invariants hold after each iteration of the while loop:*

(a) Any cell c nearer to the top of the ***tarjanStack*** *than* ***controlStack.top*** *is in the same SCC as* ***controlStack.top****.*

(b) For any cells c and d on the ***tarjanStack*** *such that d is closer to the stack's top than c, the graph contains a path from c to d.*

*(c) If for cells c, d we have $c.$**rank** $== d.$**index**, all the cells between c and d in the* ***tarjanStack*** *are in the same strongly connected component.*

Proof. (a) Suppose, for contradiction, that c is as described in 2.1, but not in the same SCC as `controlStack.top`. Since c is higher on the `tarjanStack` than `controlStack.top`, cell c was added to the stack after `controlStack.top` and hence is a depth-first descendant of `controlStack.top`. Consider the strongly connected component c resides in; let cell a be its root. Note that a must be a depth first descendant of `controlStack.top` as well; otherwise, a would be in the same SCC as `controlStack.top` and hence c would be in the same SCC as `controlStack.top`. Thus, a must have index greater than that of `controlStack.top`. Since a is the root of the SCC, a will not have edges to cells added earlier to the `tarjanStack`. Hence, a.`index` == a.`rank`. But this means that the elements in the SCC would have been removed from the `tarjanStack` when backtracking reached a and before reaching `controlStack.top`. We have reached a contradiction.

(b) Let c and d be as in Theorem 2.2.(b). Cell d is higher on the stack and hence was explored later than c. Suppose d is not a descendant of c as the other case is trivial. Consider the depth-first path from the root of the search to c that existed before we backtracked from c. These are exactly the cells on the `controlStack` from the root until c. The cell d cannot be one of these cells since they were added to the `tarjanStack` before c. Hence, d must be a descendant of at least one of the cells that were on the `controlStack` when c was explored. Let e be the lowest such cell on the `controlStack`. There must be a path from c to e, otherwise c would have been assigned to a SCC while the search was backtracking toward e (Theorem 2.2.(a)). Of course, there must be a path from e to d as well since d is a depth-first descendant of e. Thus, we have shown that there is a path from c to d.

(c) The explanation for the third part of the theorem uses similar logic to the first two and is omitted.

□

Chapter 3

Parallelizing Tarjan's Algorithm

3.1 Overview

As our design builds on the one described in [3], this paper will explain both implementations so that it is easy to appreciate the similarities and differences. This first section gives an overview of both algorithms and, unless stated otherwise, all material discussed applies to both algorithms.

The previous chapter illustrated how the sequential algorithm carries out a search. The basic idea of both concurrent algorithms is to carry out multiple searches simultaneously, each with their own Tarjan/Control stacks and `cellCount`. Ideally, the searches would independently explore different parts of the graphs, pushing cells they encounter onto their stacks and eventually assigning them to appropriate SCCs. Of course, full independence is impossible and synchronizing the inevitable interactions between the searches is the focus of a large portion of this paper. For instance, the sequential algorithm, [3]'s parallelization, and our own all require cell objects to hold information about each vertex, though the data we must store varies by implementation. Every time a search encounters a vertex, it must be able to retrieve the cell object associated with the search from a map or create one if it does not yet exist. This issue is efficiently solved in the sequential case with any standard library implementation of a hash table.

The issue is more complicated for the concurrent implementations. Consider a scenario where a search discovers a vertex, creates a cell object for it, and places it on its stack and continues with a depth first search on it. Now suppose another search, managed by a different thread, comes across the same vertex. The search must at least be able to ascertain that another thread is working on that cell, otherwise each thread would duplicate all the

others' work. Further, the search must monitor whether the vertex is not yet on any stack (hence can be added to the search's stack), on another search's stack, or assigned to a SCC and therefore can be safely ignored. This indicates that we must have one thread-safe map shared between threads. Due to the nature of depth-first search, searches are constantly considering different vertices and hence the map is used very frequently and acts as the most significant bottleneck in both concurrent implementations. Hence, as addressed in more detail later, we need a map that is not only thread-safe, but efficient; simply wrapping accesses to a non-thread-safe map in locks won't suffice.

Before describing another synchronization issue, we clarify how we encapsulate searches. At a very high level, the sequential algorithm entails the following: select a "root" vertex not yet assigned to an SCC, proceed with depth-first search until all vertices reachable from the root have been assigned, and repeat the process until the graph is completely partitioned into SCCs. When discussing the concurrent algorithms, we refer to an iteration of this process as a search. That is, a search represents the progress made from the moment the root cell is added to the stack until all vertices reachable from the root have been assigned to SCCs. The `Search` class encapsulating searches appears below.

Code 3.1

```
1 class Search{
2   Cell*  cellBlockedOn;
3   Stack<Cell*> tarjanStack
4   Stack<Cell*> controlStack
5   int cellCount;
6   int age;
7   ...
8 }
```

The `Search` class in Lowe's algorithm is similar except that it contains a status variable which we removed the need for, and our version includes an extra *age* variable to assist with memory recycling. Each time a search starts, its `cellCount` is zero and both stacks are empty. Additionally, notice the use of pointers in the code. In C++, pointers are used to represent the address of an object, so the object can be accessed by different data structures and functions without making copies.

Ideally, each thread's searches would independently work on different parts of the graph without interference. In reality, collisions are inevitable; a search will often want to explore a cell that is already owned by another

search. There are at least two ways to resolve this issue. First, we can allow searches to duplicate work; that is, we can allow the second search to add a cell owned by another search to its own stack and perform depth-first search on it as well. One could argue that any method of resolving collisions will be complicated enough that it is most efficient to just allow searches to duplicate work when collisions occur. To avoid duplicating work, both [3] and this paper temporarily suspend searches that encounter a cell owned by another search until the cell is assigned to a SCC. [3] provides some justification for the decision by comparing both approaches and finding the latter more efficient.

One may question why we defined searches as we did instead of simply equipping each thread with its own stacks and `cellCount`. Separating the thread/search definitions allows a thread whose search is suspended to resume another queued up search or start a new one. Otherwise, to ensure utilization of all the machine's cores, we would have to either create a new thread when a search suspends or have many more threads than cores both of which may be inefficient.

Simply suspending a search is insufficient. Consider two searches each suspended on a cell owned by the other search. Neither cell can become complete since it is owned by a suspended search. Therefore, the searches will remain suspended forever in the absence of interference. Of course, cycles made up of more than 2 searches can occur as well. Thus, the algorithm needs a mechanism for detecting and resolving cycles. A thread suspending a search must record the suspension somewhere in shared memory, check if suspending the search created a cycle, and resolve the cycle if necessary. Both concurrent implementations resolve cycles similarly, but have very different ways of logging suspensions and identifying cycles.

Searches are carried out by the `execute()` function whose code is presented in code 3.2 below. The function implements the algorithm's depth-first functionality; each iteration of the while loop explores a new edge of the graph or backtracks. While [3] gives a somewhat different design (i.e. functions are members of different classes, etc.), the two are close enough that we only present our own. The code is deliberately structured so that all synchronization can be done inside the helper functions called by `execute()`.

The concurrent algorithm certainly bears resemblance to the sequential. Notice the similarities between the `execute()` function below and the `search()` function in section 2.2. As before, new cells are pushed to the search's stack while complete cells are ignored since they have already been

assigned to a SCC. If a cell already owned by the search is encountered, the rank of the parent cell is updated to be at no greater than the child's `index` (lines 22-24).

The first major departure is that a search may also encounter cells on another search's stack. Once a search determines an encountered cell is not already on its stack, it calls `claim()` which attempts to claim ownership of the cell by changing the cell's status from `NEW_CELL` to the search's address. This functionality must be thread-safe as only one search should be allowed to claim each cell. Code implementing `claim()` is presented on lines 1-7; a synchronized version appears in the next section. `claim()` returns `CLAIMED`, `COMPLETE`, or `OCCUPIED` depending on whether the cell is successfully claimed, the cell is already complete or the cell has been claimed by another search. If `claim()` returns `OCCUPIED`, the search will call `suspend()` (line 38) to suspend the search until the conflicting cell becomes complete. If `claim()` returns `CLAIMED` the cell is added to both stacks, has its neighbors initialized, and the depth-first search advances to that cell.

The `buildSCC()` function called on line 47 assigns the cells between (inclusive) `tarjanStack.top()` and `curr` to the strongly connected component rooted at `curr`. The functionality is similar to its sequential analogue. One noticeable difference is the call to `unsuspendAll(curr)` on line 59. After a cell is complete, all the searches suspended on the cell need to be resumed. We detail the mechanics of suspension later. To avoid unnecessary synchronization, threads collect SCCs in local containers which are merged at the end of the algorithm. The SCCs are saved in the local collection on the last line of `buildSCC()`.

Code 3.2

```
1  Status claim(Cell* cell, Search* search){
2    if(cell->status == NEW_CELL){
3      //Mark ownership by setting status
4      //as pointer to owning search
5      cell->status = search;
6      return CLAIMED;
7    }
8    //Cell is complete, the search can ignore it
9    if(cell->status == COMPLETE_CELL)
10     return COMPLETE;
11   //If the cell is not new, not complete,
12   //and not on conquerer's stack
13   //(checked before function called)
```

```
14    //it must be on another search's stack
15    return OCCUPIED;
16  }
17  void execute(Search* search){
18    while(search->controlStackNonEmpty()){
19      Cell* curr = search->controlStackTop();
20      if(curr->hasNeighborsLeft()){
21        Cell* child = curr->getBestNeighbor();
22        if(child->onStackOf(search)){
23          curr->rank = promote(child->index);
24          continue; //Go back to start of loop
25        }
26        Status attempt = claim(child, search);
27        //Try to put the Cell on search's stack
28        if(attempt == CLAIMED){
29          search->pushToStacks(child);
30          //Push child to tarjan and control stacks
31          child.index = cellCount;
32          child.rank = cellCount;
33          cellCount += 1;
34          initializeNeighbors(child);
35        }
36        else if(attempt == OCCUPIED){
37          //Break out of loop and exit if need to suspend
38          if(suspend(search, child)) return;
39        }
40      }
41      else{
42        search->controlStack.pop();
43        //update parent's rank when backtracking
44         if(!search->controlStackEmpty())
45            search->controlStack.top()->promote(curr->rank);
46         if(curr->index == curr->rank)
47          buildSCC(search, curr);
48      }
49    }
50    reclaim_memory(search); //reclaim memory if search finishes
51  }
52  void buildSCC(Search* search, Cell* SCC_root){
53    Cell* cell;
54    SCC scc = new SCC;
55    do{
56      cell = search->tarjanStack.pop();
57      scc += cell.vertex;
58      cell.status = COMPLETE;
```

```
59      unsuspendAll(cell);
60    }while(cell != SCC_root);
61    thread.addSCC(scc);
62  }
```

Again, information about vertices is contained in cell objects. [3]'s cell objects have separate variables for the cell's status, the cell's owner, whether the cell is complete, etc. We condense this information into one variable, the cell's status. The status variable takes on the value `NEW_CELL` or `COMPLETE_CELL` in the obvious situations. When the cell is on a search's stack (claimed), its status variable equals the address (pointer) of that search. As illustrated by the next section, condensing the cell's status into one variable allows us to easily use `compare_and_exchange()` operations to safely modify the cell's status/ownership without locks. The updated `Cell` class appears below.

Code 3.3

```
1  struct Cell{ //The cell class for parallel Tarjan's algorthm
2    Vertex vertex;
3    int index, rank;
4    CellStatus status;
5    List<Cell*> neighbors; //successors not yet considered
6    int age;
7    void promote(int updatedRank){
8      rank = min(rank, updatedRank);
9    }
10 }
```

As with a standard depth first search, each edge is considered exactly once. The successors of the vertex represented by a cell be retrieved from a graph. As alluded to earlier, new cell objects cannot simply be assigned each time a vertex is encountered since other searches may have already found the vertex and assigned it a cell object. Thus, we use a dictionary to retrieve the cell object associated with each vertex (or add a cell object if it does not already exist).

The function `initializeNeighbors()` (code presented below) is invoked when a new cell is first added to a search's stack. The function uses the dictionary to create/retrieve the cell's successors. The successors not yet assigned to an SCC are placed onto the cell's `neighbors` list which represents the edges from the new cell yet to be considered.

We do not arbitrarily choose a vertex's neighbor when expanding our depth first search. Rather, to minimize collisions, we iterate through the

list to see if any of the cells are currently unoccupied by another search. This appears to be a minor optimization, but has sizable effects in practice. The improvement is probably due to the nature of depth-first search; considerable time may pass until the occupied cell is again considered by which point it may have become complete. [3] also avoids arbitrarily selecting vertices through a somewhat different procedure for choosing neighbors. The functionality discussed in this paragraph is implemented by the `getBestNeighbor()` method.

Code 3.4

```
1  void initializeNeighbors(Cell* cell){
2    List vertices<Vertex> succs = graph.getNeighbors(cell);
3    for(Vertex succ: succs){
4      //Dictionary retrieves the Cell
5      //associated with the vertex if one exists,
6      //otherwise it creates a new Cell
7      //to map the vertex to
8      //the dictionary must be thread-safe
9      Cell neighbor = dictionary[succ];
10     if(!neighbor->isComplete())
11       cell->neighbors.add(neighbor);
12   }
13 }
```

3.2 Synchronization

This section details how we synchronize access to search and cell objects during calls to `execute()`. The code is structured so that all the synchronization is done in the helper functions called by `execute()`. While [3] uses locking to guard both the `execute()` and `claim()` function, we can forgo locking the `execute()` function and focus on synchronizing only the `claim()` function. This is a common strategy we use to achieve a lock-free implementation. Synchronization with locks is much easier partly because locks guarantee mutual exclusion on as many lines of code as needed while atomics only provide one line of mutual exclusion. Our strategy is to *squeeze* all the synchronization into a line of code here and there that can be performed using atomics.

Let us apply this principle to design a wait-free `claim()` implementation. The code for the updated claim function is presented below. `claim()`

is called after verifying that the cell is not already on the search's stack. First, the function attempts to claim the cell by setting `cell.status == search` using a `compare_and_exchange()`. The `compare_and_exchange()` provides synchronization by ensuring that only one search succeeds in claiming the cell. The `compare_and_exchange()` is only attempted if the read of `cell→status` on line 4 returns `NEW_CELL`. This check does not affect the algorithm's output since the `compare_and_exchange()` carries out the same check, but is included as an optimization since reads are typically faster than `compare_and_exchange()` calls. If the `compare_and_exchange()` fails, the cell be complete or on another search's stack. Lines 8-9 return `COMPLETE` if the cell has been complete. If the cell is not successfully claimed or complete, it must be on another search's stack; hence `OCCUPIED` is returned.

Code 3.5

```
1  Status claim(Cell* cell, Search* search){
2    //if the cell is new, attempt to claim
3    // it with a compare_and_exchange
4    if(cell->status == NEW_CELL &&
5      cell>status.compare_and_exchange(NEW_CELL, search))
6      return CLAIMED;
7    //Cell is complete, the search can ignore it
8    if(cell->status == COMPLETE_CELL)
9      return COMPLETE;
10   //If the cell is not new, not complete,
11   //and not on this search's stack
12   //(checked before function called)
13   //it must be on another search's stack
14   return OCCUPIED;
15 }
```

Notice that the cell may become complete after line 8 and `claim()` may falsely return `OCCUPIED`. We could inspect if the cell is still incomplete after `claim()` returns, but we would face the same issue if the cell became complete after the check. Instead, we delegate the issue to the `suspend()` function. After suspending a search, we check if the conflicting cell has become complete in the meantime. If so, the search is unsuspended. Otherwise, the search will be unsuspended by `unsuspendAll()` later. This is discussed in more detail later. The `claim()` function is clearly wait-free as there are no loops or `GOTO`s and the function does not hold any locks.

We have argued that the `claim()` function is correctly synchronized. Next, we explain that the other modifications to shared variables in

`execute()` are thread-safe. We will prove Corollary 3.1 below in a later chapter.

Corollary 3.1. *A thread T that begins executing a search S will continue to do so until the search suspends or until the search is finished and its memory is recycled. No other thread executes S from the moment T begins executing the search until T finishes executing the search.*

Corollary 3.1 provides mutual exclusion for the `execute()` function. The following observation assures a thread executing a search that neither the search's member variables nor the cells on the search's stack will be modified by other threads. Additionally, the observation details the situations requiring synchronization.

Observation 3.1.

a) The only times a thread executing a search accesses cells not on its search's stack are in the helper functions `initializeNeighbors()`, `suspend()`, and `claim()`.

b) While a search executes, no other thread will modify the search's member variables or the cells that are on the search's `tarjanStack`.

Proof. We give a sketch of the proof since fully rigorous proof is impossible as all of the code has not been presented yet. We start with a). `execute()`'s main loop begins by setting `curr = controlStack.top()`. Thus, `curr` is on the search's stacks and hence the modifications on `curr` on lines 21 and 23 are on a cell on the search's stack. `child` $\rightarrow$ `index` is accessed on line 23, but only if the child cell is on the search's `tarjanStack`. As discussed, if the depth-first successor of `curr` is not on the search's stack, the `claim()` function called on line 26 attempts to add it to the stack. Lines 29-34 access and modify the `child` cell, but the lines are only executed if the cell is successfully added to the `tarjanStack`. Lines 45 and 46 clearly access cells that are the search's `tarjanStack`. The `buildSCC()` function assigns `curr` and all the cells above `curr` on the `tarjanStack` to a new SCC. The observation is not violated since all those cells are owned by the search. The function calls `unsuspendAll()` to unsuspend the searches suspended on the new SCC's cells. The observation holds since only suspended searches are modified.

Given a), we can prove b) by demonstrating that the three functions mentioned in a) do not modify other executing searches or the cells claimed by those searches. A cycle resolution in `suspend()` is the only part of function

that can modify multiple searches and the cells on their stacks. But the searches involved in a cycle are suspended and not being executed by a thread, thus the observation is not violated. As shown above, `claim()` only affects new unclaimed cells. We will demonstrate that `initializeNeighbors()` does not violate the claim below. □

Together Corollary 3.1 and the above observation provide crucial synchronization guarantees. Most importantly, they allow searches to interact with their member variables and the cell's on their stacks without expensive synchronization. Further, the observation states exactly the three areas we need to make thread-safe: `initializeNeighbors()`, `suspend()`, and `claim()`. We have already discussed how to make `claim()` thread-safe. The `suspend()` function is treated in great detail in later chapters. The `initializeNeighbors()` function is called when a cell is added to a search's stack for the first time to populate its list of successors. Since the cell is on the search's stack, modifying the neighbors list is thread-safe by the above observation. Thus, `initializeNeighbors()` is thread-safe as long as the dictionary used to retrieve cell objects is thread-safe.

3.3 Scheduling Threads

We use a similar scheduling pattern to [3], allocating a pool of worker threads to execute searches. Threads are not bound to their searches; a thread that concludes or suspends the search it is working on seeks new work to do. This allows each thread to continuously make progress and ensures all computing resources are utilized. The threads are encapsulated by `Worker` objects. The `Worker` class implements many functions associated with executing searches and stores memory specific to the thread. This thread local memory is very important in our algorithm; for instance, each worker has memory pools to store recycled memory. As another example, `Worker` objects also store the SCCs that its thread finishes.

The `Pending` object contains searches that were previously blocked, but are now safe to resume. After completing a job, a thread will check if there are any previously suspended searches to resume on `Pending`. Otherwise, the thread starts a new search from a cell that has not been explored yet. The `Pending` object can easily be implemented using a stack or queue guarded by locks. Alternatively, one can use a lock-free queue, which is relatively easy to

implement, to achieve the same behavior. The `Pending` object is not accessed as frequently as some of the other data structures in the algorithm and hence the choice has little impact on performance; some basic experiments seemed to verify this. The implementation is designed so that a search is simply started/resumed by calling `execute(Search)`.

Our implementation performs what [3] calls an unrooted search. That is, a list of all the graph's vertices is provided in the beginning. To begin a new search, a thread selects a vertex from the list that no thread has selected yet and probes the hash table to find the corresponding cell. If the cell is complete or owned by another search, the process is repeated until we find a new cell. Then, the thread attempts to claim the cell in a manner similar to the `claim()` method presented earlier. If claiming the cell fails, the process is repeated. Otherwise, the claimed cell becomes the root of the new search. This process ensures that a new search already owns its root so we can avoid the process of aborting a search that fails to claim its root.

Chapter 4

Memory Recycling

This chapter extends the concurrent Tarjan's algorithm by enabling the reuse of objects such as searches and cells. The first section provides motivation while the second section elaborates on how memory recycling is implemented. The final section provides more detail on the reference counting technique used to recycle cells.

4.1 Motivation and Introduction

One of our most significant modifications to the algorithm from [3] is the introduction of memory recycling. The most obvious motivation for this is reducing the algorithm's memory footprint. For the algorithm to be useful in applications such as FDR, it must be practical to execute it on graphs with at least billions of vertices. Without memory efficiency, the algorithm may exceed available disk space on most machines when processing very large graphs. Even with ample disk space, it is still desirable to reduce memory usage so that more data can fit in main memory. Reading/writing to disk is many magnitudes slower than accessing RAM or performing computations; therefore, if the algorithm does not fit into main memory, disk IO will become the primary bottleneck. Additionally, disk IO is a sequential operation on almost all systems, hence the algorithm may devolve into an effectively sequential program if it is forced to access the disk repeatedly.

The second motive for memory recycling is somewhat less obvious. On most systems, heap allocations/deallocations (i.e. objects created with `new` or `malloc`) are made thread-safe by locking the heap while servicing a thread's allocation or deallocation request. Thus, an algorithm that may appear

"lock-free" could actually be preforming a lot of locking on the system level and hence suffer from the performance penalties caused by locking. If all the threads are frequently allocating memory, performance could degenerate to that of a sequential algorithm since the locking may become a sequential bottleneck. This issue is recognized as a legitimate scalability obstacle as evidenced by the fact that top software firms such as *Google* and *Facebook* have implemented their own lock-free versions of `malloc` to use in their concurrent data structures. Reusing objects instead of allocating new memory on the heap reduces the frequency with which threads must allocate memory.

A third motive is that designing lock-free data structures that deallocate shared memory during run-time is very difficult in programming languages without garbage collection. It is only safe to delete a shared object if no other threads have references to it; but it is difficult to communicate this information without using locking. One might propose using reference counting, but then the question becomes "when is it safe to delete the memory holding the reference counter?" Standard implementations of smart-pointers do provide lock-free thread-safely as long as two threads do not attempt to access the same smart-pointer instance simultaneously, a situation that is difficult to avoid in many complex algorithms (i.e. in our implementation a search's `cellBlockedOn` field can be accessed by many threads simultaneously). One potential solution to the problem is simply freeing all memory at the end, but this creates a sequential bottleneck at the end of the algorithm and would results in holding unnecessarily large amounts of memory. Lock-free reference counting algorithms do exist, but so far, implementations are usually too complicated to be efficient.

It turns out recycling memory is easier than deleting it in a lock-free data structure as it avoids the problem of threads holding references to memory that no longer exists. Instead, objects simply maintain an `age` tag indicating how many times its been recycled. A thread verifies that a memory location still represents the same object by checking that the `age` counter is unchanged.

The above concerns are especially important because of the memory intensive nature of the algorithm. Just by default, the algorithm must keep the original read-only graph in memory for exploration. Next, each vertex is eventually assigned a cell object. Each cell object has `vertex`, `index`, `rank` fields which are at least four bytes and pointers to its `blockedList`, `neighbors` array and `status` field which are 8 bytes each. Thus, a cell object takes up at least 36 bytes, even more if the cell holds the `blockedList` and

`neighbors` list rather than pointers referencing the lists. On a graph with a billion vertices, our algorithm would use 35 Gigabytes just for cell objects. Thus, memory recycling can be very useful especially for large graphs.

Recycling search objects can be fruitful as well. Search objects must allocate memory for the control/tarjan stacks as well keep a few accounting variables. On relatively sparse graphs, the number of search and cell objects used may only differ by a constant factor.

The drawbacks of this approach include added complexity and requiring search and cell objects to hold some additional fields. Additionally, several commonly used data structures must now contain `cell-age` pairs so threads can verify the validity of the references which results in added overhead.

4.2 Implementation Details

The reuse of search and cell objects forms the crux of our memory recycling strategy. An object's `age` tag indicates how many times it has been recycled. Verifying if an object has been recycled consists of checking if the age has changed. The `age` field is atomic and accesses often require a memory barrier. A thread *reclaims* a search object by placing it on a thread local buffer of reclaimed searches. The same is done with cell objects. A thread needing to create a new cell or search will first check the appropriate buffer to see if there are any objects that can be recycled. Using thread local buffers allows us to eschew extra synchronization, but results in some wasted recycling opportunities if some threads collect or use significantly more objects than others. An alternative approach would have threads *steal* objects off other threads' buffers if theirs becomes empty; a common concurrent programming technique.

Let us address the reuse of searches first. Chapter 5 will prove that each search is executed by at most one thread at a time. Further, `execute()` and all the functions it calls except `suspend()` access only the search the thread is executing. If we ignore `suspend()` for a moment, we see that recycling a search after its `tarjanStack` becomes empty has no effect on other threads since they are executing a different search and the functions they call do not access the recycled search. A thread calling `suspend()` does access other searches during the cycle detection and resolution protocols, but Chapter 5 demonstrates that the protocols correctly guards against the possibility of recycled searches using the search's `age` field.

Note that reclaimed search objects need to have their `age` and `cellBlockedOn` fields set to the correct values (previous `age` + 1 and `null`) while they are on the buffer awaiting recycling so that a cycle detection protocol does not mistake it for a suspended search. There is no danger of searches participating in a cycle resolution protocol being recycled by another thread since they are suspended and hence incomplete.

Sometimes a cycle resolution will transfer all of the cells from one search to another. In that case, the empty search's memory must be reclaimed since it is now finished and will not be executed again. Reclaiming this memory during the cycle resolution is not dangerous since the search is suspended and hence no other thread will access its memory (except perhaps in another `suspend()` call, but that case is handled in Chapter 5, theorem 5.10). In summary, search objects are reclaimed when their stacks become empty either as a result of `execute()`'s depth-first search ending (line 50 of `execute()`) or as a result of a cycle resolution.

Before addressing cell recycling, we mention one other memory recycling technique. The `suspend()` function checks if the suspension of a search S on a cell of search T caused a cycle by constructing "suspended paths" (see beginning of 5.5 for details) which represent the candidate cycle. This protocol requires at least one dynamic array to store the paths. While the paths are usually small and can be destroyed after use, threads may invoke `suspend()` fairly often locking up the heap and defeating the payoff of having a lock-free suspension protocol. Memory recycling is easily implemented by having each thread reuse the same thread-local lists each time it calls `suspend()`. The lists are only used during `suspend()` and are not shared objects; therefore, no synchronization is required. A thread has at most one `suspend()` call on its stack, hence at most one call uses the list object at a time.

4.3 Partial Reference Counting and Cell Recycling

Cell objects are more difficult to deal with because, unlike with searches, there is not much guarantee as to when they will be accessed. For instance, if a complete cell becomes recycled during a call to `claim()`, the call could accidentally claim a recycled cell. The obvious solution, reference counting,

has significant problems. For instance, in a dense graph, a cell may be stored in many other cells' neighbors list. By the time the cell is removed from each list, the algorithm may be close to finished and hence most cells will never be reused. Additionally, cell objects are accessed fairly often in the algorithm, atomically tracking each reference may impose overhead. With these issues in mind, we instead employ *partial* reference counting. That is, only certain references increment/decrement the reference count and when the reference count reaches zero, no more valid references can be created.

In this section, we detail how cell objects are recycled using partial reference counting. The protocol is quite simple. When a new cell is created/reused, its reference count starts at `1`. Each time `execute()` explores a new `child` cell, the `child`'s reference count is incremented. The reference count is decremented when that iteration of `execute()`'s main while-loop ends. The thread that completes the cell will decrement the cell's reference count to cancel out the initial reference count of 1. When a cell's reference count reaches zero, no more countable references to the cell can be created and the thread last decrementing the reference count adds the cell object to its recycled cells buffer. This protocol has the useful property that a cell is recycled when it is complete and there are no calls to `execute()` exploring the cell. Observe that this property gives us a similar guarantee to the one we had with search objects.

Let us provide more detail. Recall the `Dictionary` object mapping vertices to their corresponding cell objects. To account for the reuse of cell objects, we modify the dictionary to map vertices to {`Cell*, Age`} tuples where `Cell` is the cell object representing the vertex when it was aged `age`. If a cell retrieved from the dictionary now has a different age than the one listed in the tuple, the cell representing the intended vertex can be assumed to have become complete and reused. We refer to these tuples as `WeakReferences` because their references to cell objects are not reference counted since they exist for a large portion of the algorithm's lifetime. The cell class's `neighbors` list now holds `WeakReferences` rather than just cell pointers; if the `WeakReference`'s `age` tag is not equal to the cell's age, the cell is assumed complete. The modified `initializeNeighbors()` function is presented below. The chief difference is that `WeakReferences` are now retrieved from the Dictionary and inserted into the cell's neighbors list. On line 15, the cell corresponding to the vertex is assumed complete if `neighbor.cell→age` $\neq$ `neighbor.age` as that indicates the cell was completed and recycled.

Code 4.1

```
1  struct WeakReference{
2    Cell* cell; int age;
3  }
4  void initializeNeighbors_MemRecycling(Cell* cell){
5    List vertices<Vertex> succs = graph.getNeighbors(cell);
6    for(Vertex succ: succs){
7      WeakReference neighbor = dictionary[succ];
8      //Has the cell become complete or been recycled
9      //(and hence became complete)?
10     if(!neighbor->isComplete() &&
11       neighbor.cell->age == neighbor.age)
12       cell->neighbors.add(neighbor);
13   }
14 }
```

Reference counting is implemented by the `Reference` class and by placing an atomic reference count field in the `Cell` class. The code for the `Reference` class is presented below. The classes are organized and implemented slightly differently in the actual implementation for efficiency reasons, but the synchronization techniques presented here are the same as in the implementation. Through operator overloading, the `Reference` object acts exactly the same as a normal pointer except that when it is destroyed (by going out of scope), it decrements the cell's reference counter.

Code 4.2

```
1  class Reference{
2  private:
3    Cell* cell;
4    Reference(Cell* _cell) {this->cell = _cell;}
5  public:
6    static Reference createReference(WeakReference weak){
7      Cell* _cell = weak.cell;
8      //If the cell's age is already higher than
9      //that stored in the
10     //WeakReference, the cell must have
11     //already become complete
12     If(_cell->age != weak.age) return EMPTY_REFERENCE;
13     int refCount; //Current number of references
14     do{
15       refCount = cell->referenceCount;
16
17         if(refCount == 0) return EMPTY_REFERENCE;
18     }while(!_cell->referenceCount.compare_and_exchange(
```

```
          refCount, refCount+1 ))
19      //The cell is already complete,
20      //so release the reference and return
21      //an EMPTY_REFERENCE
22      if(_cell->age != weak.age){
23        decrementRefCount(cell);
24        return EMPTY_REFERENCE;
25      }
26      return Reference(_cell);
27      //uses C++'s return value optimization
28    }
29    static void decrementRefCount(Cell* cell){
30      //Atomically decrement
31      int count = --(cell->referenceCount);
32      //Recycle if no more references exist
33      if(count == 0)
34        recycle(cell);
35    }
36  ~Reference(){
37    if(this != EMPTY_REFERENCE)
38          decrementRefCount(this->cell);
39    }
40    operator Cell*() -> = Same as pointer
41    operator Cell&*() = Same as pointer
42  }
```

`References` are created through the `createReference()` method (lines 6 to 32 of code 4.2 above), which converts the `WeakReference weak` into a `Reference` object. If `weak.cell`→`age` is not equal to `weak.age`, the cell `weak` refers to is already complete and hence line 12 returns an `EMPTY_REFERENCE` if `weak.cell`→`age` ≠ `weak.age`. The `EMPTY_REFERENCE` notifies the caller that the `WeakReference` refers to an already complete cell and hence it is not possible nor desirable to create a `Reference` to it. The `do-while` loop on lines 14 to 18 repeatedly attempts to increment the cell's reference count using the `compare_and_exchange()` on line 18. The `compare_and_exchange()` is atomic and therefore provides the necessary synchronization with other threads.

Any cell with `refCount` zero is awaiting recycling and any previous vertices it represented have been assigned to SCCs. Thus, `refCount` is verified to be non-zero before each `compare_and_exchange()` and an `EMPTY_REFERENCE` is returned in the event of a `refCount == 0` (line 24). This yields a useful invariant: if the cell's reference count is zero, it cannot be incremented

by `createReference()`. This guarantees that `References` to cells cannot be created while their memory is not in use. Therefore, it is safe to set `cell→refCount == 1` when assigning the cell for reuse since the `refCount` is zero while the cell awaits reuse.

A cell can become complete and be reused before a successful `compare_and_exchange()`. In that case, the cell no longer refers to the intended vertex and hence we undo the increment with a `decrementRefCount()` call and return an `EMPTY_REFERENCE` (lines 22 to 24). The `decrementRefCount()` function decrements the cell's reference count atomically and recycles the cell object if the reference count is zero after the decrement. It is easy to see from the code that `createReference()` increases the reference count of the cell and returns a non-empty reference or the call returns an empty reference and leaves the reference count unchanged.

Notice that a `Reference` object's destructor calls `decrementRefCount()` to decrement the cell's reference count when the Reference is destroyed. Thus each `createReference()` call incrementing the cell's reference count is eventually followed by a `decrementRefCount()`. Lastly, after completing a cell and unsuspending the searches blocked on it, the thread will call `decrementRefCount()` on the cell. This decrement accounts for the `refCount` initially being `1`. Thus, once the cell is complete and `unsuspend()` is called and all `References` to the cell are destroyed, the cell will be recycled. Further, note that `refCount` cannot reach zero more than once in a cell's lifetime since `createReference()` cannot increment a `refCount` of zero.

We modify `execute()` by replacing line 21 of code 3.2 with these lines to create the `Reference` and verify its validity.

```
Reference child = createReference(curr→getBestNeighbor());
 if(child == EMPTY_REFERENCE) continue;
```

The rest of `execute()` is unchanged since `Reference`'s * and → operators are overloaded to behave as a simple pointer's would. The `Reference` goes out of scope and is destroyed at the end of the iteration of `execute()`'s `while`-loop. The following observation follows from the discussion:

Observation 4.1. *Each cell is recycled when it has been completed and all References to the cell in* ***`execute()`*** *have gone out of scope. New* ***`References`*** *cannot be created until the cell is reused. The thread decrementing the reference counter last recycles the cell object.*

Notice that our implementation of the `Reference` class is lock-free. The

createReference() function contains a while-loop that repeats until the compare_and_exchange() succeeds. But each failure indicates that some other thread succeeded in incrementing or decrementing the refCount, hence a least one thread makes progress. Otherwise, there are no loops, GOTOs, or locks in the code so it is easy to see the implementation is lock-free.

Many pointers to cells are held throughout the algorithm yet only the reference to child in execute() is reference counted; we show that correct results are produced despite the limited reference counting. Let us start at the beginning of an execute() call. On line 19 of code 3.2, the cell curr is defined to be the controlStack.top and is used throughout the function. There is no point in reference counting curr because it is on the search's stack and hence it cannot be unexpectedly completed and recycled by another search.

A WeakReference to child is retrieved from getBestNeighbors() and converted into a Reference after line 21 of code 3.2. The execute() function only accesses cells child and curr, but initializeNeighbors(), suspend(), and buildSCC() may access other cells. The discussion above explains how InitializeNeighbors() is modified to be compatible with recycling. The cell child cannot be recycled during the entirely of the suspend() call since a Reference to it exists in execute(). suspend() may access other cells during the cycle detection/resolution protocols; Chapter 5 explains why the cycle detection protocol's accuracy is unaffected by cell recycling. The cycle resolution protocol is unaffected by cell recycling since any cell involved in a cycle cannot be complete. The buildSCC() function affects cells initially on the search's stack and hence cannot be arbitrarily recycled. After the cells are marked complete and the searches blocked on them are unsuspended, buildSCC() will decrement the cells' refCounts as they are now safe to recycle. Thus, our partial reference counting is sufficient to protect the correctness of the algorithm.

Chapter 5

Suspension Protocols

Our implementation and [3]'s temperately suspends searches that reach a cell on another search's stack. This chapter explains how searches are suspended and resumed as well as the techniques used to detect and resolve blocking cycles (deadlocks formed from searches suspended on one another). We use the same method of resolving cycles as [3], hence we present it first. The second section gives an overview of how [3] suspends searches and detects cycles. In next three sections, we develop our own wait-free protocols for suspending/unsuspending searches and detecting cycles. The final section brings the material in the chapter together to show that our suspension/unsuspension protocols are correct and wait-free.

5.1 Cycle Resolution

Upon reaching a cell already on another search's stack, searches will suspend until the cell becomes complete. Consider two searches S and T such that search S is suspended on cell C_t in T's stack and search T is suspended on a cell C_s on S's stack. Search S will not resume until C_t is assigned to a SCC, but for this to transpire, T must first be resumed. T must be resumed for C_t to become complete because no other search can add C_t to its stack once T has claimed it. Now T cannot resume until C_s is complete which cannot happen until S is resumed. Thus, without any cycle resolution protocol, S and T would remain suspended forever. This situation easily generalizes to the below observation.

Observation 5.1. *Blocking cycles will persist indefinitely unless they are resolved by a cycle resolution protocol.*

Our cycle resolution protocol closely resembles that of [3]; we illustrate it with an example. Suppose we are searching a graph G and searches S_1, S_2, and S_3 are among the searches exploring G. Figure 5.1 below (from [3]) illustrates this example. The situation is as follows. Search S_1 is at cell n_1 waiting for cell c_2 which belongs to S_2. S_1 is thus suspended on cell c_2 after following the edge from its own cell n_1 to c_2. Likewise, we have search S_2 at n_2, suspended on the cell C_3 which belongs to S_3. Finally, search S_3 completes the cycle by blocking on S_1's cell C_1 by following the edge from n_3 to c_1. Thus, we have a blocking cycle.

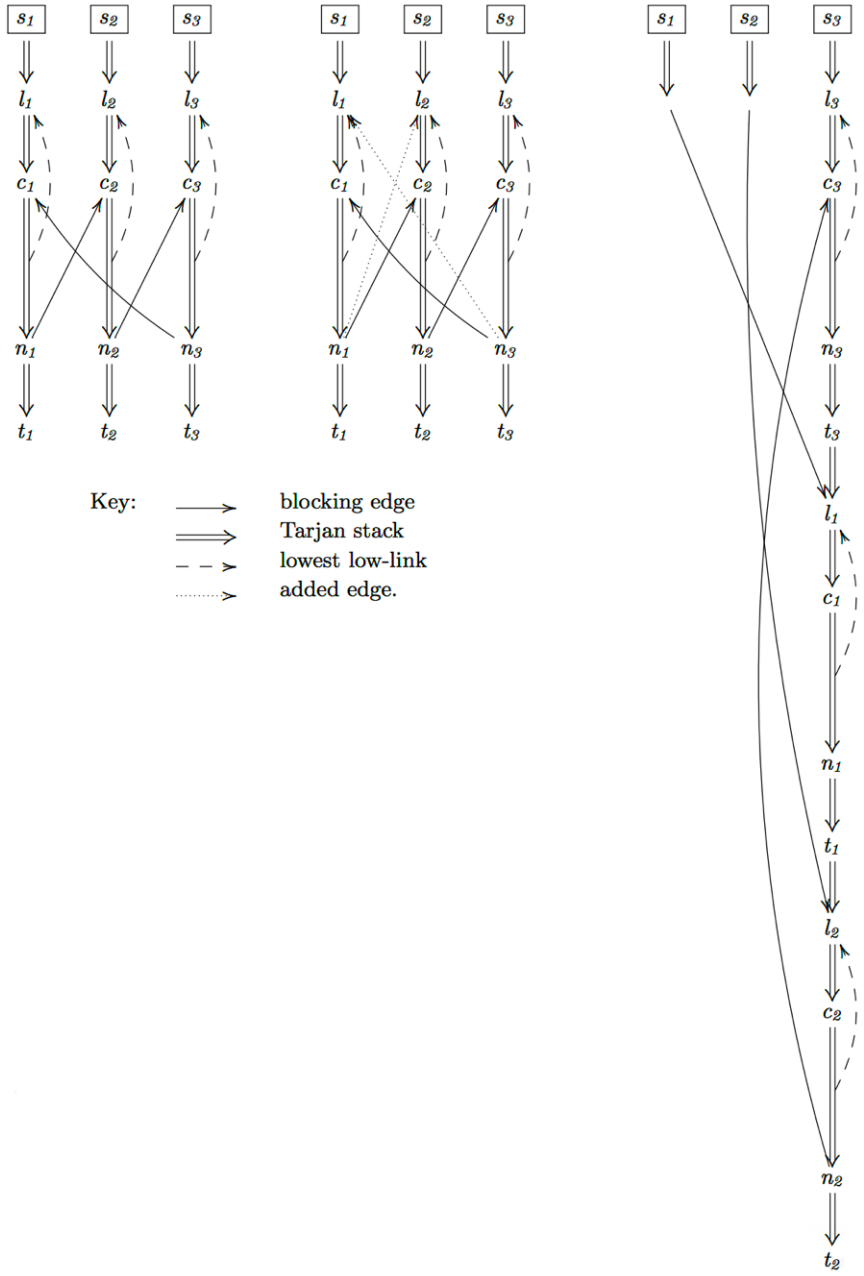
By theorem 2.2.(b), there must be a path from c_1 to n_1, from c_2 to n_2, and from c_3 to n_3. By our assumption, there is a path from n_1 to c_2, n_2 to c_3, and n_3 to c_1. Hence the cells from c_1 to n_1, from c_2 to n_2 and from c_3 to n_3 all must belong to the same strongly connected component which we will refer to as SET.

Note that n_1 must be the top of S_1's `controlStack` since S_1 followed an edge from n_1 just before suspending. The cell n_1 may not be the top of S_1's `tarjanStack`; however, as the search may have kept some cells on the `tarjanStack` when backtracking to n_1. By theorem 2.2.(a), these cells above n_1 in the `tarjanStack` must also be in SET. The analogous conclusions hold for S_2 and S_3. We denote the tops of the searches' `tarjanStack`s as t_1, t_2 and t_3 respectively.

By Theorem 2.1, if any cell between (inclusive) c_1 and t_1 has rank equal to the index of a cell L further down the `tarjanStack` than c_1, then all the cells above and including L are in SET. The newly identified members of SET, the cells (inclusive) between L and c_1, may have ranks lower equal to the indices of cells even outside this range, and so we repeat the process. We use the ranks of known members of SET in this way to isolate more members. Then we repeat the process recursively on the newly discovered members of SET. Let L_1 be the cell furthest from the top of the `tarjanStack` that can be uncovered by this recursive method. A rather simple traversal down the `tarjanStack` can identify L_1. We simply descend down the stack until a) we have found c_1 and b) the latest cell considered has an index at least as low as the lowest rank encountered so far. We define L_2 and L_3 similarly.

Suppose that we add edges from n_1 to L_2 and from n_3 to L_1 to the graph G. Clearly all the graph's SCCs remain the same since we only added edges between vertices in the same strongly connected component. Now imagine if the following had happened on the transformed graph. S_3 proceeds from cell L_3, reaching c_3, n_3 and eventually t_3 before backtracking back to n_3. Then,

Figure 5.1


s_1 s_2 s_3
l_1 l_2 l_3
c_1 c_2 c_3
n_1 n_2 n_3
t_1 t_2 t_3
Key:
blocking edge
Tarjan stack
lowest low-link
added edge.

it takes the edge from n_3 to L_1, explores all the way to c_1, n_1, and t_1 before backtracking back to n_1. Then, S_3 takes the edge from n_1 to L_2 and from there proceeds to c_2, n_2 and reaches t_2 and backtracks to n_2. Finally, S_3 takes the edge from n_2 to c_3. Meanwhile, searches S_1 and S_2 arrive at cells L_1 and L_2, respectively, and suspend as both are owned by search S_3. This execution is completely legal and hence the algorithm would produce correct SCC assignments on the altered graph had this execution occurred. As G has the identical SCCs to the altered graph, resuming the algorithm from this point would yield the correct SCC assignments. Hence, we can alter the stacks of the three searches to "pretend" this execution ordering occurred to resolve the cycle. More specifically, we transfer the cells L_1 to t_1 and L_2 to t_2 to the stack of the third search. The cycle is resolved because both S_1 and S_2 are now blocking on S_3 while S_3 can now be resumed since it now owns the cell it was blocking on. The transformation is shown in the rightmost part of figure 5.1.

To complete the transfer, we must transfer select cells from S_1 and S_2's `controlStack` to S_3's `controlStack` to emulate the situation described above. We transfer the cells from the top of S_1's `controlStack` up to and including L_1 to the top of S_3's `controlStack`. Note that L_1 must be on S_1's `controlStack`. To see this, note that L_1.`index` must equal L_1.`rank`, otherwise it could not possibly have index equal or less than the minimum rank of the cells transferred from S_1 and be the first cell on the `tarjanStack` to do so (how we defined L_1). A cell with rank equal to index cannot remain on the `tarjanStack` if it is taken off the `controlStack` because the algorithm would then assign it to a SCC. We similarly transfer the cells from S_2's `controlStack` to the new top of S_3's `controlStack`.

As a matter of accounting, we must update the index and rank variables of the transferred cells and increment the `cellCount` variable of the search receiving the cells. For instance, for search S_1, let $\delta = S_3$.`cellCount` $-$ L_1.`index`. We add δ to the index and rank variable of each cell transferred to S_3. We update S_3's `cellCount` to be larger than any of the new indexes. Finally, S_3 can be resumed and it starts with the edge from $n2$ to $c3$. Hence, we update the rank of $n2$ to be at least that of $C3$ to account for the fact that all the transferred cells are in the same SCC.

The intuition behind the cycle resolution protocol rests with a simple realization. Cycles are only formed when two or more searches explore the same strongly connected component. We have already proved this claim in the case of a cycle of length three, but the result is easily generalized. Our

example showed that at the very least, the cells the searches are blocked on, the cells the searches are paused on, and any cells on the `tarjanStack` above the tops of the `controlStacks` are all in the same SCC. In fact, if at any time two or more searches own cells from the SCC, a blocking cycle is inevitable. Each search will eventually have to block on one of the others, waiting for them to finish the SCC which can never be complete since no search owns all its cells. This result is summarized below.

Theorem 5.1. *(a). Let $S_1, S_2, \ldots, S_n$ be searches in a blocking cycle. There exists an SCC containing at least one cell from each of the searches' stacks. At a minimum, for each search, the SCC contains all the cells higher on the `tarjanStack` and including the conflict cell (c_i above) and the cell suspended on (n_i above). We refer to this SCC as the conflict set of a cycle.*

(b). Let G be a graph explored by a set of searches. If any set of searches simultaneously own cells in the same SCC, a blocking cycle containing at least those searches must eventually form.

Proof. The reasoning for both claims is provided above. □

Definition 5.1. *The **conflict set** of a blocking cycle is exactly the SCC described by Theorem 5.1.a. That is, the conflict set of a blocking cycle is the SCC containing all the cells the searches are blocked on.*

The above theorem demonstrates the fundamental connection between SCCs (specifically the conflict set) and blocking cycles. It also reveals the intuition behind the resolution protocol. A cycle occurs when more than one search explores the same scc, the conflict SCC. Recall that the cycle resolution protocol traverses the stacks of the searches to find Li, the lowest cell in the tarjanStack that we know is in the conflict set. In other words, we transfer not only the cells causing the conflict, but also any other cells we know are in that same SCC. The reason for this is made apparent by Theorem 5.1.(b). If two searches own cells in the same SCC, a cycle is destined to occur. Thus, if we do not transfer all the cells in the conflict set, another cycle will occur.

The next theorem is also significant and follows from the work done in the example of the protocol earlier.

Theorem 5.2. *All the cells transferred in the blocking cycle resolution protocol belong to the cycle's conflict set.*

The code for `transferCells` below generalizes the example to n searches. The correctness of the general protocol is argued for in [3] so it is omitted here.

We can choose any search to be the recipient search of the cell transfer by symmetry. That is, any search in a cycle resolution can play the role of S_3 in our example. In practice, we choose the receiving search be the one that detected the cycle first because it is slightly more convenient. The following observation summarizes this.

Observation 5.2. *Let $S = \{S_1, S_2, \ldots, S_n\}$ be searches in a cycle such that S_1 is blocked on a cell owned by S_2, S_2 is blocked on a cell owned by S_3, etc. Any of the searches could take the role of S_3 in the above example, i.e. any of them could be the recipient of the other search's cells in the conflict set as long as transfers are done in the correct order.*

Before finishing the section, we give another valuable result.

Theorem 5.3. *Suppose a search S received m cells in a transfer. If S is involved in another cycle and transfers any of the cells received in the first transfer, S will transfer all m of them. Additionally, the last cell S transfers in the second resolution will not be any of the m cells.*

Proof. For simplicity, consider the example with searches S_1, S_2 and S_3 above. The result is easily generalizable. Suppose S_3 is involved in another transfer where it is not the receiving search. The protocol will transfer all the cells from the `tarjanStack` top up to and including some cell L. Suppose, for contradiction, some, but not all, of the m cells from the original transfer will be transferred. Thus, L is one of the m cells. L must meet the condition that $L.\texttt{index} \leq min(\textit{\textbf{ranks}}\ \textit{seen so far})$. None of the transferred cells from t_2 to n_2 meet this condition because their own rank must be less than their index otherwise S_2 would not have kept them on its `tarjanStack` when backtracking to n_2. L also cannot be n_2 or any of the original m cells below n_2 in the `tarjanStack` because we must then have $L.\texttt{index} \leq n_2.\texttt{rank} = c_3.\texttt{index}$ which is lower than the index of any of the m cells. Thus, it is impossible to meet the condition $L.\texttt{index} \leq min(\textit{\textbf{ranks}}\ \textit{seen so far})$ if L is any of the m cells. Therefore, if any of the m cells is transferred, L will be a cell lower in the `tarjanStack` than any of the cells, proving both claims. □

Code 5.1

```
1  void transferCells(List<Search*> S, List<Cell*> C){
2    for(int i = 1; i < n; ++i){ //For searches S1 to Sn-1
3      Stack<Cell>  tarjanStack = Si->tarjanStack;
4      Stack<Cell>  controlStack = Si->controlStack;
5      Cell* next = tarjanStack.top();
6      /* transfer cells from src's tarjanStack,
7      starting from the top and proceeding until
8       a) The conflict cell is transferred to dest
9       b) The latest cell transferred has rank no greater
10      than the minimum rank of the cells transferred so far
11      */
12     int minrank = infinity; bool reachedCi = false; Cell next
         = null;
13     do{
14       next = tarjanStack.top();
15       //The top of the tarjanStack since previous cells are
        gone
16       minRank = min(minRank, next->rank);
17       if(next == Ci) reachedCi = true;
18       transfer(next, Si, Sn);
19     }while(!reachedCi || next->index > minRank);
20     Cell* last = next;
21     int delta = (dest->cellCount - last->index);
22     for(cell in All cells transferred){ //Do accounting
23         cell->index += delta; cell->rank += delta; cell.
        status = Sn
24     }
25     //Transfer cells from src control stack
26     //to destination control stack
27     //starting from the controlStack's top
28     //and proceeding to the last cell transferred
29     Cell* cell = null;
30     do{
31         cell = Si->controlStackTop();
32         controlStacktransfer(cell, Si, Sn);
33         //transfer cell from top of Si controlStack
34         //to top of Sn controlStack
35     } while(cell != last)
36     if(!Si.done()){
37         last->blockedList.push_back(Si);
38         Si->suspendOn(last);
39     }
40     dest.cellCount = dest->tarjanStack.top().index + 1;
41   }
```

```
42 }
```

5.2 Suspension and Cycle Detection: Lowe's Approach

With regards to suspending searches, we have only confronted one piece of the puzzle; resolving cycles. We presented it first because our solution was heavily based on [3] and hence we could just present the algorithm once. Our protocols for suspending and unsuspending searches and for detecting cycles are quite different from [3]. We will explain Lowe's approach and our own and provide reasoning for the changes.

Each cell has a list, `BlockedList`, that logs the searches that have suspended on it. When the cell is added to an SCC and marked complete, the searches blocked on it can be resumed. There are several potential race conditions that can arise with `BlockedList`. All potential race conditions are resolved by wrapping any of the cell's functions that read or modify its status or `BlockedList` with a lock belonging by the cell. Before suspending, a search obtains the lock on the conflict cell, makes sure it is not yet complete (hence it is still safe to add to `BlockedList`) and then adds itself to the `BlockedList` prior to releasing the lock. Meanwhile searches have a field `cellBlockedOn` which holds a reference to the cell the search is blocked on. This variable is used in the cell transfer protocol to identify the cells causing the conflict (the C_i variables we discussed in 5.1).

Lowe uses a map from search to search to store which searches are blocked on which. The map is encapsulated in a `SuspensionManager` object. A search S preparing to suspend on a cell belonging to another search S_1 will first examine whether suspending will result in a blocking cycle. The cycle detection protocol detects if S_1 is blocked on any search S_2. If S_1 is not suspended, there clearly is no cycle. Otherwise, the protocol probes the map to see if S_2 is blocked on any search. If not, there is no cycle. If yes, the process is repeated for S_2. If repeating this process eventually finds a search $Sn = S$, we have found a path from S_1 to S. Because S is about to suspend on S_1, we have a cycle. If there is a cycle, the protocol inputs the path $\{S_1, S_2, \ldots, S_n\}$ to the cycle resolution protocol to resolve the cycle. If a cycle is not detected, the mapping $S \rightarrow S_1$ is registered to the map to record the suspension.

The protocol is made thread-safe by having a thread hold `SuspensionManager`'s lock from the moment it begins probing the map to investigate whether a path from Sn to S exists until it finally adds the entry $S \rightarrow S_n$ to the map. All these operations must be done atomically to ensure the integrity of the path. While the lock must be held during the entirety of a typical suspension, [3] uses a clever trick that allows the lock to be released the majority of the cycle resolution protocol in the event of a cycle as long as the lock is reacquired before updating the suspended map with the new suspensions. This could meaningfully enhance performance by preventing the cycle resolution algorithm from slowing down threads that want to access the suspended map for typical suspensions and resumptions.

A detail must be added to the cycle resolution protocol for it to be comparable with this implementation. If a search S suspends on a cell of search T that is later transferred to another search R, we must replace the entry $S \rightarrow T$ in the map with $S \rightarrow R$ otherwise future cycle detection could give incorrect results. Thus, each time a cell is transferred, the search clears the cell's `blockList` and stores the former contents in some list L. Afterwards, the old entry in the map is exchanged with the correct one.

Searches on a cell's `BlockedList` must be resumed after the cell becomes complete. The `unsuspend()` function, protected by `SuspensionManager`'s lock, updates the suspension map accordingly and then adds the search to the `Pending` object for resumption.

5.3 A Lock-Free Suspension and Cycle Detection Protocol

The algorithm given in [3] proposes an efficient protocol for recording suspensions and detecting cycles. Still, we believe we have found areas to advance on. First, the algorithm uses three levels of locking:

1. Cell level locking: each time a cell is transferred, its lock must be acquired to identify the searches blocked on it so that the suspension map can be updated to reflect that the searches are now blocked on the cells' new owner.

2. Search level locking: blocking/unblocking a search, involving the search in cell transfer, and resuming the search acquire the search's lock to protect variables such as `cellBlockedOn` and the status of the search.

3. `SuspensionManager` level locking: As detailed above, suspending/unsuspending and cycle detection necessitate holding the `SuspensionManager`'s lock. Requiring a lock for these fairly common operations may result in a sequential bottleneck.

Second, [3] employs several data structures and variables; the suspension map, the `BlockedList`, the search status, and `cellBlockedOn` to achieve an arguably simple purpose: recording suspensions and detecting cycles. We contend that our algorithm improves upon these two areas, though at the cost of some added complexity.

Our aim in this chapter is to provide correct suspension, cycle detection and cycle resolution protocols. We have already explained how the latter is achieved. The two former goals are pursued in the `suspend()` and `unsuspendAll()` functions whose pseudocode is presented in the next section. The `suspend()` function, called in line 38 of code 3.2 in the `execute()` function, is invoked by a search that reaches a cell owned by another search and thus should suspend. The code for `suspend()` is presented in code 5.2 below.

The first two lines of the `suspend()` method officially suspend the search on the conflicting cell. Lines 7-12 of `suspend()` try to unsuspend the search if the conflicting cell became complete while the search was suspending. The rest of `suspend()` detects and resolves any cycles produced by the suspension. The `unsuspendAll()` method is invoked when a cell becomes complete to place all the searches blocked on it on the `Pending` object for resumption. In the previous section, cycles are identified using the `SuspendedOn` map to construct a path of searches suspended on one another. Paths that develop into a loop represent a cycle.

Note that if the `SuspendedOn` map is modified while a thread is building a path, earlier edges of the path may become invalid. For instance, we could falsely detect the path $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_1$ when in fact the search S_2 was no longer suspended by the time we discovered S_4 was suspended on S_1. Additionally, without synchronization, a cycle could go undetected if say two searches simultaneously construct paths before both recording the suspension in the map since neither detects the other's suspension in time. Thus, locking is used to prevent `SuspendedOn` from being modified while a thread constructs a "suspended on" path.

To avoid locking, we take an alternative approach. Threads discovering a cyclic path will traverse the path again to verify its integrity. The cycle

is confirmed if the searches have not resumed since the path was formed. Our second significant change is eliminating the suspension map and status field entirely. Having searches only record which cell they are suspended on actually makes things a lot easier. As mentioned earlier, we believe "squeezing" information into one variable facilitates the transition to lock-freedom. Further, if a search S is blocked on a cell that is subsequently transferred, we no longer have to update what search S is blocked on. This alone removes the need for cell level locking in transfers. Thus, suspension is now centered on a search's atomic `cellBlockedOn` field; a non-null value of the variable indicates that the search is suspended. When unsuspending a search, we usually use a `compare_and_exchange` to set `cellBlockedOn` back to null to ensure that more than one thread does not resume the search. The intimate relationship between the `cellBlockedOn` field and suspension is reflected in the definitions below.

Definition 5.2. *A search S becomes **suspended** on a cell c by setting its `cellBlockedOn` field to c. A search is **unsuspended** when its non-`null` `cellBlockedOn` field is set to another value.*

Providing formal definitions for suspension and unsuspension help us reason about the events. Further, defining the terms as atomic events makes linearizing the events with respect to one another straightforward. A search can be suspended on a cell in two scenarios: (a) line 8 of `suspend()` (b) if it is unsuspended from one cell and suspended on another during a cell transfer.

Every iteration of the main while loop in `execute(`S`)` will result in some progress on S; the exploration of an edge from `controlStack.top()`, the backtracking of the `controlStack`, or the search being suspended because progress is temporarily impossible. Progress is also made when S's `tarjanStack` becomes empty and the search's memory is reclaimed. We define a thread as executing a search when it is in a position to make progress on S.

Definition 5.3. *A thread with `execute(`S`)` is on its function call stack is said to **execute** the search S until it suspends the search. A `suspend()` call that returns `RESUME` allows the thread restart execution. A thread reclaiming S's memory is also said to be **executing** S.*

Our main goal for the remainder of this chapter is to construct an algorithm where the following rule holds:

Rule 5.1. *Every suspended search will eventually be resumed and executed by some thread. Every search is executed by at most one thread at a time.*

Before providing any detail, we must comment on the memory management techniques used by the algorithm. Once search and cell objects are no longer needed, their memories are recycled. Neither classes have full reference countering; therefore, in certain situations, it is possible that some of the objects will be recycled while another function has a pointer to their memory. During the `suspend()` call, we do not need to worry about the memory of the cell we are suspended on, `conflictCell`, being recycled during the call because there is a reference to the cell in the `execute()` call which evoked `suspend()`. The memory for the search `suspend()` is suspending cannot initially be recycled, but it can be recycled anytime after line 6 of suspend if the search is resumed and finished by another thread.

Code 5.2

```
1  Status suspend(Search* Sn, Cell* conflictCell){
2    const int Sn_age = Sn->age;
3    //Informs us if the search resumes
4    //and is recycled during the call
5    conflictCell->blockedList.push_back(Sn);
6    Sn->cellBlockedOn = conflictCell;
7    if(conflictCell.isComplete()){
8      if(Sn->cellBlockedOn.compareAndExchange
9          (conflictCell, null))
10       return RESUME;
11     else
12       return SUSPEND;
13   }
14   List<Search*> S; //The blocking path of searches
15   List<Cell*> C; //The cell the previous search in the path S
16                  //is blocked On
17   List<int> A; //Ai is the age of the cell object Ci in C
18   List<int> L; //Li is the age of the search object Si in S
19    //Find the search S0 that is directly blocking Sn
20   Search* S0 = conflictCell->getOwner();
21   /******* Check for cycle ********/
22   //Build Path: first pass
23   Search* Si = Sn; Cell<Vid>* Ci; int Li, Ai;
24   do{
25       Ci = Si->cellBlockedOn;
26       if(Ci == null)
27         return SUSPEND; //Not a cycle, so suspend regularly
```

```
28          Ai = Ci->age;
29          //Next confirm that Si is still suspended on Ci
30          //now that we recorded age
31          if(Si->cellBlockedOn != Ci)
32            return SUSPEND;
33          Si = Ci->getOwner();
34          if(Si == null)
35            return SUSPEND; //Not a cycle, so suspend regularly
36          //Odd numbers signal that the search
37          //is currently in a transfer, so we can safely
38        //suspend knowing any cycles are being resolved
39          if(Si->age is odd)
40            return SUSPEND;
41          C.append(Ci);
42          S.append(Si);
43          L.append(Si->age);
44          A.append(Ai);
45          while(Si != Sn); //Stop if we detect a potential cycle
46          //Second pass, verify if the path is actually a cycle
47          Si = Sn; int pathSize = S.size();
48          //Suspend normally if the path S or
49          //any of its associated vectors are
50          //different the second time around
51          for(int i = 0; i < pathSize; ++i){
52            Ci = Si->getBlockingCell();
53            //Make sure the search's age has not
54            //changed in between reads of cellBlockedOn
55            if(Si->age != L[(i-1 + pathSize) % pathSize])
56              return SUSPEND;
57            if(Ci != C[i])
58              return SUSPEND;
59            Si = Ci->getOwner(); //Returns null if Ci is complete
60            if(Si != S[i]) return SUSPEND;
61            if(Ci->age != A[i]) return SUSPEND;
62          }
63          //If Sn became suspended on a different cell, a later
64          //thread can handle any cycles it is involved in
65          //We also verify that Sn was not recyled
66          //since suspending
67          if(Sn->getBlockingCell() != conflictCell ||
68              Sn_Age != L[n])
69            return SUSPEND;
70          //Make the age of the search with lowest
71          //memory address odd
72          if(!S[minPtr]->age.compare_and_exchange
```

```
73              (L[minPtr], L[minPtr]+1))
74            return SUSPEND;
75        /* Sn can resume after cell transfer protocal transfers
76        * the conflicting cell to Sn
77        * Sn's cellBlockedOn field is made null before the
78        * transfer to prevent any false cycle detections
79        * during theintermediate steps */
80          Sn->removeCellBlockedOn();
81          transferCells(S, C); //Resolve the cycle
82          (S[lowest]->age)++;
83          //Make age of search even again to signal
84          //that the search is no longer in transfer
85          for(Search* Si: S)
86            if(Si->done())
87            //Reclaim memory from the finished search
88              thread.reclaim(Si);
89          return RESUME;
90      }
91  }
```

5.4 The Algorithm Part 1 (Unsuspend() and first half of Suspend())

A cell's `BlockedList` tracks the searches suspended on it so they can be unsuspended once the cell is complete. A search does not always stay suspended on a cell until the cell becomes complete; a cell transfer could susped the search on another cell. Thus, we sometimes need to undo additions to a cell's `BlockedList` so searches are not spuriously added to the `Pending` object. Partly because prohibiting deletions allows for a fast lock-free `BlockedList` implementation, we use another method to undo insertions into `BlockedList`. Namely, prior to adding a search to `Pending`, we use a `compare_and_exchange(search.cellBlockedOn, null)` on the search's `cellBlockedOn` field to ensure it is actually suspended on the newly complete cell. This eliminates the need for removing searches from a `BlockedList`.

The code for `unsuspendAll()` below offers a simplified version of the function called when a cell becomes complete.

Code 5.3

```
1  void unsuspendAll(BlockedList<Search*>* blockedList,
2      Cell<Vid>* const completeCell){
```

```
3     for(Search* search: blockedList)
4       if(search->cellBlockedOn.compareAndExchange
5            (completeCell, null))
6         pending.add(search);
7   }
```

Having `compare_and_exchange` set `cellBlockedOn` to `null` ensures that the search will only be placed on `Pending` once even if `blockedOn` contains more than one copy of the search.

The code for the `suspend()` function was displayed earlier in figure 5.2. The `suspend()` function begins by immediately adding the search to the conflicting cell's `BlockedList` and setting the search's `cellBlockedOn` field to the appropriate cell. We intentionally add a search to the `BlockedList` immediately prior to setting `cellBlockedOn` to maintain the invariant as closely as possible. For any search S and cell c, if S.`cellBlockedOn` $= c$, s is on c's `BlockedList`. The purpose of this strategy is to make `cellBlockedOn` synonymous with suspension (as implied by Definition 5.1) while relegating the role of `BlockedList` to simply an accounting variable. The invariant described above has a weakness; if a search S was added to the `BlockedList` while `unsuspendAll()` was iterating through it, the iterator may miss the entry. Lines 7-12 of `suspend()` address this concern. Let S and c be a search and cell, respectively, and suppose S added itself to c's `BlockedList` after `unsuspendAll()` was called. c's `BlockedList` iterator may have passed the location of S's entry before it was set. But `unsuspendAll()` is only called on a complete cell, hence c must be complete by the time line 7 evaluates c.`isComplete()`. Thus, the conditional will execute and S.`compare_and_exchange(`c`,null)` will be called. The operation will succeed and unsuspend and resume S if S is still suspended on c. Thus, S will be unsuspended even if it was added to `BlockedList` too late. The `compare_and_exchange` sets `cellBlockedOn` to `null`, removing the danger that the search will be unsuspended twice. If S was added to `BlockedList` prior to `unsuspendAll()` being called, then S is reachable (in c's `BlockedList` and the entry has not been passed by `unsuspend()`'s iterator yet) by the time S.`cellBlockedOn` is set to c since the latter is done later in `suspend()`. When a search is suspended on a cell during a transfer, it is also first placed on the `blockedList` before setting `CellBlockedOn` to the appropriate cell. In this case, we do not have to worry about the race condition described above as `unsuspendAll()` will never be called on a cell in the middle of a transfer since cells in cycles belonging to searches in cycles

cannot become complete. We have just proved the following theorem:

Theorem 5.4. *For any search S and cell c, if $S.$`cellBlockedOn` $= c$, then S is reachable in c's* `BlockedList` *or S was suspended by a* `suspend()` *call and will be unsuspended exactly once from c by the time the* `suspend()` *call exits.*

The following theorem follows from the same design considerations and is also useful.

Theorem 5.5. *Arbitrary search(es) can be added to any cell's* `BlockedList` *at any point of execution without changing the SCC output of the algorithm.*

Proof. The only time the contents of a `BlockedList` are used is when the cell becomes complete and calls `unsuspendAll()`. The function `unsuspendAll()` iterates through the searches in `BlockedList` and sets the searches' `cellBlockedOn` field to `null` if the search was blocked on that cell.

Suppose we arbitrarily add a search s to the `BlockedList` of some cell c. Let E be the event of the s.`cellBlockedOn.compare_and_exchange(`c`, null)` being evaluated (line 5 of unsuspendAll()). If s.`cellBlockedOn` is not c, then our adding c has no effect since the `compare_and_exchange` will return `false`.

To complete the proof, we need to show adding c had no effect even when the `compare_and_exchange` succeeds. Specifically, we show that if the `compare_and_exchange` had not occurred, an equivalent one would happen later.

Thus, let us assume E was a successful `compare_and_exchange` call. Since E was successful, `cellBlockedOn` must have been equal to c until event E. s.`cellBlockedOn` could have been set to c in exactly two situations. First, s.`cellBlockedOn` could have been set in a transfer. During a transfer, searches transferring cells to a destination search will set their `cellBlockedOn` field to the last cell they transferred from their `tarjanStack`, L and add themselves to L's `BlockedList`. Had we not artificially added s, a later `compare_and_exchange` would succeed and s would be resumed anyway since the transfer mechanism added another copy of c to `BlockedList`. Further, once the `compare_and_exchange` succeeds, `cellBlockedOn` becomes `null`, so `compare_and_exchange` fails on the entry of s added by the transfer and hence s is, as intended, added to `Pending` only once.

s.`cellBlockedOn` could have also been set to c by line 6 of a call to `suspend()`. Because s was added to c's `BlockedList` on line 5, a copy

of s would have been in `BlockedList` even if it had not added artificially. Thus, in most cases, adding s is irrelevant since there is another copy of s in the `BlockedList`. The search s is only resumed once since the succeeding `compare_and_exchange` sets `cellBlockedOn` to `null`.

There is one interesting case requiring special attention. Suppose c became complete and `unsuspend()` was called at the same time as a `suspend()` call by another thread added s to c's `BlockedList`. It is possible that the thread calling `suspend()` will not add s to the `BlockedList` quick enough for the `BlockedList`'s iterator in `unsuspendAll()` to process that entry (this is a consequence of not using any locking).We must show that the outcome is equivalent to the one with event E. Suppose `suspend()` did not add s to the `BlockedList` in time for the iterator to consider it. This implies that c must have been complete after line 5 in `suspend()` otherwise `unsuspendAll()` would not have been called yet. Therefore, c.`complete()` on line 7 of `suspend()` must evaluate to `true`. The `compare_and_exchange` on line 8 of `suspend()` fails because E sets `cellBlockedOn` to `null`. In this scenario, the thread calling `suspend()` does nothing else to s since `suspend()` returns `SUSPEND` while the thread discovering calling `unsuspend()` adds s to `Pending`. Thus, s is resumed exactly once, as intended. Now suppose we never added s and event E never happened. In this case, the `compare_and_exchange` on line 8 of `suspend()` would succeed unless another `compare_and_exchange` in `unsuspendAll()` or in a `suspend()` call successfully set s.`cellBlockedOn` to *null* before that (events linearized at the `compare_and_exchange` call). If successful, the `compare_and_exchange` on `suspend()` line 8 would have the thread calling `suspend()` continue the search s. If a `compare_and_exchange` in `unsuspendAll()` succeeded on s.`cellBlockedOn` instead, s would be placed on `Pending` and resumed by another thread. Either way, because the `compare_and_exchange` sets s.`cellBlockedOn` to `null`, the search is resumed exactly once. □

The above theorems demonstrate that suspension is aptly defined around the `cellBlockedOn` field and the `BlockedList` is simply a background accounting variable. First, we see that the `BlockedList` always contains the necessary entries; whenever S.`cellBlockedOn` $= c$, the search S is effectively in c's `BlockedList`. Second, any extraneous entries in the `blockedList` have no effects. The second fact indicates that from a `BlockedList` is unnecessary when unsuspending a search.

5.5 The Algorithm Part 2 (Cycle Detection)

This section details our cycle detection algorithm and argues its correctness. The second part of the `suspend()` method detects and resolves cycles. Lines 18 - 43 of figure 5.2 above identify a candidate cycle by constructing a blocking path to the suspending search. Afterward, lines 44-66 verify the integrity of the path. Specially, let S_n, C_0 and S_0 be the suspending search, the conflicting cell, and the search owning the conflicting cell, respectively. If S_0 is not suspended, the function returns as there is no cycle. Otherwise, let C_1 be the cell S_0 is blocked on and S_1 be the search owning C_1. Inductively, if S_i is not suspended, we return for there is no cycle. Otherwise, we define C_{i+1} as the cell blocking S_i and S_{i+1} as the search owning C_{i+1}. Should we eventually reach $S_i = S_n$, we have our candidate path $\{S_0, S_1, \ldots, S_n\}$. To later verify the cycle's legitimacy, we construct the following lists on the first pass:

1. $S = \{S_0, S_1, \ldots, S_n\}$ where the S_i is defined as above
2. $C = \{C_0, C_1, \ldots, C_n\}$ where the C_i is defined as above
3. $A = \{A_0, A_1, \ldots, A_n\}$ where the A_i is the age of C_i
4. $L = \{L_0, L_1, \ldots, L_n\}$ where the L_i is the age of S_i

To verify the integrity of the cycle, lines 44 - 66 again start with S_0 and attempt to recreate the path. Once a cycle is created, none of the involved searches or cells can progress. Hence, if anything is different on the second pass, even an age variable, `suspend()` can safely return. Additionally, if any of the C_is are complete or any of the search's ages are odd (odd ages signify the search is in a transfer), `suspend()` will return. Conversely, if the paths are identical, the cycle is confirmed and `suspend()` runs the cell transfer protocol to resolve it. Afterwards, S_n, now holding all the transferred cells, is resumed.

There are two small details to note. A "consensus" mechanism is provided so searches suspending at approximately the same time can decide which will resolve the cycle. Namely, the thread successfully using a `compare_and_exchange()` to increment the age of the search with the smallest memory address goes on to resolve the cycle (line 67). The odd number marks the cycle as being resolved. After the transfer, the search's age must

be made even again by incrementing it rather than by returning it to its previous value. This is to prevent a stalled thread from erroneously executing a successful `compare_and_exchange()` call on the search's original age after the cycle is resolved.

Second, the receiving search in the transfer, S_n, must be unsuspended before the cell transfer. Otherwise, if S_0 is still in its `suspend()` call during the transfer, S_0 may falsely detect the cycle $\{S_0, S_n\}$ since both searches will technically be suspended on one another.

Formally defining cycles will provide clarity as the section proceeds. We define cycles in terms of the algorithm's sequentially consistent variables; the searches' `cellBlockedOn` field and the cells' status field.

Definition 5.4. *Let $S_0, S_1, \ldots, S_n$ be searches and $C_0, C_1, \ldots, C_n$ be cells. The* ***cycle*** $\Omega = \{\{S_0, S_1, \ldots, S_n\}, \{C_0, C_1, \ldots, C_n\}\}$ *is said to exist while $S_i.\texttt{cellBlockedOn} = C_{i+1 \pmod n}$ and $C_{i+1 \pmod n}.status = S_{i+1 \pmod n} \forall i \in [0, n]$. In other words, the cycle persists while each search S_i is suspended on the cell $C_{i+1 \pmod n}$ and $C_{i+1 \pmod n}$ is on $S_{i+1 \pmod n}$'s stack.*

Theorem 5.6. *None of the searches or cells involved in a cycle Ω make any progress while the cycle persists. In other words, the fields of the involved searches and cells remain the same while the cycle exists.*

Proof. This statement follows from observation 5.1. A search cannot continue while it is suspended so its fields will remain unchanged. Similarly, the fields of the cells owned by the suspended searches cannot change. □

In our implementation, we mark all the cells in an SCC as complete before unsuspending the searches blocked on them to prevent them for resuming and again blocking on the same cell. With this implementation detail, we will show that searches never suspend on the same cell twice. This fact underpins our cycle detection protocol.

Lemma 5.1 (Uniqueness of Suspension Lemma)**.** *A search cannot suspend on a cell more than once.*

Proof. Let S be a search suspended on a cell c ($S.\texttt{cellBlockedOn} = c$). We must show that S will never suspend on c again. Let SET be the SCC containing c. Consider the four situations below:

(a) S is suspended on a cell x such that c and x are in the same SCC. Both cells are owned by the some search R. x may equal c. Search S has been suspended on c once.

(b) S is not suspended, has only blocked on c once, and holds c in its `tarjanStack`.

(c) The situation was identical to that of (a) when the SCC holding c and x is finished.

(d) S, having only been suspended on c once, assigns the cell to an SCC after having the cell on its stack.

We will structure the proof as follows. First, we show that our initial assumption leads to one of (a), (b), (c) or (d). Next, we show that (c) and (d) leads to the desired result. To complete the proof, we show that situations (a) and (b) must lead to (a), (b), (c) or (d) and hence we can recursively follow the transitions until we reach (c) or (d).

Let S, c and x be as in (a). When the SCC containing c and x is finished, all of its cells are marked complete before any searches blocked on the SCC's cells are unsuspended. Thus, S cannot suspend on c once it is resumed because the `execute()` function checks if a cell is complete before suspending on it. Therefore, (c) leads to the desired result. The analysis for (d) is similar.

Next, let us show that our initial assumption yields one of the four scenarios. If S remains suspended on c until the latter becomes complete, we trivially have (a) followed by (c).

Thus, let us study the more interesting scenario where S is involved in a transfer before c becomes complete. First, suppose S is not the receiving search in the transfer. S will suspend on the last cell it transfers, x, the one that was furthest from the top of its `tarjanStack`. Even if x is not c, the cells must be in the same SCC by Theorem 5.2. Since S has only been suspended on c once and the receiving search holds both c and x, we have (a). If, on the other hand, S is the receiving search of the transfer we have situation (b). Now we show that (a) and (b) always yield one of the four scenarios.

Let cells c and x and searches S and R be as in (a). If R simply goes on to complete the SCC, we trivially have (c). First, suppose R transfers some cells to a search R' to resolve a cycle not involving S. A transfer must have originally caused S to suspend on L before c became complete; thus Theorem 5.3 demonstrates that either both or neither of c and L are transferred to R'. Either way, we still have (a).

Next, suppose that R is involved in a cycle that also includes S. Let SET' be the conflict set of that transfer and d be the conflict cell that caused the previous search in the cycle to suspend on S. We must show $SET = SET'$. Clearly, $d \in SET'$. Further, d must have been in the `tarjanStack` during the first transfer since S has been suspended since. The cell d must also have been lower in the `tarjanStack` than c at the time of the first transfer otherwise it would have been transferred to R (see mechanics of transfers earlier). Thus, since c was closer to the stack's top than d, Theorem 2.2.(b) asserts there must be a path from d to c and hence from SET' to SET. Similarly, since SET has not yet been completed by the time R is exploring cells from SET', there must be a path from be a path from SET to SET'. Thus $SET == SET'$ as both SCCs have paths into each other.

Suppose S is the transfer's receiving search. Then, since $SET' == SET$, theorem 5.3 states that S receives c as part of the transfer before resuming. This leads to situation (b). Now suppose some other search R' is the transfer's receiving search. Because $SET == SET'$, Theorem 5.3 states that c must be transferred to R' while S now blocks on the most recent cell it transfers to R', L'. Thus, we have situation (a). Note that since S is suspended on a cell owned by R, any cycle containing S must also contain R. Thus, we do not cover the case where S is in a cycle without R.

Now let S and c be as in (b). If S goes on to complete the SCC containing c, we have (d). Consider the case where S ends up suspending on a cell y owned by some search T instead. It is possible that T will eventually be in a cycle not involving S and transfer y to another search T', but this does not change the analysis. If T eventually resumes on its own, we have situation (b) again.

Finally, we must consider the case where S and T are involved in a cycle. Let R' be the receiving search of the transfer. If $S == R'$, we again have (b). Let us study the more interesting case where $S \neq R'$. First, consider the subcase where the c is not involved in the transfer. Then, the situation effectively remains the same; S will be blocked on some other search T' while still owning c. The second subcase is where c is transferred to the search R' and S blocks on the last cell it transfers, L. Cell $L \neq c$ by Theorem 5.3; therefore, S has still blocked on c once. Since both L and c were transferred in this resolution, they are in the same SCC and hence we have (a). □

We make some useful observations about cycles. We will use these facts without reference as they are self-evident.

Observation 5.3.

(a) For any cycle $\Omega_n = \{\{S_0, S_1, \ldots, S_n\}, \{C_0, C_1, \ldots, C_n\}\}$, $S_i == S_j$ *implies* $i == j$.

(b) A search S_i *cannot be part of more than one cycle.*

Proof. S_i.`cellBlockedOn` can only equal one search at a time, hence a search can only be blocked on one cell at a time. Additionally, each cell is owned by at most one search (given by the status field). Both observations follow from these facts. □

To demonstrate the correctness of our cycle detection protocol, it is sufficient to show that (a) Any detected cycle is a legitimate cycle (b) each cycle is identified and resolved by exactly one search. We will begin with (a).

Theorem 5.7. *Every cycle identified by the cycle detection protocol is a legitimate cycle that persists until it is resolved.*

Proof. Consider a `suspend(`S_n, C_0`)` call that has detected a cycle (i.e. reached line 69 without returning). By observation 5.1, it suffices to demonstrate that the cycle existed at some point during the `suspend()` call. Let event E be the moment between the first and second pass of the cycle detection protocol. We will show that the cycle existed during E. On the first pass, the cycle detection protocol will have constructed the lists $S = \{S_0, S_1, \ldots, S_n\}$, $C = \{C_0, C_1, \ldots, C_n\}$, $L = \{L_0, L_1, \ldots, L_n\}$, and $A = \{A_0, A_1, \ldots, A_n\}$ representing the suspended path of searches, the conflicting cells on the path, the ages of the searches and the ages of the cells respectively. The lists are defined more precisely above.

We study event E. Observe that we read S_i.`cellBlockedOn` $== C_{i+1 \pmod n}$ on the $(i+1)$th iteration of the first and second pass loops (lines 23 and 50 of `suspend()`). We know the S_i pointer represents the same search as S_i.`age` $== A_i$ before the first read and after the second read of S_i.`cellBlockedOn`. We can give the same guarantee with search S_n as line 64 verifies that S_n's age has held constant since it was suspended. Lines 26-29 capture the cell's age when the search first suspended on it, while line 59's read of C_i's age occurs after the read of S_i.`cellBlockedOn`, ensuring that during both reads the search is suspended on the same cell. Therefore, the Uniqueness of Suspension Lemma indicates that S_i must have been suspended on $C_{i+1(mod n)}$ between the reads of S_i.`cellBlockedOn` since

a search cannot suspend on the same cell twice. Therefore, we have that S_i.`cellBlockedOn` == $C_{i+1 \pmod n}$ for all i during E.

Next, we want to show C_i.`getOwner()` == S_i for all i during event E. Let $i \in [0, n]$ be arbitrary. We read C_i.`getOwner()` == S_i during the $(i+1)$th iteration of the second pass while loop (line 57). On the next iteration of the while loop, we read S_i.`cellBlockedOn` == C_{i+1} (for $i == n$ we later read S_n.`cellBlockedOn` == C_0 on line 64.) Therefore, the read of C_i.`owner` occurs between the first and second pass reads of S_i.`cellBlockedOn`. Since S_i was suspended the entire time between the two reads of S_i.`cellBlockedOn`, S_i could not have gained cells between event E and the later read of C_i.`getOwner()` == S_i. Therefore, C_i.`getOwner()` == S_i for all i during event E. We have established that the cycle $\Omega = \{S, C\}$ existed during event E which is sufficient to prove our claim.

Finally, we describe how other false detections are avoided. The cell S_0.`cellBlockedOn` will be on S_n's stack at some point during the cell transfer. Since Sn is suspended on a cell on S_0's stack, we set S_n.`cellBlockedOn` = *null* before initiating the cell transfer otherwise a false cycle between S_0 and S_n could be detected. Line 64 uses the Uniqueness of Suspension Lemma to ensure that S_n has not been unsuspended by another thread since the call to`suspend()`, protecting against erroneous cycle detections that could result from S_n resuming in the middle of `suspend()`. □

Theorem 5.8. *For every cycle* $\Omega = \{\{S_0, S_1, \ldots, S_n\}, \{C_0, C_1, \ldots, C_n\}\}$, *exactly one search* S_i *will detect and resolve the cycle.*

Proof. Let $\Omega_n = \{\{S_0, S_1, \ldots, S_n\}, \{C_0, C_1, \ldots, C_n\}\}$ be a cycle. Assume, for contradiction, that the cycle went undetected. It follows that the involved searches will not make any more progress as interfering false cycle detections are impossible by theorem 5.8.

Each Si was suspended by a `suspend()` call or by a cell transfer. All of the `suspend()` calls reach the cycle detection protocol since line 7's conditional fails because the searches suspend on incomplete cells. Therefore, each `suspend (Search,Cell)` call first sets `Search.cellBlockedOn = Cell` building up the lists S, C, A, L, defined above,to represent the suspended path. We first show that at least one search S_i will correctly read S_j.`cellBlockedOn` = $C_{j+1 \pmod n}$ and $C_{j+1 \pmod n}$.`status` = $S_{j+1 \pmod n}$ for all j during the first pass.

We can use the line S_i.`cellBlockedOn` = C_{i+1} in `suspend()` to generate a sequentially consistent ordering of suspensions since `cellBlockedOn`

is an atomic variable. Since the `cellBlockedOn` field is set before cycle detection, any search S_j setting its `cellBlockedOn` field to the appropriate cell after some search S_i does the same will be able to read S_i.`cellBlockedOn` $== C_{i+1 \pmod n}$ while building the suspended path. If S_i and S_j set `cellBlockedOn` simultaneously, they are both able to read the appropriate value for each other's `cellBlockedOn` field for the same reason. Let S' be the search that sets its `cellBlockedOn` field after all the others (if there is a tie, choose any). We must ensure S' was suspended by a `suspend()` call as only those searches run cycle detection. Suppose, for contradiction, that S' was suspended on the last cell it transferred to a search R as part of a cycle resolution. But R is unsuspended after the cycle resolution. Thus, R would have to subsequently suspend and join the cycle or R would have to transfer the cell S is blocked on to some search that would subsequently suspends or repeats the process. This contradicts our assumption that S' suspended last; therefore, we can confirm S' was suspended by a `suspend()` call. We have shown S' will run the cycle detection protocol after every search has set their `setBlockedOn` field to the correct value.

Let us study the `suspend()` call of S'. Without loss of generality, suppose $S' = S_n$. After setting `cellBlockedOn` appropriately, `suspend()` starts the cycle detection by identifying the search owning S_n.`cellBlockedOn` $== C_0$. C_0.`status` $== S_0$ is read on line 31 since the search S_0 cannot receive C_0 after S_n suspends without resuming and re-suspending and breaking the assumption that S_n suspended last.

The age of search S_0 is even as S_0 is not involved in a cycle resolution by our initial assumption. Now, let's examine the next iteration of the path building. S_0.`cellBlockedOn` must be C_1 as S_n was the last search to block on the appropriate cell. Line 57 must read C_1.`getOwner()` $== S_1$ as S_1 must already be suspended and a suspended search cannot receive new cells without first unsuspending. S_1 even for the same reasons as S_0. Each subsequent iteration similar. The `suspend()` call correctly reads S_i.`cellBlockedOn` $== C_{i+1 \pmod n}$ because each search suspended before we started the cycle detection and each C_i must already be on S_i's stack since the search would have to resume to obtain C_i. This is repeated until we reach S_n.

Afterwards, the `suspend()` call again begins from S_n and repeats the same mechanism to confirm that the path is identical. Recall that all of the S_i's must have been suspended since this cycle detection mechanism started. Further, suspended searches cannot make any progress and outside inter-

ference by false cycle detections is impossible by Theorem 5.7. Thus, the second pass encounters the same searches and cells in the same order as the first pass. None of the C_i's could have become complete since they are owned by suspended searches. Cells are only recycled when they become complete, thus the ages of the cells are identical on both passes. Similarly, a searches' age field is only modified in two scenarios. First, the age field is incremented by two when the search is finished and recycled. Second, line 67 of `suspend()` increments the age field of the search with the lowest pointer value by one to indicate to other threads that the cycle is being taken care of. After the cycle is resolved, the age field is incremented again to announce that the cycle has been resolved. The searches' age fields must stay constant during both passes since neither scenario is possible. We have shown that both passes will encounter the same cells and searches of the same age. Further, the `compare_and_exchange()` on line 67 succeeds because none of the S_i.`age` variables could have changed. Therefore, `suspend()` executes the cycle detection protocol; contradicting our original assumption. We can conclude that at least one search will execute the cycle resolution protocol. The `compare_and_exchange()` on line 67 acts as a flag ensuring that if multiple searches detect the cycle, only one will run the resolution protocol. Recall that the path building portion of `suspend()` does not allow searches with odd ages, thus no thread will reach line 67 with the new incremented age variable. Therefore, the `compare_and_exchange()` will succeed for exactly one thread. □

We have demonstrated the correctness of our cycle detection and resolution protocol. Each cycle will be identified and resolved by exactly one thread and only legitimate cycles will be detected.

5.6 Correctness of Suspension/Unsuspension

We put together the material in the previous sections to demonstrate the overall correctness of our system of suspending/ unsuspending threads. We start with the observation that if a cycle is not detected, any code after line 12 on `suspend()` has no effect.

Observation 5.4. *Unless* ***`suspend()`*** *resolves a cycle, the function makes no modifications to* S_n*,* ***`conflictCell`*** *or any other non-local variables after*

line 12 below. In fact, a successful `compare_and_exchange` *on line 8 is the only possible write to a non-local variable after line 6 if no cycle is resolved.*

Proof. A successful `compare_and_exchange` on line 67 makes a modification to S_n's age, but also results in the `runCellTransfer()` function being executed on line 75. Thus, by the correctness of the cycle detection protocal proven in the last section, a cycle must have occurred. Similarly, while the code on lines 76-81 has the potential to modify a search, they are executed if and only if `runCellTransfer()` is called because there are no jump/break/return statements between lines 74 and 82. □

There is one more issue to address about `suspend()`. Notice that S_n could be resumed by another thread at any point after line 6. For instance, `conflictCell` could become complete, find S_n in its `BlockedList`, and add it to `Pending` after a successful `compare_and_exchange` in `unsuspendAll()`. Thus, `suspend()` must account for the possibility that the search S_n has been resumed at any point after line 6. In fact, the resumed search could be finished and have its memory reused to execute a new search S'_n. Thus, we must account for the fact that S_n might even represent a different search after line 6.

The following theorem allays our fears by stating that if S_n resumes between line 6 and `suspend()`'s return, the outcome would be no different than if that call to `suspend()` ended early right after line 6 and returned `SUSPEND`. For the theorem to be useful, we must explain why it is desirable for `suspend()` to act as in the second outcome if the search resumes early.

In short, everything after line 6 is meant to handle special cases where the search would not resume on its own (cycles or late additions to `BlockedList`). Thus, if S_n resumes in the middle of `suspend()`, the outcome should be the same as if `suspend()` ended at line 6. In more detail, S_n resuming during the `suspend()` call indicates that the `suspend()` function did not need to resolve a cycle. Since there was no cycle, Observation 5.4 above states that only a successful compare_and_exchange on line 8 could have had a meaningful effect on the algorithm. But the `compare_and_exchange` is unnecessary as S_n resumed on its own. Thus, the below theorem states that we can pretend that a search resuming in the middle of a `suspend()` call was resumed after the call completed.

Theorem 5.9. *Suppose a thread calls suspend* (S, c) *on a cell* c *and some search* S *which is resumed by another thread between line 6 and the end of*

`suspend()`. The output of the algorithm would be unchanged if `suspend()` had returned `SUSPEND` immediately after line 6, skipping the rest of the function.

Proof. Let S and c be as above and suppose S resumed after line 6 of a `suspend(`S`,`c`)` call executed by a thread T. In fact, S may have resumed, suspended and then resumed again any number of times before line 6, the analysis does not change. We need to consider the reads and writes to S by the `suspend()` function and how the outcome is influenced if any of them are done on a resumed S.

The first interaction with S after line 6 happens during the `compare_and_exchange` on S.`cellBlockedOn` in line 8. If c.`complete()` evaluates to `false` on line 7, line 8 is ignored and thus we can proceed. Suppose, for the time being, the c.`complete()` evaluates to `true` and hence we reach line 8. If S resumes after the `compare_and_exchange`, it must be that the `compare_and_exchange` was unsuccessful otherwise `cellBlockedOn` would have been set to `null` making resumption impossible. In that case, `SUSPEND` is returned and our claim holds. Now we study the scenario where S resumes sometime before the `compare_and_exchange` on line 8 (it does not matter exactly when, since only reads/writes of S matter). If the `compare_and_exchange` fails, then the function returns `SUSPEND` without altering non-local variables after line 6. Therefore, in this case, our claim holds.

If the `compare_and_exchange` succeeds, `suspend()` returns `RESUME` which results in the current thread executing S', the search located in the memory location where S was stored. S' may be S or another search reusing S's memory address after S was completed and had its memory reused. If `suspend()` returned after line 6 and S' wasn't resumed, another `suspend()` call or `unsuspendAll()` call on c would have resumed the search. Upon returning from `suspend()`, T can safely keep running `execute(search)` as if search was still S. This design is a bit peculiar, but works because the entire function is enclosed in a while loop and has no local variables aside from S. The successful `compare_and_exchange` on line 8 sets S'.`cellBlockedOn` to `null`, therefore, the `compare_and_exchange` in `unsuspendAll()` will fail and S' will not be added to `Pending`. Therefore, S' is correctly resumed exactly once. If, instead, `suspend()` had simply returned `SUSPEND` after line 6, `cellBlockedOn` would have never been set to `null`. Thus, either `unsuspendAll()` would have placed the search on `Pending` or another

"rogue" `suspend()` call would have taken over the search. Either way, only one thread would resume S'. Thus, we have shown that if S resumes after line 6 and the `compare_and_exchange()` succeeds on line 8, the output would be as if `suspend()` just returned at line 6; the only difference is which thread ends up working on S'.

Next, we assume that c.`complete()` evaluated to `false` on line 7 and hence the first `compare_and_exchange` is skipped. We will show that the call to `suspend()` does not detect a cycle, resulting in an equivalent execution to `suspend()` exiting after line 6 by Observation 5.4. Suppose, for contradiction, that `suspend()` detected and resolved a cycle. By earlier assumption, S resumes sometime in the `suspend()` call as the search S'. S' could either be the resumed S, or S could have been finished and S' eventually represents the search occupying the recycled memory where S resided. The detected cycle must be legitimate by Theorem 5.4, hence S could not have resumed between the detection and resolution of the cycle since no progress is possible while the cycle exists. S could not resume after the cycle is resolved because S' is either the receiving search (resumed after `suspend()` returns) or is blocked on the receiving search at least until the receiving search resumes (and hence after `suspend()` returns). Therefore, S' must have resumed before the cycle detection finished.

Since the detected cycle is legitimate, S' must have suspended again after resuming. If the resumed S' represents the original search, it cannot be suspended on c by the Uniqueness of Suspension Theorem. If S had its memory recycled, S' now has a different age. Either way, the conditional on line 64 ($S_n \rightarrow$ `getBlockingCell` $\neq$ `conflictCell` $\parallel$ Sn_`age` $\neq L[n]$) fails. This leads to `suspend()` returning without detecting the cycle, yielding a contradiction. Therefore, if S resumes any time during the `suspend()` call, the call will not detect a cycle. Should the resumed S' enter a cycle, theorem 5.9 guarantees another thread will later resolve the cycle. □

Lemma 5.2. *Assuming the algorithm runs on a finite graph G, the total number of suspensions is bounded.*

Proof. Each new search starts with a root cell that is not yet on any search's stack. Thus the total number of searches are bounded by the number of vertices in the graph, $|G|$. By the Uniqueness of Suspension theorem, each search can only suspend on a cell once so the number of suspensions is bounded by $|G|^2$. □

Lemma 5.3. *Assuming the algorithm runs on a finite graph G, the number of cycles and transfers that occur is bounded.*

Proof. Each cycle can be uniquely associated with the `suspend()` call resolving it since a `suspend()` call resolves at most one cycle. Since each `suspend()` call suspends a search, each cycle can be uniquely associated with a suspension. The number of cycles/transfers are therefore bounded by the number of suspensions which is shown to be bounded by the previous theorem. □

The following lemma states that if a search S is suspended on a cell c owned by a search R and reachable on c's `BlockedList`, S will eventually be unsuspended from c by exactly one thread. The lemma follows from the work done in this chapter so far. Essentially, because S is reachable on c's `BlockedList`, S will be unsuspended when the cell becomes complete. The cell c could be owned by a search in progress, a search suspended on a search in progress, or a search suspended on a chain of other suspended searches. The last search on the chain of suspended searches makes progress unless the chain develops into a cycle, in which case the cycle will be correctly resolved by the protocols described earlier and again one of the searches will make progress. These last two lemmas demonstrate that the total number of suspensions and cycles is bounded, hence a chain of suspensions and cycles cannot delay the completion of c indefinitely. We present this logic more formally in the proof below.

Lemma 5.4. *Let S be a search suspended on a cell c such that S is reachable in c's `BlockedList` and c is owned by a currently executing search R. S will be unsuspended from c by exactly one thread.*

Proof. Consider the following situations:

(a) A search S is suspended on a cell c such that S is reachable in c's `BlockedList` and c is owned by a currently executing search R.

(b) Search S is unsuspended from c by exactly one thread.

Clearly, (a) is exactly the initial situation described by the lemma while (b) is the end result. To complete the proof, we demonstrate that (a) either leads to situation (b) or a number of situations identical to (a) are encountered and must recursively be resolved before (a) results in (b). We will show that (a) can only yield an identical situation to (a) in the case of a suspension

or cycle. Because the number of suspensions and cycles is bounded by the previous two theorems, we know that (a) must eventually yield (b) as we cannot infinitely encounter new suspensions or cycles. The proof implicitly relies on the fact that all cycles are correctly detected and correctly resolved as shown by the work in this chapter. Any cycles will be resolved before the last `suspend()` call involved in the cycle returns, ensuring progress.

Let S, c and R be as in (a). If R completes c, S will be unsuspended since it is reachable on c's `BlockedList`. This yields (b) as only one thread can unsuspend S because the `compare_and_exchange` unsuspending S sets S.`cellBlockedOn` = `null`.

Let us consider the more interesting case where R suspends on another search R' before completing c. If no cycles are formed, we simply have another situation (a)'; R is a search suspended on a cell d owned by a now executing R'. Further, R will be reachable on d's `BlockedList` or resumed before the `suspend()` call suspending R returns (Theorem 5.4). We recursively apply the logic in this proof to the situation (a)' before resuming the work on (a). We can do this because the execution of (a)' cannot interfere with S without forming a cycle. If recursively following the suspensions yields a cycle that includes S, S will either become the receiving search in the subsequent transfer or S will unsuspend from c and suspend on the last cell it transfers to the receiving search. This, of course, yields (b). Lemma 5.2 ensures that the number of times we recursively follow interfering suspensions is bounded and hence R will eventually resume, complete c, and unsuspend S.

Next, assume that R is suspended before c is complete and a cycle develops. We have already explained that if the cycle involves S, (b) follows and we are done. Thus, assume S is not involved. Let R' be the receiving search in the cycle. If $R == R'$, R resumes and we have exactly the same situation as before. Thus, we assume $R \neq R'$. If c was not transferred to R', we have the same situation (a)' as above as the transfer will place R in the correct `BlockedList`. R is suspended on a cell of search R' and we must recursively wait for R to be unsuspended before continuing with (a). Because the total number of suspensions and cycles is bounded, the resolution of (a) cannot be delayed by cycles indefinitely. If c was transferred to R', then S simply waits for R' to complete c. (b) results if R' completes c after the cycle is resolved. Alternatively, R' itself could suspend before c is complete. In that case, we recursively follow the suspension of R' and use the same logic to show R' eventually unsuspends. A suspended search can only be unsuspended from c once because `cellBlockedOn` is set to another value

when the search is unsuspended. □

The following theorem is a cornerstone of the chapter; asserting the each suspended search is correctly unsuspended.

Theorem 5.10. *Any search that is suspended on any cell c will be unsuspended from c by exactly one thread.*

Proof. Let S be a search suspended on some cell c that is on the stack of another search R. Search S may have been suspended by a transfer or by a `suspend()` call. In the first case, a cycle resolution unsuspended S from another cell c_0 and immediately suspends the search on c, last cell S transfers to the receiving search R. S is inserted into c's `BlockedList` during the transfer and is clearly reachable since c cannot be complete until after the transfer. Since R is resumed after the transfer, the situation meets the conditions of Lemma 5.4, finishing the first case.

Now suppose S was suspended on by line 6 of a `suspend()` call. Suspended and reachable on c's `BlockedList`, S may be unsuspended and resumed by another thread at any time after line 6 even if the call has not yet returned. Let us begin with this case that S has been unsuspended at least once. If unsuspended by a `compare_and_exchange`, S's `cellBlockedOn` field was set to `null` ensuring S is not unsuspended more than once. Alternatively, iS could have been unsuspended and resumed as the receiving search in a cell transfer. Before the transfer is finished, the receiving search S is already unsuspended and hence any later `compare_and_exchange` fails. By Theorem 5.9, the algorithm behaves as if the `suspend()` call initially suspending S returned `SUSPEND` after line 6. Therefore, S and no other searches are erroneously unsuspended by a `suspend()` call whose actions are undone before the call returns. We have shown that if S is unsuspended before `suspend()` returns, the search is unsuspended exactly once.

Next, we study the case where S is not unsuspended by another thread before `suspend()` returns. By Theorem 5.4, S is either reachable on c's `BlockedList` after being suspended, or S will be unsuspended exactly once from c by the time `suspend()` returns. Clearly, only the first scenario requires attention. If the conditional on line 7 evaluates to true, the `compare_and_exchange` on line 8 either unsuspends S while setting S.`cellBlockedOn` to `null` or fails because another thread already unsuspended S. Either way, S is only unsuspended once. If the conditional failed, c was not complete by line 8. If c belongs to a currently running search, we

are done by Lemma 5.4. The cell c may belong to some suspended search T which is part of a chain of searches suspended on one another. In that case, we apply Lemma 5.4. to the last search in the chain, then the second to last and so on to get the desired result. If the chain develops into a cycle, the cycle will be correctly identified and resolved by our work earlier in the chapter.

Lastly, theorem 5.8 and the transfer protocol's correctness ensure that false cycle detections do not extraneously unsuspend searches. □

We have finally covered the material necessary to justify Theorem 5.1, which is restated below.

Theorem 5.11 (Restatement of Theorem 5.1)**.** *Any suspended search S will eventually be resumed. No search is ever executed by multiple threads at a time.*

N.B.: Recall that a suspend `(Search, Cell)` call returns either `RESUME` or `SUSPEND`. This determines whether the thread's `execute(Search)` call continues to execute `Search` (or the search stored at the address `Search` resided in) or the `execute(Search)` will be exited once `suspend(Search, cell)` returns.

We present an observation before the proof of theorem 5.12. A thread that creates a search S will call `execute(`S`)` to begin executing the search. The thread stops executing the search when it suspends S. Obviously, no thread should execute a suspended search until the search is unsuspended.

Observation 5.5. *A thread creating a new search calls `execute()` to execute it. The only other time a thread can start executing a search is if some thread unsuspended the search. For each unsuspension, the search is not resumed more than once.*

Proof. A thread begins executing a search in three circumstances: (a) the thread just created the search, (b) the thread just retrieved the search off the `Pending` object, (c) the thread's `suspend()` call, used to suspend the search or a previous one at the same memory address, returns `RESUME`. Note that circumstance (b) only occurs after a search is unsuspended by the `unsuspendAll()` function and placed on `Pending`. Circumstance (c) only occurs after the search is unsuspended in either line 8 or 74 of `suspend()`. Therefore, a thread only executes a search after creating it or some thread unsuspends it. □

Proof. (Theorem 5.11) Let S be an arbitrary search executed by the algorithm. Initially, only the thread T creating S executes the search through an `execute(`S`)` call. If S is never suspended, T will execute S until the search is finished and its memory is reclaimed. In this case, no other thread executes S since the search was never unsuspended. Let us now study the case where S is suspended at least once before it is finished.

S is initially suspended by Line 6 of a `suspend()` call on some cell c. By definition, from that moment, no thread is executing S. It is possible that the suspension eventually causes a cycle; Theorem 5.9 guarantees the cycle will be resolved. If S is the receiving search of that cycle resolution, S is unsuspended by line 74 and resumed by the thread that executed the cycle resolution. Thus, by Observation 5.5, only the thread resolving the cycle resumes S's execution. Now let us study the case where S is not the receiving search in the transfer. If all of S's cells are transferred to the receiving search, the search is finished and the thread running the cycle resolution will reclaim S's memory. Clearly, this will be the only (and final) thread executing S after the transfer. If S does not transfer all of its cells, the search unsuspends from c and subsequently suspends on the last cell it transfers, c'. In this case, no thread resumes execution of S so the situation is identical to the one after S first suspended on c.

Let us again consider S's initial suspension on c. We must show that exactly one thread will resume execution of S. By theorem 5.9, S will be unsuspended by exactly one thread. Thus, it is sufficient to show that no matter how S is unsuspended, exactly one thread will resume its execution. In the last paragraph, we demonstrated that if the unsuspension is a result of S being the receiving search of a transfer or of S being finished during a transfer, S is resumed by exactly one thread. S may also be unsuspended by a cycle resolution just to be subsequently suspended on another cell. We showed above that this suspension on c' is identical to the original (no thread is executing S) and hence we can recursively apply the logic in this proof to that suspension. Lemma 5.2, of course, prevents us from infinitely following a trail of suspensions. There are two other ways c could have been unsuspended.

First, S could have been unsuspended by the `unsuspendAll()` call following c's completion. The successful `compare_and_exchange()` unsuspending S leads to the search being placed on the `Pending` object. Afterward, S is executed by the thread retrieving it from `Pending`.

Second, S could have been unsuspended by line 8 of a `suspend()` call. S

resumes after this successful `compare_and_exchange` call causes `suspend()` to return `RESUME`.

We have shown that S will eventually be executed by exactly one thread after its initial suspension on c. From the previous paragraphs, it is apparent that S is no longer suspended when it is resumed. By Observation 5.5, no extraneous threads will attempt to execute S will it is being run by the current thread. If S later suspends on another cell d, we follow exactly the same logic to show exactly one thread resumes S's execution. Repeatedly applying this logic to any suspensions encountered guarantees that some thread will eventually resume execution of S following any suspension yet no more than one thread ever executes S at a time. □

Corollary 5.1 (Restatement of Corollary 3.1)**.** *A thread T that begins executing a search S will continue to do so until the search suspends or until the search is finished and its memory is recycled. No other thread executes S from the moment T begins executing the search until T finishes executing the search.*

Proof. Following the pseudocode for `execute()`, it is evident that the function returns only if `suspend()` is called and returns `SUSPEND` or the search is finished and hence recycled. The second statement follows directly from Theorem 5.11. □

Lastly, we examine the functions and data structures associated with suspension, unsuspension, and resumption to show these are wait-free. `unsuspend()` iterates through a cell's `BlockedList` and places any of the searches it was able to unsuspend on the `Pending` object. The number of searches on the `BlockedList` is finite by the same logic that demonstrates that the number of suspensions is finite. In chapter 6, we show that any interaction with a cell's `BlockedList` is wait-free. From code 5.3, it follows that `unsuspend()` is wait-free since `compare_and_exchange()` is atomic and *Pending* can easily be implemented using a wait-free queue. Resuming searches consists of verifying that *Pending* is non-empty and retrieving a search from the object; this is wait-free using a wait-free queue/stack implementation of *Pending*.

Next, we examine `suspend()` whose code appears on code 5.2 above. Each line in the function is completed in a bounded number of steps, including the calls to cell member functions such as `getOwner()` on line 33 and

search member functions such as `getBlockingCell()` as they are simple getter/setter functions that do not use locking. The `transferCells()` function (code 5.1) invoked by `suspend()` is wait-free as it does not use any locks and it is completed in a number of steps proportional to the number of cells being transferred. Lastly, `suspend()` has two while-loops that build and verify a potential cycle. If we assume the graph is finite, the number of suspensions is bounded and hence the path cannot indefinitely be extended. If the graph is infinite, it is technically possible for new suspensions to extend the path built in the first while-loop quicker than the path is traversed. The method is still lock-free on an infinite graph because the path is only extended by another suspension succeeding elsewhere. Therefore, `suspend()` is wait-free (lock-free) on a finite (infinite) graph.

Chapter 6

The `BlockedList`

6.1 High Level Overview and Discussion

We have referred to the `BlockedList` data structure without providing many implementation details. We fill in that gap in this chapter. Recall, a cell's `BlockedList` object records the searches that may be suspended on the cell. An implementation could simply use a standard library list data structure wrapped with a mutex to prevent concurrent access. This is the approach taken by [3]. Because the algorithm commonly accesses `BlockedList` objects, we were motivated to design an optimized wait-free `BlockedList`.

Since each cell has its own `BlockedList`, optimizing the list's memory footprint is crucial for maintaining scalability. Conversely, since cell objects are reused, many cells will eventually need to store suspended searches. Allocating the necessary memory during runtime requires expensive synchronization (i.e. a `compare_and_exchange()`). Observing that a `BlockedList` almost never contains more than eight entries at a time, we give the `BlockedList` an initial capacity of eight and optimize the data structure to make accessing those first eight elements extra efficient. We optimize by making this `firstBuffer` array a member variable of `BlockedList`, thus the memory is automatically allocated upon `BlockedList`'s creation and accessing this memory avoids pointer indirection.

If necessary, the `BlockedList` can grow its capacity by allocating additional arrays to store search pointers. The buffers array holds pointers to these additional buffers. Memory for the additional buffers or the buffers array is allocated on demand, rather than initially, since it is rarely needed. We recognize that different applications may place different priority on speed

or memory. Therefore, we employ C++'s templates to allow users to modify the initial capacity of the data structure and how fast the capacity grows.

Theorem 5.5 above states that arbitrary elements can be inserted into the `BlockedList` without modifying the output of the algorithm. The fact holds partly because we verify if searches are actually suspended on the newly complete cell before unsuspending them. This result allows important optimizations to the `BlockedList`. For instance, the data structure does not need to support deletions since leaving items in the list has no effect. This allows us to simplify the implementation and avoid the potentially expensive synchronization of deletions and reads.

6.2 Implementation Details

We present code for the data structure in code 6.1 below. New entries are inserted into the back of the list; i.e. the first entry has index `0`, the second has index `1`,..., etc.

The atomic `head` member variable records the index of the last element inserted into the list.

The `push_back()` function handles insertions (invoked on line 37 of `transferCells()` and 5 of `suspend()`). The function calls `head.getAndIncrement()`, which increments head and returns *idx*, head's previous value. The head variable is the crux of our synchronization. Because the `getAndIncrement()` is atomic and head is not decreased during a cell's life, no two `push_back()` calls receive the same value of *idx*. We summarize this result below.

Observation 6.1. *During a cell's lifetime, no two calls to* ***`push_back()`*** *attempt to write elements to the same location.*

The rest of the function deals with inserting the element at the location represented by `idx`. Since `BlockedList` is implemented with non-contiguous arrays, we must find out which array `idx` is in and what location it corresponds to. In the usual case where `idx`< 8, we can optimize away the calculations by simply inserting at `firstBuffer[idx]`. Otherwise, `calcBufferToInsertIn()` does the necessary calculations.

If `push_back()` must insert into an auxiliary buffer and the buffer isn't allocated, `buffers.compare_and_exchange(null, new Buffer)` is used create the buffer and add its pointer to the buffers array. Using the

`compare_and_exchange()` is important because several threads may try to set the pointer simultaneously. The insertion on line 40 of the `BlockedList` class is safe because the buffer must exist after the `compare_and_exchange()` even if it fails since that would mean another thread created and added the buffer first. Notice the buffers array itself must be created and set with a `compare_and_exchange()` before additional buffers can be added.

The content of the `BlockedList` is used by the function `unsuspendAll()` when a cell becomes complete. The `BlockedList`Iterator is used to iterate through the `BlockedList`; it makes necessary calculations such as jumping from one buffer to the next. Consider the situation where head is incremented to `9`, but either the buffers array or buffers[1] is not yet set when `unsuspendAll()` scans the list. Before accessing the relevant arrays, the iterator ensures that they have actually been added by verifying that `buffers` $\neq$ `null` and `buffers[1]` $\neq$ `null`. As an optimization, if the list has fewer than `8` elements, `unsuspendAll()` simply iterates through the `firstBuffer` to avoid unnecessary overhead.

Due to the limited synchronization, if threads call `unsuspendAll()` and `push_back()` concurrently, the iterator may pass an entry in the `BlockedList` before it is even added. We explained how this situation is handled by Theorem 5.4. Essentially, lines 7-12 in `suspend()` make sure the search is unsuspended in case the cell became complete during the `suspend()` call.

When a cell is recycled, the `reset()` function clears each buffers' entries and sets head to zero. We have a race condition if `push_back()` and `reset()` are called simultaneously; for instance, `push_back()` could set the head variable back to a non-zero integer. We will demonstrate that this is impossible. `push_back()` is can be invoked in two cases; on line 5 of `suspend()` before suspending a search or during a cell transfer. Of course, a cell cannot be recycled during a cell transfer since none of the involved cells can become complete until the cycle is resolved. Recall that `suspend()` is called to suspend a search on a conflict cell referenced in `execute()`. Partial reference counting ensures that a cell cannot be recycled while any thread has a reference to the cell in its `execute()` call. Thus, it is impossible for `push_back()` and `reset()` to be invoked simultaneously. The fact that `push_back()` calls return before is recycled also ensures that `push_back()` calls from different lifetimes of the cell object do not interfere with each other. Clearly, it is crucial that `reset()` occurs strictly later than `unsuspendAll()`. Therefore, a thread completing a cell releases its reference on the cell only after calling

`unsuspendAll()` to circumvent the error.

Finally, we argue that our implementation is wait-free. Resetting and iterating through the `BlockedList` is wait-free. The `push_back()` function uses a few atomic synchronizations; a `getAndIncrement()` on head and a potential `compare_and_exchange()`. While the outcome of the operations depends on the actions of other threads, the function proceeds regardless of what is returned by the operations. For instance, the function proceeds with insertion even if its `compare_and_exchange()` fails to add a buffer because some other thread must have successfully added a buffer for the operation to fail. Execution of `push_back()` is linear; there are no loops or `GOTO` statements that could result in execution of earlier lines of code. Thus, `push_back()` is completed in a bounded number of steps.

Code 6.1

```
1  class BlockedList{
2    const static int BASE = 8;
3    //By default BASE = 8, can be set to another power of 2
4    atomic<int> head = 0;
5    Search* firstBuffer[BASE];
6    //Atomic pointer to array of extra buffers,
7    //which are allocated if needed
8    atomic<AtomicPointerArray> extraBuffers;
9    void push_back(Search* search){
10     int idx = head.getAndIncrement();
11     //Retrieve index to insert item and increment head
12     //if index is less than base, we just insert
13     //the search in the correct location of firstBuffer
14     if(idx < BASE){
15       firstBuffer[idx] = item;
16       return;
17     }
18     int (buffer, pos) = calcBufferToInsertIn(idx);
19     //calculate the auxilary buffer we need to insert into
20     //and the index of the new buffer we must insert into
21     //Initialize the array of auxilary buffers
22     //if it has not yet been initilized
23     //We also initialize the first auxilary buffer
24     //right away to avoid doing two compare_and_exchanges
25     //since we would have had to do it anyway
26     if(buffers == null){
27       AtomicPointerArray bufferArray = new AtomicPointerArray
28           [NUM_BUF]();
29       bufferArray[0] = firstBuffer; //For completeness
```

```
30          bufferArray[1] = new Search*[BASE^2];
31          //Attempt to set the array of auxiliary buffers,
32          //if the step fails
33          //another thread has created the buffer array
34          //so we can proceed either way
35          buffers.compare_and_exchange(null, bufferArray);
36        }
37        //Allocate the buffer we plan to insert search
38        //into if it has not yet been created
39        if(buffers[buffer] == null)
40          buffers[buffer].compare_and_exchange(
41              new Search*[BASE * (buffer+1)], newBuffer);
42        buffers[buffer][pos] = item; //Insert the item
43      }
44      //Reset the contents of the existing
45      //buffers to null when the cell is reset
46      void reset(){
47          for(Buffer buffer: buffers)
48          if(buffer is allocated)
49            fill buffer with null values;
50      }
51    };
```

Chapter 7

Experimental Results

7.1 Experimental Setup

Graphs

A random graph generator is useful to appropriately test and benchmark any graph algorithm. Further, the correctness and performance of an algorithm should be verified under different conditions that it may experience in real life applications. Hence, our graph generator should be flexible and be able to produce graphs with a wide variety of properties.

A rather obvious approach to generating random graphs starts with creating `desiredSize` vertices. Then, for any two vertices, v and w, we form an edge from v to w with probability `edgeProb`. This simple implementation allows us to customize two properties we are very much interested in: graph size and density. This is the first random graph generator we used.

The problem with these graphs is that most of their vertices are typically held by one SCC and the rest of the vertices are in trivial SCCs. As explained by papers such as [3] and [9], concurrent depth-first search algorithms perform very poorly on those types of graphs. For instance, our concurrent searches would continually collide with one another in the large SCC, leading to a large overhead. [3]'s results show the algorithm running over twice as slowly on the graph *tring*2.1, which has a large SCC taking up about two thirds of the graph, when using 8 threads compared to one thread. Our implementation seems to perform a bit better on these graphs than [3] (we either get a slight speed up or slight slowdown), but not enough to warrant using this algorithm over another concurrent SCC algorithm. Thus, while the first two random graph generators were useful for testing the implementation's correctness, they are not useful for bench-marking because other SCC algorithms should

be run on these graphs.

Our second random graph generator creates n clusters of vertices of a given density. Each cluster usually has one large SCC containing most of its vertices and several trivial SCCs. The clusters are then connected together into a graph in such a way that if cluster x has a path to cluster y, there is no path from cluster y to x otherwise they would merge into one SCC. This represents a typical graph the concurrent Tarjan's algorithm should be run on; a graph with many SCCs of different sizes and no SCC containing a significant portion of the graph's vertices (no SCC is larger than any of the clusters). The graphs $Clust\_1M$, $Clust\_10M$, $Clust\_25M$, and $Clust\_50M$ appear in our benchmarking. The graph $Clust\_xM$ has x million vertices total and 1000 vertices per cluster. The remainder of our benchmarking is done on CSP graphs from the FDR3 test suite. The majority of these graphs are the same graphs that [3] tested their algorithm on. We omitted some of the graphs [3] benchmarked because we felt it used too many graphs with only trivial SCCs. We also added some larger CSP graphs that [3] did not benchmark on since we are primarly interested in whether this algorithm could speed up computation of large graphs. We benchmark on 33 graphs in total.

Environment

To ensure robustness across environments, we benchmarked our algorithm on two devices. The first device (Mac) was a Mac Pro with a 3.0GHz 8-Core Intel Xeon E5 processor with 25MB L3 cache and 16 GB of RAM running MacOS Sierra. The second device (Linux) has a 2.0 GHz 16-core intel Xeon Processor E5-2650 with 20 MB L3 cache and 264.6 GB of RAM running Debian GNU/Linux 8.9. Both machines support hyper-threading with two threads for each core. The code was compiled with `GCC 4.9.2` with the `-std=c++14` flag and the `-O3` optimization flag. The graphs have anywhere from 200 thousand to 50 million vertices.

Experiments

Each execution time or memory benchmark appearing in this chapter is the average of 50 executions. The performance of the algorithm is tested using the Mac and Linux environments with various numbers of threads. Most of the tests are on the CSP graphs, a few are on the cluster graphs described above. We also benchmark the algorithm on a subset of the graphs with and without memory recycling to investigate the effectiveness of that technique in reducing memory use and execution time.

7.2 Hash Table

A concurrent depth-first search algorithm consists of several threads simultaneously exploring new edges of the graph. Threads use a shared hash table to map the edges' receiving vertices to shared cell objects storing information about the vertices. Since threads must access the hash table each time they explore a new edge or start a search, it is no surprise that the hash table is the main bottleneck in both our implementation and [3]'s. This dissertation and Lowe [3] each test out several hash tables in an attempt to minimize the bottleneck. The fastest hash tables accounted for around 50% and 20% of the execution time in the respective implementations with slower hash tables taking significantly longer. Using the most obvious implementation of a concurrent map, a hash table wrapped in locks, the algorithm performs very poorly, even worse than the sequential. That implementation is useless because it only allows one thread to access it at a time; the remainder of the hash tables in this section use various techniques to allow concurrent access into the map.

In [3], the best performing non-resizable hash table is based on open addressing and lightweight locking. The best performers permitting resizing employ sharding, a technique dividing a map into multiple sub-maps (shards) so that each shard can be synchronized independently. Then, if a shard needs to be locked due to an insertion or resize, all of the other shards can remain unlocked, avoiding a bottleneck. We use these results from [3] to guide our own search for a hash table implementation minimizing the bottleneck. We briefly discuss the maps we tried out below.

The number in parenthesis following each map's name gives the ratio of average run-time of the algorithm on the bench-marking graphs with 8 threads when using that map compared to the best performing map.

TBB concurrent_unordered_map (2.775) This hash table is part of Intel's *Thread Building Blocks* concurrent software library [17]. The map does not support deletions (which we do not need) in order to provide an optimized implementation does not use any visible locking.

cuckoohash_map (1.912) The highly concurrent table, from [18], was shown to be both space and time efficient compared to many other open source concurrent maps. It uses uses per-bucket locking and cuckoo hashing to resolve collisions.

Sharded_Lock (2.306) Motivated by the success of sharding in [3], we developed a simple sharded hash table which uses the C++ standard library

std::unordered_map for each shard. An entry is placed into a shard based on the last n bits of its hash key and then hashed into a location in that shard based on the rest of the hash key. Access to shards is guarded with a mutex lock.

Sharded_SpinLock (2.242) Accessing an entry in a hash table is relatively fast, hence it makes more sense for a waiting thread to spin rather than use a performance heavy operating system lock. Therefore, this variant of `Sharded_Lock` guards shards with light weight spin-locks instead of mutexes.

Open_Addressed (3.112) The concurrent hash tables considered so far build upon open source or standard library maps. Instead of using these general purpose containers, it might be to our advantage to build an in-house hash table optimized for this specific application. Namely, we do not need to support concurrent deletions and each entry consists of a vertex identifier and a `WeakReference`.

We use the open addressing technique on an atomic array to resolve collisions (if the desired location is taken, select the next available entry in the table). A `compare_and_exchange(empty_entry, new_entry)` is used for thread-safe insertions into the table. These lock-free insertions and retrievals are easy to implement since we not support concurrent deletions and hence a deletion cannot invalidate the location we choose using open addressing. Resizing is not lock-free; when the load factor exceeds 50% a thread will lock the table are resize it. Resizing could be made lock-free and perhaps more efficient by allowing other threads to assist with resizing; we did not pursue this.

Open_Sharded (1.000) This technique uses many shards, each of which is an `Open_Addressed` table from above. The advantage of having multiple shards is reduced contention and resizing without locking out threads accessing other shards.

The results show our choice of hash table significantly affects the algorithm's performance; the best performer results in almost half the execution time than when using the second best map. The results buttress [3]'s finding that sharding is a very helpful technique to produce productive hash tables for this algorithm. Further, the maps perform better when we use many shards (512 to 8096); the above experiments were performed with 8096 shards.

It is surprising the sharded open addressed table outperforms the regular version so significantly given that most accesses are lock-free. The open addressed map uses several atomic accounting variables to track its size, capacity and how many threads are currently using the map. Shard-

ing the map probably reduces contention on those variables, contributing to the speed up (`compare_and_exchange` is slowed by contention due to increased chance of failure and atomic increments are often implemented with `compare_and_exchange`). Overall, we believe `Open_Sharded` yields the best results because it uses a single atomic array to store the entries which makes resizing and deallocation quicker, allows lock-free access, is memory efficient, and employs sharding to prevent locking out other threads during resizing.

We tried two interesting techniques to speed up the maps. The map is most often accessed by the `initializeNeighbors()` function which retrieves the cells corresponding to each neighboring vertex of a cell. For the `Sharded_Lock` and `Sharded_SpinLock` maps, a thread encountering a locked shard would attempt retrieve a different cell from another shard first and then return to that cell later. This only leads to very modest improvements, however. This is probably because our maps use so many shards that contention is already fairly rare and hence performance can only be improved by upgrading the map's overall implementation.

The second technique has each thread maintain a thread local hash table. To retrieve a cell, a thread first probes its thread local table. A thread only accesses the shared memory hash table if the thread local table does not contain the value. Afterwards, the value is added to the thread local table so it can be retrieved locally next time.The technique increased performance on very slow concurrent maps, but slows down the algorithm when we use the efficient maps presented this section. We probably do not see a performance improvement for the maps in this section because most of them use sharding and other techniques which already reduce contention to the point where contention is not the map's main bottleneck. Additionally, this technique may prove more useful on very dense graphs, but our intended application, FDR model checker, tends to use very sparse graphs. For graphs with certain properties, a modified version of the technique where the thread-local table acts as a cache (removes old values when it gets to large) could yield useful performance results. The thread local map method is unlikely to be useful for CSP purposes due to those graphs being very sparse.

7.3 Evaluating Memory Recycling

In this section we examine whether our memory recycling techniques provide sizeable improvements in either memory use or runtime. We test on

Figure 7.1: Memory usage with and without recycling

Graph	Vertices	Memory, Gig				Cells Used	Searches used		Ratio (without/with recycle)		
		without recycling	Cell/Search recycling	Cell recycling	Seach recycling	with recycling	without recycling	with recycling	Cells used	Seaches used	Memory used
alt10.4.0	202,512	0.09	0.04	0.05	0.08	360	107,812	44	562.53	2,450.27	2.14
alt10.2.3	1,514,882	0.59	0.29	0.32	0.57	7,313	151,896	130	207.15	1,168.43	2.01
alt10.3.0	7,676,754	3.08	1.42	1.77	2.46	1,379	2,273,762	267	5,566.90	8,515.96	2.17
alt10.3.3	7,830,225	3.04	1.43	1.72	2.76	1,442	2,412,820	254	5,430.11	9,499.29	2.13
alt10.3.4	7,830,225	3.14	1.51	1.80	2.83	1,484	2,412,616	350	5,276.43	6,893.19	2.08
alt10.3.1	20,710,362	8.64	4.44	4.93	7.98	2,141	5,022,781	353	9,673.22	14,228.84	1.95
alt10.2.0	36,999,926	16.17	8.77	10.90	14.76	1,826	15,319,932	595	20,262.83	25,747.78	1.84
Clust_1M	1,000,000	0.97	0.71	0.94	0.81	61,847	146,203	6,058	16.17	24.13	1.37
Clust_25M	25,000,000	5.24	1.87	2.40	4.61	310,842	1,185,521	39,638	80.43	29.91	2.80
Clust_50M	50,000,000	24.85	7.91	9.95	21.23	416,791	5,246,731	55,343	119.96	94.80	3.14

7 CSP graphs and 3 cluster graphs using the Mac environment. Figure 7.1 displays the results of these tests. On the left, the figure shows the heap memory used with recycling disabled, search recycling enabled, cell recycling enabled and both recycling mechanisms enabled. On the right side, the figure demonstrates how the number of cell and search objects used changes with recycling enabled. Note that the figure omits the number of cells used without recycling as that is simply equal to the number of vertices.

On average, the algorithm used 2.04 and 2.44 times more memory without recycling enabled on CSP and cluster graphs respectively. As we can see from the figure, the result that memory cycling approximately halves the memory footprint seems fairly robust for CSP graphs all various sizes. For the cluster graphs, memory recycling appears to save more memory on larger graphs. The differences are due to the nature of the graphs; CSP graphs have many small and trivial SCCs while cluster graphs are made up of many similarly sized large SCCs and trivial SCCS. The memory savings also depend on the underlying implementations of the search and cells objects. Before we switched to more memory efficient implementations of the tarjan and controlStacks, the savings from search recycling were far larger.

Let us try to interpret the results. On one hand, halving the implementation's memory footprint certainly is not revolutionary and will not effect the space complexity or overall growth in memory usage as graphs get larger. On the other hand, we do not think the contribution is trivial. Memory recycling reduces the number of searches used by an approximate factor of 2,400 to 25,000 and the number of cells used by a factor of 207 to 20,000 for CSP graphs. Cell and search object creation is reduced by a factor of 16 to 120 for cluster graphs.

Most importantly, it appears we have reduced the growth of cell/search

Figure 7.2: Performance with and without memory recycling

Graph	Vertices	Speed (vs recycling)		Ratio
		without	with	
alt10.4.0	202,512	0.10	0.07	1.39
alt10.2.3	1,514,882	0.46	0.34	1.34
alt10.3.0	7,676,754	2.08	1.45	1.43
alt10.3.3	7,830,225	2.12	1.56	1.36
alt10.3.4	7,830,225	2.10	1.42	1.48
alt10.3.1	20,710,362	6.37	4.76	1.34
alt10.2.0	36,999,926	37.42	9.34	4.01
Clust_1M	1,000,000	0.58	0.46	1.25
Clust_25M	25,000,000	15.23	8.60	1.77
Clust_50M	50,000,000	90.32	18.84	4.79

object use from linear to logarithmic as a function of graph size. While this only improves the memory footprint by a constant factor, it means that we have significantly reduced the number of heap allocations. As mentioned in 4.1, in practice heap allocations usually use locking to ensure thread safety which can introduce a sequential bottleneck since the number of searches and cells allocated without recycling is both linear with respect to vertices. CSP graphs consist mostly of trivial, small and isolated SCCs. This means new searches (along with new tarjan and control stacks) would have to be constantly allocated for CSP graphs. This assertion is supported by figure 7.1; the number of searches allocated without recycling is only a constant factor fewer than the number of vertices on CSP. Recycling can also have cache friendly behavior as we reuse the most recently reclaimed searches and cells.

Figure 7.2 displays the run-time (in seconds) of the algorithm with and without recycling and the overall speed up resulting from recycling. Ignoring the two outliers, the speed up averages about 1.42 without significant variation which is a fairly reasonable speed up considering we are just changing memory management techniques rather than the underlying algorithm. The largest graphs are outliers with a speedup exceeding 4. This is because the algorithm exceeded RAM space and hence was significantly slowed from having to read/write to disk. Running the algorithm on the Linux environment which has 264.6 GB of RAM, we saw an average speedup from 1.50 for the two outlier graphs, which is more in line with the 1.42 average speed up for

the rest of the graphs. Thus, while the outliers don't prove that the speed up generally increases with larger graphs, they demonstrate that memory recycling could yield large speed ups when handling graphs that exceed the machine's RAM limits by decreasing the memory that needs to be stored in disk.

Figure 7.1 indicates that cell recycling accounts for the majority of the memory footprint reduction. Similarly, informal profiling suggests that cell recycling accounts for an average of 80% of the speedup associated with memory recycling. We believe search recycling is still worth it as it is much easier to implement and has less overhead than cell recycling.

Lastly, we mention a caveat with these results. We benchmarked the algorithm with memory recycling against an implementation that simply deletes all the memory at the end. It is certainly possible that using reference counting or other techniques, an implementation could efficiently delete during run-time. This could significantly improving base case's performance by reducing the maximum memory footprint and the number of objects deallocated at the end (sequential bottleneck). An issue with this approach is that implementing lock-free reference counting for deleting objects has much more overhead than when the objects are recycled. In situations where memory is bountiful, a fruitful alternative approach may to be allocate cells and searches in batches of increasing size and delete them at the end. Allocating the objects in bulk should speed up allocations and deallocations and avoids frequently locking the heap. This requires more memory than our approach, but it would avoid the overhead associated with reference counting and `age` tracking.

7.4 Benchmarking Results

Finally, we benchmark the algorithm's performance against the sequential on both the Mac and Linux environments. The machines have 8 and 16 cores respectively and support twice as many threads with hyper-threading (full details earlier). All benchmarking is done using the `openSharded` hash table described earlier. The map initially has 4096 shards except the smallest 11 graphs for which the map initially has 256 shards (because for small graphs it takes more time to initialize the 4096 shards than to run the algorithm). Figure 7.3 displays the average run times of 50 executions on each graph using the Linux environment with 1 (sequential), 2, 4, 8, 16, 32

threads. The right side of figure 7.3 shows the relative speedups compared to the sequential algorithm; the sequential runtime divided by the runtime with n threads. Figure 7.4 displays the analogous data for the Mac system except benchmarked only to 16 threads since the system only has eight physical cores. Figure 7.5 gives a graphical representation of the two charts.

There are three main types of graphs. The graphs whose names begin with "*Clust*" were randomly generated by the clustered graph generator described in section 7.1. Each vertex has about 6 edges and the graph contains many SCCs with cardinality around 1000 in addition to numerous small SCCs. The rest of the graphs are CSP graphs. The CSP graphs not starting with "*alt*" only contain trivial SCCs. The graph "*matmul*.6" has no edges at all (scaling is below average on *matmul*.6 since new searches start by retrieving a vertex by incrementing a shared atomic counter which would be slowed by contention if there are no edges to explore - this can easily be fixed if we expect to encounter many graphs like this). The graphs that have names starting with "*alt*" are typical CSP graphs containing many small or trivial SCCS and averaging 0-6 edges per vertex.

Figure 7.3: Linux Benchmarks with Different Thread Counts

Graph	Vertices	Time, sec						Time (N)/Time (1)				
		1.00	2.00	4.00	8.00	16.00	32.00	x2	x4	x8	x16	x32
alt10.2.0	36,999,926	60.23	34.80	22.30	17.06	13.47	7.19	1.73	2.70	3.53	4.47	8.37
alt10.2.4	36,974,589	58.98	33.78	21.40	16.26	15.14	8.45	1.75	2.76	3.63	3.89	6.98
Clust_25M	25,000,000	35.60	32.81	19.98	13.42	10.11	6.44	1.08	1.78	2.65	3.52	5.53
alt10.3.1	20,710,362	34.46	17.69	11.52	8.93	7.01	4.24	1.95	2.99	3.86	4.92	8.12
Clust_10M	10,000,000	13.62	12.13	7.37	5.07	3.79	2.42	1.12	1.85	2.69	3.59	5.63
alt10.3.3	7,830,225	15.65	5.97	3.91	3.13	2.60	1.49	2.62	4.00	5.01	6.02	10.51
alt10.3.4	7,830,225	10.80	5.76	3.95	3.19	2.67	1.51	1.87	2.74	3.39	4.04	7.15
alt10.3.0	7,676,754	11.19	5.81	3.91	3.17	2.60	1.47	1.93	2.86	3.54	4.30	7.61
solitaire.1	4,001,297	2.52	3.82	2.38	1.76	1.41	1.14	0.66	1.06	1.43	1.78	2.21
alt10.3.2	2,324,588	3.62	1.81	1.36	1.13	0.92	0.49	2.00	2.67	3.20	3.95	7.42
matmul.6	2,252,800	0.65	1.15	0.85	0.62	0.57	0.36	0.56	0.76	1.04	1.13	1.78
alt10.2.2	1,514,882	2.17	1.38	1.01	0.86	0.73	0.39	1.57	2.15	2.52	2.99	5.63
alt10.2.3	1,514,882	2.21	1.40	1.04	0.89	0.77	0.40	1.58	2.13	2.48	2.87	5.55
comppuz.0	1,235,030	1.00	0.86	0.63	0.53	0.45	0.28	1.16	1.58	1.89	2.23	3.52
Clust_1M	1,000,000	1.12	1.24	0.75	0.53	0.42	0.30	0.90	1.49	2.11	2.69	3.76
alt11.3.0	990,167	1.57	0.84	0.62	0.57	0.46	0.25	1.88	2.52	2.74	3.39	6.17
alt10.3.7	990,167	1.54	0.84	0.60	0.53	0.47	0.27	1.83	2.55	2.91	3.31	5.70
alt11.3.1	990,167	1.20	0.83	0.61	0.54	0.48	0.27	1.44	1.96	2.23	2.48	4.50
alt10.3.5	990,167	1.07	0.81	0.60	0.52	0.46	0.24	1.32	1.80	2.06	2.33	4.44
alt10.3.6	990,167	1.14	1.01	0.61	0.47	0.39	0.32	1.12	1.87	2.43	2.90	3.57
soldiers.0	714,480	0.37	0.62	0.37	0.28	0.23	0.19	0.59	0.98	1.29	1.56	1.93
cloudp.0	691,692	0.38	0.49	0.34	0.28	0.26	0.17	0.76	1.11	1.32	1.45	2.27
alt11.2.1	589,149	0.93	0.55	0.40	0.44	0.27	0.17	1.69	2.31	2.13	3.40	5.59
alt10.2.6	589,149	0.69	0.53	0.39	0.32	0.26	0.15	1.31	1.79	2.16	2.65	4.61
alt10.2.7	589,149	0.94	0.67	0.41	0.31	0.26	0.22	1.40	2.26	2.99	3.65	4.34
alt11.2.0	589,149	0.83	0.66	0.40	0.29	0.23	0.20	1.27	2.11	2.89	3.56	4.21
solitaire.0	494,372	0.37	0.68	0.39	0.26	0.19	0.14	0.54	0.94	1.40	1.97	2.57
cloudp.2	480,984	0.29	0.41	0.29	0.24	0.22	0.13	0.72	1.01	1.23	1.33	2.23
virtroute.2	390,625	0.34	0.52	0.31	0.23	0.17	0.13	0.65	1.08	1.49	1.98	2.66
alt12.2.1	344,221	0.49	0.36	0.28	0.23	0.19	0.11	1.36	1.78	2.18	2.66	4.39
alt10.4.1	289,570	0.30	0.29	0.23	0.23	0.18	0.10	1.03	1.31	1.29	1.68	2.98
alt10.4.0	202,512	0.16	0.19	0.12	0.09	0.07	0.05	0.80	1.25	1.71	2.09	2.84
alt10.4.3	201,977	0.19	0.19	0.12	0.09	0.07	0.06	0.99	1.55	2.07	2.48	3.21

Figure 7.4: Mac Benchmarks with Different Thread Counts

Graph	Vertices	Time, sec					Time(N)/Time(1)			
		1.00	2.00	4.00	8.00	16.00	x2	x4	x8	x16
alt10.2.0	36,999,926	47.424	23.4252	14.242	7.823	5.4342	2.02	3.33	6.06	8.73
alt10.2.4	36,974,589	44.1343	19.5048	11.9728	7.6813	6.0402	2.26	3.69	5.75	7.31
Clust_25M	25,000,000	29.3577	14.2398	9.17424	5.5762	4.56055	2.06	3.20	5.26	6.44
alt10.3.1	20,710,362	28.242	13.414	7.231	5.124	3.768	2.11	3.91	5.51	7.50
Clust_10M	10,000,000	11.6741	5.59123	3.64472	2.2315	1.838	2.09	3.20	5.23	6.35
alt10.3.3	7,830,225	8.3242	4.372	2.092	1.013	0.783	1.90	3.98	8.22	10.63
alt10.3.4	7,830,225	8.37105	3.6131	2.035	1.322	1.021	2.32	4.11	6.33	8.20
alt10.3.0	7,676,754	8.304	3.08332	1.9434	1.2842	0.94347	2.69	4.27	6.47	8.80
solitaire.1	4,001,297	1.70218	1.26919	0.87319	0.5981	0.48006	1.34	1.95	2.85	3.55
alt10.3.2	2,324,588	2.34417	0.95753	0.63858	0.4529	0.34759	2.45	3.67	5.18	6.74
matmul.6	2,252,800	0.4723	0.5532	0.3424	0.2324	0.1932	0.85	1.38	2.03	2.44
alt10.2.2	1,514,882	1.46201	0.70298	0.47147	0.3357	0.26191	2.08	3.10	4.36	5.58
alt10.2.3	1,514,882	1.42381	0.69029	0.46539	0.3333	0.26113	2.06	3.06	4.27	5.45
comppuz.0	1,235,030	0.55081	0.48735	0.38439	0.2935	0.23769	1.13	1.43	1.88	2.32
Clust_1M	1,000,000	1.10414	0.59013	0.41892	0.293	0.26719	1.87	2.64	3.77	4.13
alt11.3.0	990,167	0.92815	0.43197	0.30449	0.2329	0.18947	2.15	3.05	3.99	4.90
alt10.3.7	990,167	0.90122	0.43471	0.30724	0.2354	0.19381	2.07	2.93	3.83	4.65
alt11.3.1	990,167	0.92354	0.43714	0.30818	0.2352	0.19472	2.11	3.00	3.93	4.74
alt10.3.5	990,167	0.90904	0.43153	0.30639	0.2338	0.19058	2.11	2.97	3.89	4.77
alt10.3.6	990,167	0.90025	0.43507	0.30784	0.2354	0.19384	2.07	2.92	3.82	4.64
soldiers.0	714,480	0.29908	0.25909	0.20399	0.1681	0.15169	1.15	1.47	1.78	1.97
cloudp.0	691,692	0.231	0.3231	0.2121	0.1823	0.1732	0.71	1.09	1.27	1.33
alt11.2.1	589,149	0.5132	0.30432	0.18242	0.1381	0.1023	1.69	2.81	3.72	5.02
alt10.2.6	589,149	0.51857	0.29599	0.17876	0.1333	0.0923	1.75	2.90	3.89	5.62
alt10.2.7	589,149	0.4832	0.3005	0.18152	0.1363	0.095	1.61	2.66	3.54	5.09
alt11.2.0	589,149	0.5323	0.302	0.18322	0.1346	0.1142	1.76	2.91	3.95	4.66
solitaire.0	494,372	0.1442	0.2341	0.1484	0.0996	0.08932	0.62	0.97	1.45	1.61
cloudp.2	480,984	0.1436	0.14911	0.09529	0.0725	0.0613	0.96	1.51	1.98	2.34
virtroute.2	390,625	0.1342	0.1953	0.10442	0.0923	0.07943	0.69	1.29	1.45	1.69
alt12.2.1	344,221	0.2743	0.1932	0.1132	0.0704	0.06132	1.42	2.42	3.89	4.47
alt10.4.1	289,570	0.1823	0.12323	0.08	0.0613	0.05321	1.48	2.28	2.97	3.43
alt10.4.0	202,512	0.0869	0.0673	0.0421	0.0324	0.02734	1.29	2.06	2.68	3.18
alt10.4.3	201,977	0.0861	0.06011	0.0415	0.0331	0.02832	1.43	2.07	2.60	3.04

Figure 7.5: Linux (top) and Mac (bottom) Speed Ups over Sequential as a Function of Thread Count

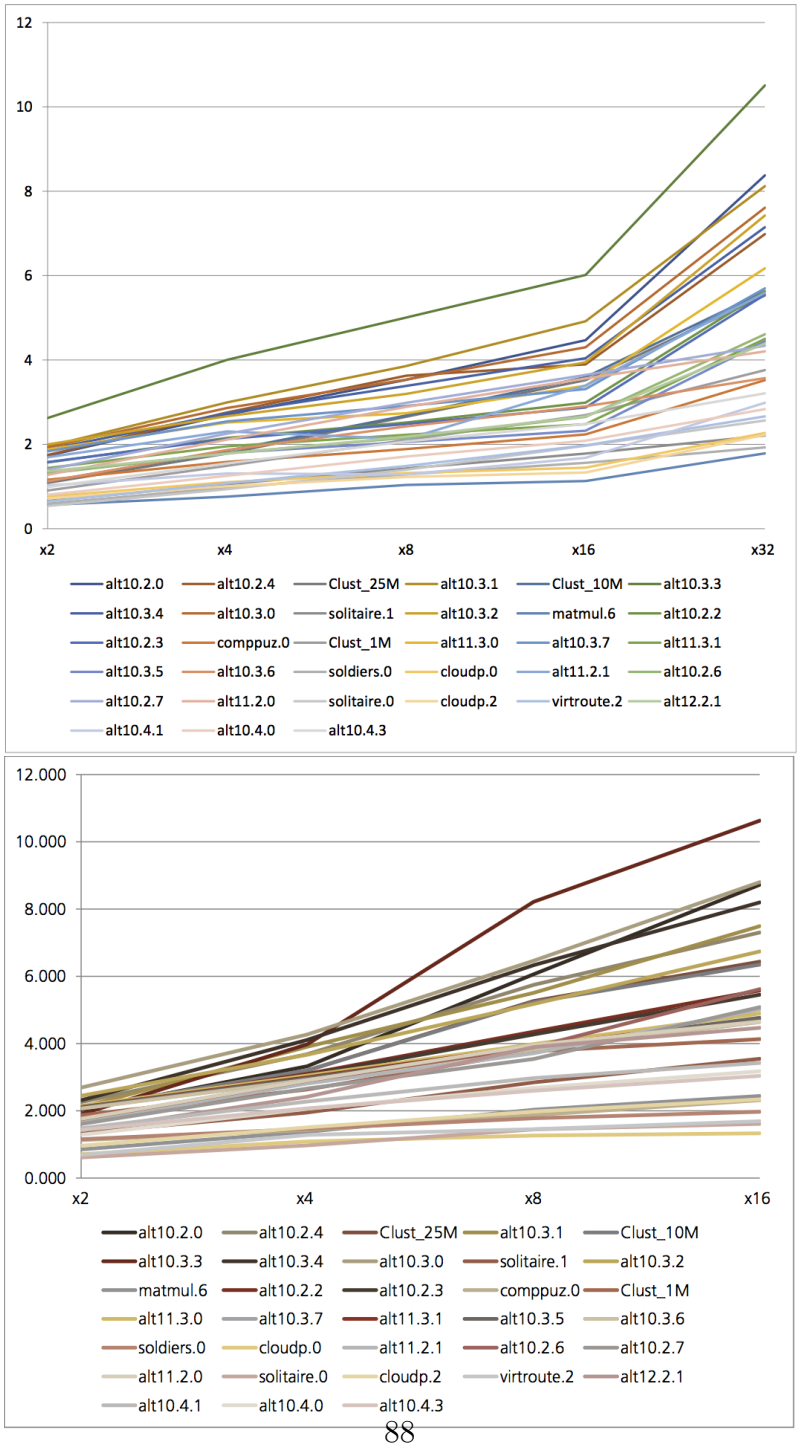


Figure 7.6 summarizes the results by displaying the average speed up across all graphs on the Mac and Linux systems with varying amounts of worker threads. We have a 4.89 times speed up on the Mac and 4.79 speedup on Linux using hardwares' maximum number of worker threads. Overall, these results are encouraging. As [3] notes, concurrent algorithms rarely scale one for one with the number of threads because they are much more complex, use shared data structures, and incur overhead from synchronization and collision resolution. Mathematically, a concurrent algorithm with only a 20% sequential bottleneck cannot have a speed up of more than 5 even with unlimited cores. Further, while runtimes do not decrease one for one with the number of cores, the algorithm seems to scale rather well. We see significant speedups when increasing thread count from 8 to 16 and from 16 to 32. Extrapolating from the results would suggest that we would achieve additional sizable performance gains on hardware with 32 or 64 cores; a difficult feat for a concurrent algorithm. Figure 7.7 illustrates this assertion, performance gains do not seem to be plateauing at 16 or 32 cores.

The algorithm performs better on large graphs with average max-thread speedups of 6.28 and 5.88 on graphs with over a million vertices on Mac and Linux respectively. On graphs with over 5 million vertices, speedups average around 7.5-8.5 in both environments (albeit the very small sample size). One could argue these numbers are more important since we are mainly concerned with efficiency process very large graphs. There are undoubtedly situations where many small graphs must be processed quickly. But in that case the concurrent Tarjan's algorithm is useless anyway; it would be more wise to have each core run the sequential algorithm on a different graph. The superior performance on large graphs is unsurprising; searches will collide less and constant factor overheads are drowned out. Given that the shared hash table accounts for about half the execution time, we believe the results could be significantly improved with a more efficient hash table implementation. Unfortunately, there are not many open source high performing C++ concurrent hash tables to choose from nor did we have time to design a state of the art map. We make some additional observations concerning the results. First, notice the that the speed up from 16 to 32 threads is disproportionately large on Linux while the the speed up from 8 to 16 threads is disproportional small from 8 to 16 threads on Mac OS. This almost certainly results from switching to hyper-threading on the two machines as the physical cores are exhausted. The decreased performance on Mac when switching to hyper threading is unsurprising given that we are no longer increasing the num-

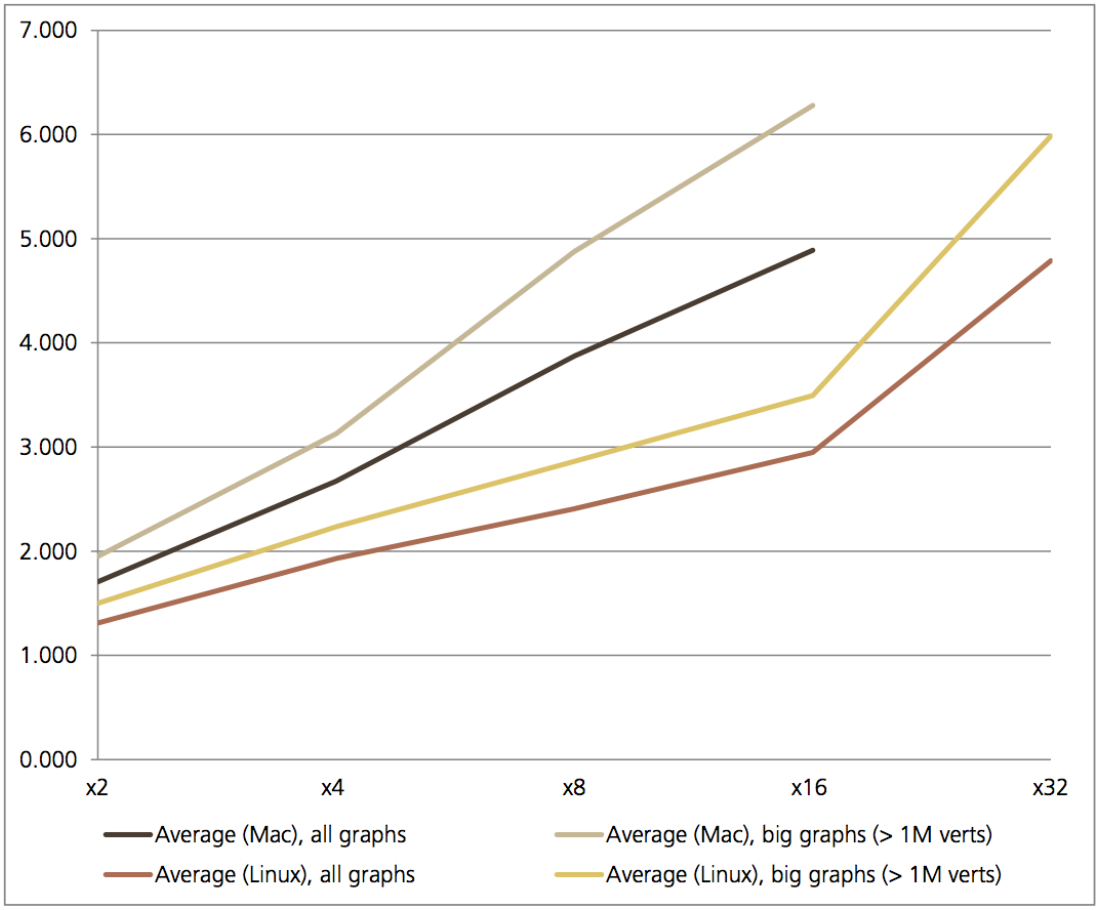
Figure 7.6: Summary of Speedups for Linux and Mac

	Time(N)/Time(1)				
	x2	x4	x8	x16	x32
Average (Mac), all graphs	1.71	2.67	3.87	4.89	
Average (Mac), big graphs (> 1M verts)	1.95	3.13	4.88	6.28	
Average (Linux), all graphs	1.31	1.93	2.41	2.95	4.79
Average (Linux), big graphs (> 1M verts)	1.50	2.23	2.86	3.49	5.99

ber of physical cores. The hyper-threading performance jump on Linux is somewhat unexpected, but probably a result of more favorable scheduling or memory sharing when hyper-threading is enabled.

The algorithm achieves a larger speed up on the Mac despite the machine having half as many cores and less efficient hyper-threading. There are several potential explanations. Most obviously, all the development stages of this software occurred on devices using Mac OS. This gave us more time to catch the bottlenecks specific to Apple hardware/OS and optimize the implementation to perform on that system. Second, the Mac model was released less than a year ago and hence threads and atomic operations may have been implemented more efficiently on the newer hardware. Last, informal testing suggested that threads are scheduled rather differently on the two systems leading to more cycles on Linux. Our implementation performs most poorly on the the non-alt CSP graphs. This is probably because the graphs only have trivial SCCs and we did not really consider those graphs enough when designing the implementation.

Figure 7.7: Average Speedup over Sequential across all graphs



Chapter 8

Conclusion

8.1 Concluding Remarks and Future Work

We built upon the work of [3] to develop a parallelization of Tarjan's algorithm that is fully lock-free until the end of execution when the number of worker threads exceeds the number of available tasks (searches). To our knowledge, no other lock-free parallelization of Tarjan's algorithm has been proposed. Overall, our results are pretty good; we achieve a 4.89 speed-up over the sequential algorithm on our 8-core Mac Pro with a 6.28 speed up on graphs with over a million vertices. Further, the algorithm seems to scale fairly well; we see a similar speed-up each time we increase the number of physical cores. Results are somewhat hurt by a very large bottleneck with the shared memory hash table but this indicates a need to find a better C++ concurrent hash map rather than an underlying issue with the implementation.

We worked toward a lock-free and more efficient implementation by removing the cell level, search level, and `suspensionManager` level mutex locking present in [3]. We avoid cell level locking by using atomic operations and implementing a wait-free `blockedList` data structure to track the searches suspended on a cell. Search level locking is avoided with atomic operations and by structuring the algorithm so that many race conditions on search objects are naturally avoided without synchronization. We eschew `suspensionManager` level locking by implementing wait-free suspension and cycle detection/resolution protocols. To detect cycles, we construct a candidate cycle by following the path of suspended searches as in [3]. Rather than protecting the path's integrity with a mutex; however, we use a method that re-traverses the path to verify its integrity.

it is possible to take a completely different approach with resolving deadlocks caused by suspension cycles. Both this dissertation and [3] have a search check if it created a cycle each time it suspends. An alternative approach would be to have a separate thread investigate suspended searches for deadlocks and resolve them if necessary. This could potentially simplify the suspension mechanism and may require less synchronization since deadlocks do not need to be detected immediately. We leave this investigation for future work.

We developed a lock-free memory recycling system for search and cell objects. Memory recycling reduces the number of cell objects by a factor of 200-600 for small graphs and 5,000-21,000 for larger graphs. The effect is similar for search objects. This reduces the growth of cell/search object use from linear to logarithmic as CSP graphs grow. This can yield performance gains by decreasing the frequency of heap allocations and deallocations during execution and hence avoiding system level overhead and the sequential bottleneck of heap locking. Memory recycling approximately halves the algorithm's memory usage on the graphs we examined. While this constant factor reduction does not alter the algorithm's overall space complexity, it has the practical benefit of keeping a larger percentage of memory in Cache/RAM. The memory recycling technique alone results in a speed up of around 1.4 over the base case of deleting cell and search objects at the end; a solid speedup for a memory management technique. Still, we believe there are other memory management methods that could deliver fruitful returns as well. For instance, allocating cell and search objects in bulk and deleting them at the end would be much simpler and have less overhead while also avoiding frequent heap allocations/deallocations (but this method would not half the memory usage).

Because reference counting each pointer instance can introduce a lot of complexity and overhead, we instead use partial reference counting for cell objects. Namely, the code is structured so that we can make guarantees about when cell objects cannot be recycled by other threads to avoid incrementing the reference count. In other areas, we verify from time to time if the cell has been recycled instead of reference counting the object. A cell's reference count is only ever incremented when it is first assigned a vertex and in `execute()`. Search objects do not need to be reference counted at all.

While our results are solid, we believe that they could be significantly improved with a more efficient shared hash table. This is supported by the fact that the map accounts for about half the execution time. Unfortunately,

none of the concurrent hash tables we discovered online outperformed our in-house `OpenSharded` map. We believe future work should focus on resolving this bottleneck as it accounts for half the execution time. In section 7.2, we mentioned some interesting strategies for overcoming contention in concurrent map objects such as using thread local maps and temporarily backing off locked shards to retrieve other objects. These techniques do not produce significant improvements in our implementation as the hash table's slowness does not appear to originate from contention but from the underlying implementation. We believe these techniques could be useful in similar problems; however, so we mention them in the text. Perhaps a better solution would be to use a separate hash set for completed cells as that would only need to hold vertex identifiers (the hash table can be purged of entries representing completed cells during recycling). This would certainly improve the memory footprint but we leave it to future work to investigate whether it could help resolve the bottleneck.

Finally, we propose a potentially fruitful project for future work. In the related works section, we mentioned that most parallel SCC algorithms perform well on graphs with large SCCs (such as FB) or on graphs with many small SCCs (such as coloring algorithm) and struggle with the other type of graph. The algorithm that this dissertation and [3] worked on has the same issue, performing poorly on graphs where a single SCC contains a significant portion of the graph's vertices (over 20%). State of the art methods such as Multi-Step [11] resolve this issue by first using an algorithm like FB to identify large SCCs and then switching to another algorithm to find the rest of the SCCs. This strategy could be applied with our algorithm to produce a more universal SCC algorithm. For instance, the algorithm could initially run FB to identify the large SCC and then execute our implementation to collect the remaining SCCs. Give that our implementation scales rather well, we believe combining our algorithm with another parallel algorithm that identifies the large SCCs could challenge current state of the art methods especially if a better hash table implementation is found.

References

1. Sedgewick, Robert, and Kevin Wayne. *Algorithms.* 4th ed, 2016. Print.

2. Tarjan R (1972) Depth-first search and linear graph algorithms. SIAM J Comput 1(2):146160.

3. Gavin Lowe, Concurrent depth-first search algorithms based on Tarjan's Algorithm, *International Journal on Software Tools for Technology Transfer (STTT),* v.18 n.2, p.129-147, April 2016

4. Dhingra, Swati, Poorvi S. Dodwad, and Meghna Madan. "Finding Strongly Connected Components in a Social Network Graph." *International Journal of Computer Applications 136.7* (2016): 1-5.

5. Jiri Barnat, Petr Bauch, Lubos Brim, Milan Ceska, Computing Strongly Connected Components in Parallel on CUDA, *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, p.544-555, May 16-20, 2011

6. Harold N. Gabow, Path-based depth-first search for strong and biconnected components, *Information Processing Letters,* v.74 n.3-4, p.107-114, May 2000

7. Reif, J.H., Depth-First search is inherently sequential, *Information Processing Letters* 20(5), 229234 (1985)

8. L. K. Fleischer, B. Hendrickson, and A. Pinar. On Identifying Strongly Connected Components in Parallel. *Parallel and Distributed Processing*, volume 1800 of LNCS, pages 505511. Springer, 2000

9. Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (SC '13). ACM, New York, NY, USA, Article 92, 11 pages.

10. W. McLendon III, B. Hendrickson, S. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901910, 2005.

11. G. M. Slota, S. Rajamanickam, K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems", *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp.* (IPDPS), 2014.

12. S. Orzan. On Distributed Verification and Verified Distribution. PhD thesis, Free University of Amsterdam, 2004.

13. J. Barnat and P. Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of LNCS, pages 316330. Springer, 2006.

14. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Parallel explicit model checking for generalized Bchi automata. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems* (TACAS) 2015

15. L. Hundt, "Loop Recognition in C++/Java/Go/Scala," Proc. Scala Days, 2011.

16. Maurice Herlihy, Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2008

17. Thread Building Blocks Library Concurrency Library, Intel 2017

18. Fan, B., Anderson, D. G., And Kaminsky, M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), NSDI'13.