Implementing Parallel Topological Sort in a Java Graph Library

Jochem Schutte
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.jochem.schutte@gmail.com

ABSTRACT

Graphs are a very commonly used representation of many real-world models, situations and applications. Very large graphs are commonly analysed by parallel algorithms to speed up the calculations. Although any highly used programming language would certainly benefit from a library of these shared memory, parallel graph algorithms, Java still lacks such a library. This study has set out to implement and verify such a library. This study has extended the library implemented by De Heus [4] with the parallel topological sort algorithm and has improved the control sequences used by the library. The result is an extended, improved Java implemented library of parallel graph algorithms.

Keywords

Graph algorithm, parallel, correctness verification, Java, Library

1. INTRODUCTION

Graphs are a commonly used construct in probably every aspect of the scientific field, spanning from electrical circuits and computer networks to maps and dependency models. For almost every model or data representation there exists an abstraction to a suitable graph representation. This abstraction makes it possible to subject many applications to the same set of algorithms. However, although most of these algorithms have acceptable, polynomial-time complexity, these algorithms take a lot of time to run on large graphs.

The solution to this time-consuming problem is parallel computing. Performance improvements have traditionally been gained by decreasing the size of computer chips and increasing the amount of transistors, allowing more calculations to be run in the same amount of time. However the performance gained decreases as the size of chips decreases, while at the same time complications rise. This is why many chip makers turn their attention from increasing the transistor count to developing and improving multi-core chips [7]. The reason why the single-machine, shared-memory approach is chosen, as opposed to a distributed memory approach, is that it does not have the overhead accompanying distributed programming and it also does not require access to a cluster of distributed processing units. Taking the ability to run multiple processes and applying that principle to a parallel, shared-memory algorithm, we are able to diminish the amount of time needed to run an algorithm on a very large graph.

Due to the widespread applications of graphs and the tremendous performance of parallel algorithms, every popular programming language will benefit from a generic library of parallel graph algorithms. According to surveys, Java is one of the most commonly used programming languages [5], due to features such as platform independence, worry-free memory access and its similarity to the C language [13]. However, although Java is one of the most popular programming languages, there is no parallel, shared-memory graph algorithm library for Java available. Although many libraries approach this sort of library, none of them contain the important features: parallel algorithms and a shared memory approach. There are libraries concerning the visualization of graphs and libraries with graph algorithms, but most of these algorithms are sequential [1]. Even the libraries armed with parallel algorithms take a distributed memory approach [10], introducing unnecessary overhead when run on a single machine.

The goal of this study was to implement a library that does contain all of the aforementioned features. The offset of this study has been the research performed by De Heus [4]. She has developed the necessary structure of classes and interfaces as well as a selection of parallel algorithms. The contribution of this study is to prove and improve the control sequences (termination detection) of the parallel algorithms and to extend the library with the topological sort algorithm. The reason why topological sort was chosen is because of its use in applications such as dependency and task rearrangement. The sequence of actions that was taken to accomplish this goal is to (1) demonstrate that the used termination detection algorithm does not operate properly, (2) to adjust the existing algorithm to a valid algorithm and to provide a mathematical proof that this algorithm is correct. (3) The constructed algorithm for parallel topological sort is an adaptation of the sequential algorithm first described by Kahn [9].

This paper is divided into four parts.
Firstly, the research that was done will be illustrated. We will start with a brief description of the background of field of graph libraries (section 2) and the topics and concepts that will used in this paper (section 3). Then the purpose of this paper will be explained and illustrated with research questions (section 4). Finally the methods that were used to answer these questions will be described (section 5).

The second part concerns the termination detection algorithm. In this part the previously implemented termination detection algorithm will be concluded to be insufficient and a new algorithm will be devised and proven to be correct (section 6). In the third part of the paper the implementation of the parallel topological sort algorithm will be discussed (section 7). After the implementation has been discussed, we will list and discuss the results of the speed comparison tests that were run on the sequential and parallel topological sort algorithms (section 8).
The final part will interconnect and finalize the different sections of the paper. The work performed by others and alternatives that were not used will be explained in the related work section (section 9). This will be followed by work that could, and should be, done in the future (section 10). This part, and in fact the entire paper, will end with a concluding chapter, summarizing the paper (section 11).

2. OTHER GRAPH LIBRARIES

To give a picture of what other graph libraries originally exist, some other graph libraries are listed below

2.1 Parallel Boost Graph Library

The Parallel Boost Graph Library (PGBL) is an open source library of parallel, generic graph algorithms. The work on this project was initialized by the Pervasive Technology Labs of the University of Indiana, but it is now licensed under an open source license and being extended by a community of programmers. The library can handle several graph representations and can be mapped as shared-memory or distributed-memory [8]. The library is however programmed in C++ which contradicts the goal of this study.

2.2 JGraphT

JGraphT is an open source Java-based library of graph algorithms. The library supports many commonly used graph representations and uses generics for flexibility. Although powerful, JGraphT uses only sequential algorithms, instead of the intended concurrent algorithms [1].

2.3 HipG

HipG is a Java-based library of parallel graph algorithms. This library comes very close to the intended goal of this study. However the library is not intended for parallel execution on a single machine. Instead it takes a distributed-memory approach. Parallel execution of the algorithms can be simulated by running multiple instances of the program on a single machine, but the program will still run as if it were executed in a distributed memory environment. This creates a lot of unnecessary overhead and it requires the graph to be broken up and divided along the workers, something that is unnecessary when working in a shared-memory environment [10].

2.4 De Heus

This study has extended the research performed by De Heus in 2011. She has developed a library of interfaces to represent any graph representation and has implemented some parallel graph algorithms (reachability and connected components) and a termination detection algorithm, based on research of other graph libraries and the subject of concurrent programming. She intended to implement the topological sort algorithm herself, but was unable to due so due to the limited time for her study. The research performed to write this paper has set out to extend the library with the topological sort algorithm and to improve the aforementioned termination detection and to prove it to be correct [4].

3. BACKGROUND

This paper will utilize concepts that may require an explanation. For the understanding of the reader these concepts are explained below.

3.1 Topological sorting

The topological sorting algorithm sorts every node n in a directed acyclic graph such that all directed edges point in the same direction. If the algorithm is run on a graph that contains cycles then the algorithm will return an error, because then a topological sorting is impossible [3].

An example of the application of such an algorithm is the sorting of a set of ordered tasks. When represented as a directed graph, these tasks can be organized in a way that it is immediately clear which task precedes a certain task t and thus has to be performed before task t can be performed. This example demonstrates the importance of such an algorithm in the planning of many dependent tasks.

3.2 Termination detection

The algorithms of the developed library run in a multi-threaded environment. Such an environment requires a construction that concludes when all tasks have been performed and all processes can terminate. Such a task is performed by a termination detection algorithm. the termination detection algorithm can run in the background, or can be called by a process during the normal execution of the main task.

3.3 LTL

The proof for correct termination detection uses a reasoning method called Linear Temporal Logic (LTL). For the understanding of the reader the concept of LTL is explained briefly.

LTL is built on top of the basic logical reasoning so the basic operators ($\forall, \exists, \wedge, \vee, \Leftrightarrow$) can be used in combination with the LTL operators. LTL has the following operators.

- $\square$: Always. $\square a$ means that proposition a will always hold.
- $\diamond$: Eventually. $\diamond a$ means that eventually proposition a will hold.
- $\bigcirc$: Next. $\bigcirc a$ means that in the next state proposition a will hold.
- $\cup$: Until. $a \cup b$ means that proposition a will hold until proposition b holds.

3.4 Atomic operations

This paper uses the concepts of atomic methods as implemented in the Java language by the java.util.concurrent. atomic package. The following description of this package is extracted from the Sun Oracle website for a brief understanding of the concept of atomic operations.

A small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update operation of the form:

boolean compareAndSet(expected,update);

This method (which varies in argument types across different classes) atomically sets a variable to the updateValue if it currently holds the expectedValue, reporting true on success. The classes in this package also contain methods to get and unconditionally set values, as well as a weaker conditional atomic update operation weakCompareAndSet described below.

[...]

get has the memory effects of reading a volatile variable.

set has the memory effects of writing (assigning) a volatile variable.

[...]

compareAndSet and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables. [11]

4. PURPOSE OF THIS RESEARCH

4.1 Goal

The problem addressed by this research is that there did not exist a complete shared-memory parallel graph algorithm library in Java, despite the many applications of graphs and the fact that Java is one of the most commonly used programming languages. The goal of this study was to extend, prove and improve the library implemented by De Heus in her bachelor thesis paper [4]. More specifically, proving and improving the algorithm used to detect termination and extending the library by implementing the topological sort algorithm. This will work towards a Java library of shared memory, parallel graph algorithms.

To reach this goal, the following research questions were composed.

Are the control sequences of the library correct?
The control sequence this relates to is the already implemented termination detection. The correctness of the termination detection translates to whether the algorithm terminates when the task is completed and does not terminate when the task has not yet been completed. This algorithm is a crucial part of the library and is used by multiple graph algorithms, including the parallel topological sort algorithm. This is why it is important to prove that the termination detection is correct.

Is a parallel version of the topological sort algorithm possible?
This explores the possibility of a parallel topological sort algorithm. The reason why this may not exist is that topological sort requires extra restrictions on the traversal of a directed graph. These restrictions may not be taken into account for parallel depth-first search or parallel breadth-first search.

How do the parallel algorithms perform when compared to their sequential counterparts? This explores the performance (in terms of speed) of the parallel algorithms compared to the sequential version of the algorithm. This will be a comparison of both the real world speed and the theoretical order of complexity.

5. RESEARCH METHOD

To resolve the aforementioned problem and answer the research questions, the following methodology has been set up.

5.1 Correcting termination detection

In this part of the study the previously implemented termination detection was proven to be incorrect. This was done by translating the termination detection algorithm to Promela and using the validation tool Spin [2]. With the use of the Spin tool a specific path of interleaving was followed, demonstrating that the termination detection algorithm can terminate while it should not.

Following the conclusion that the old termination detection algorithm is incorrect, the old algorithm was altered by making the necessary executions more atomically. This new algorithm was then proven to be correct by formulating a formal, mathematical proof for the correctness of the algorithm, answering the first research question.

5.2 Library development

The second part of the study will involve extending the library with the parallel topological sort algorithm and the corresponding graph representation. While the sequential topological sort algorithm is a fairly simple adaptation of the depth-first search (DFS) algorithm, the parallel algorithm involves a little more consideration. This is due to the fact most of the known shared memory, parallel depth-first search algorithms are a mix of depth-first search and breadth-first search. This mix would allow the topological sorting algorithm to insert a parent node into the list before the children of that node, because multiple nodes might be evaluated at once in a breadth-first search fashion. This will result in an incorrect sorting of the nodes. Therefore a different algorithm must be used. The used solution is an adaptation of an early algorithm first described by Kahn [9]. This will work towards the goal of the study: a concurrent graph algorithms library in Java.

5.3 Speed tests

In this final section the developed library will be tested according to the last criteria set in the research questions: speed. The speed will be tested by a combination of complexity reasoning and speed benchmark tests on some randomly generated, large graphs. The complexity reasoning will supply us with the formal speed comparison, while the benchmark tests will provide a real-world runtime comparison.

6. TERMINATION DETECTION

In this section the old termination detection used by De Heus is proven to be incomplete, answering the first research question: Are the control sequences of the library correct? It is important to validate that the termination detection algorithm is correct because we will use it for our implementation of topological sort.

6.1 Disproving old termination detection

The termination detection algorithm devised by De Heus works as follows, as listed in listing 1: every thread has a value in the arrays *load* (line 1) and *waiting* (line 2).

The value in the *load* array is the amount of tasks left at the moment of the last return from the call *process()*, the value in the *waiting* array is used to keep a thread waiting when there is a temporary absence of work (line 12-14) and to release a waiting tread after more work becomes available (lines 15-16). After every call of *process()* (line 6) the thread will loop over all loads of all threads (lines 7-10). If all loads are zero, then the boolean *done* is set to true and the algorithm will terminate.

```
 1 AtomicInteger[numberProcessors] load;
 2 AtomicBoolean[numberProcessors] waiting;
 3 AtomicBoolean done;
 4
 5 while(!done){
 6     load[threadNumber] = process();
 7     for(int i = 1; load[i-1]==0 && i <
 8         numberProcessors; i++){
 9         if (i== (numberProcessors-1) &&
               load[i]==0)
10             done = true;
11     }
12     if(load[threadNumber]==0)
13         waiting[threadNumber]=true;
14     else
15         for(int i = 0; i <
               numberProcessors; i++)
16             waiting[i]=false;
17     while(waiting[threadNumber]&&!done);
18 }
```

Listing 1. Old termination detection algorithm

The termination detection algorithm devised by De Heus [4] has not been validated to be correct, even though correctness is a very important requirement for a concurrent program. The important correctness property of a termination detection algorithm is that the program terminates if and only if there is no more work left to do. For this particular implementation this translates to that the boolean *done* is only set to false when and only when all work is done and every current and future call to *process()* yields zero.

The main concern with this termination detection algorithm is the situation in which a thread *t* is checking for termination (lines 6-11), while another thread, not yet checked, wakes up a thread concluded to be idle (line 16). In that case the thread waking up another thread may afterwards go idle while the recently woken up thread continues to work, making it possible for thread *t* to conclude termination while the graph is still being processed. This may occur because the termination check is not performed atomically and the data in the load array may change during the termination check.

This flaw was exposed by translating the pseudo code algorithm to Promela and carefully manipulating the interleaving of the threads using the tool Spin. The used implementation of the method *process* is a breadth-first exploration of a graph. The used graph is a chain of 10 nodes. The first result of the validation test of the "old" termination detection algorithm was that the algorithm contains at least one flaw. The algorithm has the flaw that when a thread is in the method call of *process()*, then the value of *load[processID]* is still that of the last call of *process* (possibly zero). This causes a thread to evaluate outdated data and leads to a conclusion that a thread is waiting, when it is in fact processing. This can cause problems due to the non-atomic nature of the execution of the method process.

Luckily, the aforementioned problem can easily be overcome by setting the value of *load[processID]* to a non-zero value before calling *process()* or after the wait loop. However the algorithm also contains the flaw that was mentioned earlier in this section. It is indeed possible for a thread that has been evaluated to be waiting to wake up and keep processing, while the other threads are evaluated to be waiting. This can indeed cause the algorithm to conclude termination while in fact the graph has not been fully explored.

The problem with the non-atomic termination check, contrary to the problem concerning outdated information, is far harder to solve. That is why it was decided to implement a new termination detection algorithm.

6.2 New termination detection

In this section the adapted termination detection algorithm is illustrated.

```
 1 AtomicInteger working = 0;
 2 AtomicBoolean done = false;
 3 AtomicBoolean waiting = false;
 4
 5 termdet(){
 6 working.getAndIncrement();
 7 while(!done){
 8     int temp = process();
 9     if(temp == 0){
10         waiting.set(true);
11         working.getAndDecrement();
12         done.CAS(false,working == 0);
13         while(waiting && !done){
14             // spinning
15         }
16         working.getAndIncrement();
17     }else{
18         waiting.set(false);
19     }
20 }
```

Listing 2. New termination detection algorithm

The new termination detection algorithm (listing 2) is very much the same as the old algorithm. In the new version, however, the problem with the non-atomic termination check has been solved by making it atomic. A single atomic integer is used instead of an array with boolean values (line 1). A thread that is waiting is represented by decreasing the value of that integer (line 11). When the value of working is evaluated to be zero (line 12), then all threads have concluded that there are no more tasks to perform, indicating that the calculations have finished. (the next section will elaborate on these implications)

Every thread running the algorithm loops over the following sequence:

A call to (an implementation of) *process*() is performed. This returns an integer representation of the amount of tasks currently available (line 8). If there are still tasks left then the thread will set *waiting* to false, waking up all threads currently waiting (line 18).
When there are currently no tasks available then the following section of the algorithm is executed.

- The boolean *waiting* is set to true to keep threads waiting until more tasks become available. (line 10)
- The value of *working* is decreased by one to indicate that the thread is waiting. (line 11)
- *working* is evaluated. If it is equal to zero, then *done* is atomically set to true, indicating that the program should terminate, else it is left unchanged. (line 12)

- The thread will keep spinning until *done* is true or another thread sets *waiting* to false. (lines 13-15)
- When the thread exits the spinning loop it increments the value of *working* to indicate that the thread is no longer waiting. (line 16)

6.3 Termination detection proof

In this section the newly implemented termination detection will be proven to be correct. The properties to be validated are that the algorithm is deadlock-free and that termination is detected properly. The way this will be done is by first setting up requirements for the implementation of *process()*. Then some observations about the algorithm will be made and explained. Using these requirements and invariants, the deadlock-free property and correct termination detection can be deduced.

6.3.1 Definitions

The following definitions are made to shorten the proofs and make them easier to read.

finished The entire graph has been processed, there are no more tasks to perform. Note that this only marks that the processing has finished, not that the algorithm has concluded that it is finished. This is what will be proven later.

p.hold Process p is on hold, which means that its execution is between lines 12 and 15 (listing 2).

p.spinning process p is spinning, which means that its execution is looping over lines 13-15 (listing 2)

p.lastProcess() The result of the last call to *process()* made by process p.

process() The result of a call to *process()* if it were performed at this state. Which process makes a call is not important for this statement.

P The set of all processes.

W The set of processes which are on hold.

6.3.2 Requirements for process()

The implementation of *process()* needs to satisfy the following requirements in order to guarantee a correct execution of the program.

- *process()* needs to be deadlock free
- New tasks cannot arise without a cause, only during the execution of *process()*
- $finished \Leftrightarrow \Box process() = 0$. When the calculations have finished then a call to *process()* will always yield zero. The inverse means that when a call to *process()* will always yield zero, then the calculations must have finished.

6.3.3 Observations

The following observations can be made. Note that some of the invariants only hold while $\neg done$. When *done* is true then the rest is trivial, because then the program will terminate.

$p.hold \Rightarrow p.lastProcess() = 0$
If a process is on hold, then the result of its last call to *process* was zero. This is due to the structure of the *if*-statement.

$working = |P| - |W|$
The value of the integer *working* is the amount of all processes minus the amount of processes on hold. This is due to the fact that the code segment *hold* is located between decrementing and incrementing the value of *working*

$\forall p \in P : p.lastProcess() = 0 \Rightarrow \Box process() = 0$
When all calls to *process()* yield zero, the calculations are finished and the system should terminate. This means that if every call, from every process, to the method *process()* yields zero then every current and future call will yield zero. This proposition holds because tasks can only arise from a call to *process()*. When all last calls return zero, then there are, and will be, no more tasks.

$$\begin{array}{ll} \Box process() = 0 & \Rightarrow \Diamond \forall p \in P : p.hold \\ \Diamond \forall p \in P : p.hold & \Rightarrow \Diamond P = W \\ \Diamond P = W & \Rightarrow \Diamond working = 0 \end{array}$$

When *process()* always returns zero, then eventually all processes will be on hold, so eventually the set of processes will be equal to the set of processes on hold, so eventually working will be zero. This follows from the algorithm and the definitions. This statement is valid up to and including the moment *done* becomes true.

From the statements above the following statements are deducible (up until and including the moment when *done* becomes true).

$$P = W \Rightarrow \Box P = W$$

$$working = 0 \Rightarrow \Box working = 0$$

$working = 0 \Rightarrow \Diamond done$
When $working = 0$, then $\Box working = 0$. This means that eventually the execution of $working == 0$ yields true and *done* will be set to true. Note that for this to be true, deadlock-freeness must first be proven.

6.3.4 Deadlock-free

In this section it is proven that the algorithm is deadlock-free. It is a requirement for the implementation of *process* that it is free of any deadlocks, so we need to prove that the termination detection algorithm is also deadlock-free. This means that a thread should always eventually exit the wait loop, meaning that always eventually *waiting* should be false or *done* should be true. This statement is proven by induction.

To prove: $\Box\Diamond(\neg waiting \vee done)$
Base step
The base step is trivial because at the start of the while loop (line 8) *waiting* is false, so $\Diamond(\neg waiting \vee done)$ is true.

Induction step.
What we need to prove is that if in this state eventually $\neg waiting \vee done$ is true, then this is also true in the next situation: $\Diamond(\neg waiting \vee done) \Rightarrow \bigcirc\Diamond(\neg waiting \vee done)$
This is trivial for the cases where $\bigcirc(\neg waiting \vee done)$ is true, for then $\bigcirc\Diamond(\neg waiting \vee done) \Rightarrow \bigcirc\Diamond(\neg waiting \vee done)$ is also true. The first interesting case is $(\neg waiting \vee done) \wedge (\bigcirc\neg(\neg waiting \vee done))$. When $\neg(\neg waiting \vee done)$ gets set to false, then waiting.set(true) was executed. This means that there is at least one thread not spinning, for it just executed line 10. This leads to the following scenarios.

Scenario 1: $finished \Rightarrow \Box process() = 0$

$$\begin{aligned}\Box process() = 0 &\Rightarrow \Diamond \forall p \in P : p.hold\\ \Diamond \forall p \in P : p.hold &\Rightarrow \Diamond P = W\\ \Diamond P = W &\Rightarrow \Diamond working = 0\\ \Diamond working = 0 &\Rightarrow \Diamond done\\ \Diamond done &\Rightarrow \Diamond(\neg waiting \vee done)\end{aligned}$$

Note that the implication $working = 0 \Rightarrow \Diamond done$ is only valid when The system is not in a deadlock state.. The reason why this implication is however valid is that there is at least one process that is on hold but not spinning (process p), meaning that the system is not in a deadlock state and that there is at least one process that has to execute $done = working == 0$ (line 12).

Scenario 2: $\neg finished \Rightarrow \neg\Box process() = 0$

$$\neg\Box process() = 0 \Rightarrow \exists q \in P : \Diamond q.lastProcess() \neq 0$$

Then for process q:

$$\begin{aligned}q \in P : \Diamond q.lastProcess() \neq 0 &\Rightarrow \Diamond q.temp \neq 0\\ \Diamond q.temp \neq 0 &\Rightarrow \Diamond q.waiting.set(false)\\ \Diamond q.waiting.set(false) &\Rightarrow \Diamond \neg waiting\\ \Diamond \neg waiting &\Rightarrow \Diamond(\neg waiting \vee done)\end{aligned}$$

If not all current and future calls to *process()* yield zero, then eventually a thread q will yield non-zero, meaning that *q.temp* is non-zero (line 8). This means that thread q will eventually set waiting to false, because it will execute the *else*-case of the *if*-statement (line 18). So eventually $\neg waiting \vee done$ will be true.

What is left to prove is $(\neg(\neg waiting \vee done)) \wedge (\bigcirc\neg(\neg waiting \vee done) \Rightarrow \Diamond(\neg waiting \vee done)$. The statement $\Diamond(\neg waiting \vee done)$ stays true until $\neg waiting \vee done$ becomes true. So:

$$\begin{aligned}&(\Diamond(\neg waiting \vee done)) \wedge (\neg(\neg waiting \vee done)) \wedge\\ &(\bigcirc\neg(\neg waiting \vee done)) \Rightarrow \bigcirc \Diamond(\neg waiting \vee done)\end{aligned}$$

In conclusion, for every combination of the variables in the current and next state, it follows that if $\Diamond(\neg waiting \vee done)$ holds, then it also holds in the next state. When combining this with the base step then this forms the inductive proof for $\Box\Diamond(\neg waiting \vee done)$, proving that the algorithm is deadlock-free

■

6.3.5 Termination detection

What needs to be proven is that if, and only if, the processing has finished, then the system will terminate. This is illustrated by the fact that eventually the value of the boolean *done* will be true. This section will use the definitions, requirements and observations of sections 6.3.1, 6.3.2 and 6.3.3 and will chain them to conclude the required relations.

The relation to be proven is as follows: $(finished \Rightarrow \Diamond done) \wedge (done \Rightarrow finished)$.

To prove: $finished \Rightarrow \Diamond done$

$$\begin{aligned}finished &\Rightarrow \Box process() = 0\\ \Box process() = 0 &\Rightarrow \Diamond \forall p \in P : p.hold\\ \Diamond \forall p \in P : p.hold &\Rightarrow \Diamond P = W\\ \Diamond P = W &\Rightarrow \Diamond working = 0\\ \Diamond working = 0 &\Rightarrow \Diamond done\end{aligned}$$

The second part can be proven by contradiction. Assume that done is true, but the processing has not yet finished:

$$done \wedge \neg finished$$

Then:

$$\begin{aligned}done &\Rightarrow working = 0\\ working = 0 &\Rightarrow P = W\\ P = W &\Rightarrow \forall p \in P : p.hold\\ \forall p \in P : p.hold &\Rightarrow \forall p \in P : p.lastProcess() = 0\\ \forall p \in P : p.lastProcess() = 0 &\Rightarrow \Box process() = 0\end{aligned}$$

And:

$$\neg finished \Rightarrow \neg\Box process() = 0$$

In summation:

$$done \wedge \neg finished \Rightarrow \Box process() = 0 \wedge \neg\Box process() = 0$$

This contradiction proves the relation $done \Rightarrow finished$

■

7. TOPOLOGICAL SORT

Now that the new termination detection algorithm is proven to be correct we can direct our attention to the topological sort algorithm. In this section the implemented parallel topological sort algorithm will be displayed and described.

7.1 The algorithm

The sequential topological sort algorithm is a fairly simple adaptation of a depth-first search traversal of a graph [12] (examined in the related work section). However, the parallel counterpart of the algorithm requires some more thought[8][10]. This is because the traversals of many parallel depth-first search algorithms are not in accordance with the topological order of graphs. This is why another approach was taken.

The chosen sequential algorithm was first described by Kahn[9] (listing 3).

```
 1 kahnTopSort(Graph g){
 2     Stack result = new Stack();
 3     Stack queue = new Stack();
 4     for(v in g.vertices){
 5         if(v has no incoming edges){
 6             queue.push(queue);
 7         }
 8     }
 9     while(!queue.empty){
10         Vertex v = queue.pop();
11         result.push(v);
12         for(e in v.outgoingEdges){
13             g.removeEdge(e);
14             Vertex endV = e.endVertex;
15             if(v has no incoming edges){
16                 queue.push(endV);
17             }
18         }
19     }
20     if(g has edges){
21        return error;
22     }
23     return result;
24 }
```

Listing 3. Sequential topological sort by Kahn

The algorithm works by adding every vertex that has no incoming edges to a queue (lines 4-8). Then for every vertex (line 10) in the queue the following code is executed.

Firstly, the vertex is added to the resulting stack (line 11). Then every edge from *v* to *vEnd* is removed (line 13) and if *vEnd* has no more incoming edges it is added to the queue (lines 15-17). This process is repeated until the queue is empty (line 9). This means that every vertex that is not part of a cycle has been processed. Then a final check is performed to conclude that every vertex and edge has been visited and removed. When this is not the case, then the graph was not a directed, acyclic graph and *null* is returned (line 21). Otherwise the algorithm will return a topological ordering of the graph (line 23). An alternative algorithm will be discussed in the related work section.

This algorithm can easily be transformed to a parallel algorithm by placing the code in *while(!queue.empty)* in the method *process* of the termination detection algorithm. This does however pose a problem when two vertices, both having an edge to the same node, execute the code *result.push(endV)* (line 16) simultaneously. This causes the vertex to be pushed to the queue multiple times. For this the construct listed in listing 4 has been devised.

```
//AtomicBoolean[] visited;
boolean vis = visited[endV.id].getAndSet(
    true);
if(!vis){
    queue.push(endV)
}
```

Listing 4. Adaptation for topological sort

With this construct multiple threads executing the code *visited[endV.id]getAndSet(true)* will only yield false once and only one thread will push *endV* to the queue.

The resulting algorithm is listed in listing 5.

```
 1 Graph g;
 2 Stack result = new Stack();
 3 Stack queue = new Stack();
 4 AtomicBoolean[] visited;
 5
 6 void init(Graph graph){
 7     g = graph;
 8     for(v in g.vertices){
 9         if(v has no incoming edges){
10             visited[v.id].set(true)
11             queue.push(v);
12         }
13     }
14     for(t in threadPool){
15         t.start();
16     }
17     for(t in threadPool){
18         t.join();
19     }
20     for(b in visited){
21         if(!b.get()){
22             //a vertex is not visited so
                   not a DAG
23             result = null;
24         }
25     }
26 }
27
28 void doTopSortProcess(){
29     Vertex v = queue.pop();
30     int result = 0;
31     if(v != null){
32         result.push(v);
33         for(edge in v.outgoingEdges){
34             graph.removeEdge(edge);
35             Vertex endv = edge.endVertex
36             if(endv has other no other
                   incoming edges){
37                 boolean vis = visited[
                       endv.id].getAndSet(
                       true);
38                 if(!vis){
39                     queue.push(endV)
40                 }
41             }
42         }
43         result = queue.size();
44     }
45     return result;
46 }
```

Listing 5. Parallel topological sort

This algorithm works very much the same as the sequential version. However, now all threads operate on the shared *queue* and *result* stacks and the aforementioned control sequence is inserted to ensure that a vertex is not added multiple times. The code *t.start()* (line 15) will result in that thread calling the termination detection algorithms, which will repeatedly call *doTopSortProcess()* (line 28). After the processing has finished the threads will join (lines 16-18) and a check if the graph was acyclic (lines 20-24)

7.2 Correctness

The important properties of a parallel topological sort algorithm are that a vertex is added to the list once (and only once) and that the resulting list retains the topological order of the graph. The first property is guaranteed by the earlier discussed modification, displayed in listing 4. The second property also holds. This is because a vertex cannot be added to the list as long as it has incoming edges and the algorithm ensures that a vertex is added to the list before its edges are removed and the connected vertices can be inserted into the list (listing 5, lines 32, 33-42).

This implementation also satisfies the requirements for an implementation of *process()*, set in section 7.2. A vertex has only a limited amount of outgoing edges, so the *for*-loop (line 33) will eventually break. The algorithm has no infinite loops and a thread running the algorithm is never paused, which means that the algorithm is deadlock free. The other two requirements are also met. Tasks are represented by a vertex in the queue, which can only be added by the execution of *doTopSortProcess()*. This means that when the entire graph has been processed (or at least the vertices that are not part of a cycle), then every call to *doTopSortProcess()* will return the value zero, because then the queue is empty and queue.size() will return zero (line 43). The inverse of this rule is also true. New vertices can only be added by evaluating a vertex, thus when the queue is empty it will stay empty, indicating that the calculation has finished.

7.3 Graph structure

For this algorithm to work however, certain information should be retrievable from a graph, vertex or edge. For this the following structure of interfaces has been created. The structure exists of four graph interfaces.

- **Graph** contains the most basic graph functions, for example: getting the vertices and edges of a graph, retrieving the vertices of an edge and getting all edges of a vertex. This interface has earlier been devised by De Heus [4].
- **DirectedGraph** contains basically the same functions as *Graph*, but extended with the functionality of a directed graph and retrieving vertices and edges

in a certain direction. This interface has been devised by De Heus [4].

- **VertexIdGraph** contains the functionality to assign every vertex a unique identifier and retrieving the identifier of a vertex or the vertex with a certain identifier. This is a newly created interface.
- **TopologicalSortGraph** is an interface containing methods that are needed for the implemented topological sort algorithms. Mainly, removing an edge from a graph and telling whether a vertex has incoming edges.

Figure 1 provides an outline of how these interfaces extend each other.

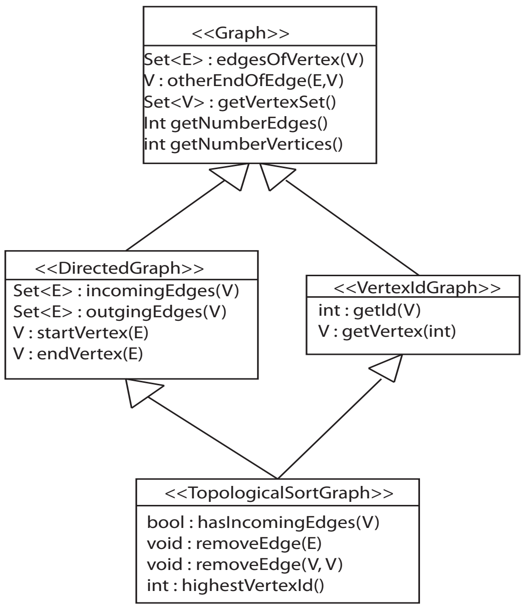

Figure 1. Interface structure

8. SPEED TEST

In this section the results of the speed tests of the sequential and parallel topological sort algorithms will be presented.

8.1 Setup

The speed tests were run by submitting the sequential and parallel algorithms to some (pseudo-)randomly generated graphs. The graphs were generated by repeatedly adding a vertex to the graph and, with certain possibility, adding an edge from every other vertex to that vertex. Whether an edge was actually added to the graph was decided by comparing a randomly generated value to a specified possibility threshold. The generated graphs each had a size of 500, 1000, 2000, 5000 or 10000 vertices and had a probability threshold. The probabilities that were used where 0.001, 0.005 and 0.01. The algorithms were run five times on each of the graphs and the average time, measured using the Java *System.nanotime()* method, was calculated.

The machine used for the tests had the following specifications:

- Processor: Intel Core i7-3770 CPU at 3.40GHz (4 cores)
- Memory: 4x 8192 MB DDR3 at 1333 MHz
- Chipset: MSI Z77A-GD65

8.2 Results

Table 1 displays the average run times for the algorithms to run on different graphs. The numbers on the top represent the complexity of the graph represented as the likelihood of an edge being added to the graph upon creation, the numbers on the left are size of the graphs expressed in the amount of vertices in the graph. For every complexity and size there are three values: The time (in milliseconds) it took the sequential DFS algorithm, the time it took the parallel algorithm with one thread and the time it took the parallel algorithm with four threads to solve the graph.

size/complex		0,001	0,005	0,01
500	DFS	2968	1225	803
	1 core	2907	3499	14988
	4 cores	102303	10177	101361
1000	DFS	665	401	980
	1 core	1032	1700	2427
	4 cores	101520	101914	102474
2000	DFS	580	2291	6117
	1 core	2121	5543	10072
	4 cores	830775	104574	107741
5000	DFS	13215	12252	23562
	1 core	17798	33051	74445
	4 cores	110676	120437	132351
10000	DFS	14660	48473	105947
	1 core	25591	129034	263600
	4 cores	114401	148720	205740

Table 1. Results of speed tests

8.3 Discussion

The complexity of the topological sort algorithm devised by Kahn, which was used for the parallel implementation has complexity $O(|V| + |E|)$. This complexity is the same as the complexity of the depth-first search topological sort algorithm. When the parallel part of the parallel algorithm is run in parallel it should have an approximate complexity of $O(\frac{|V|+|E|}{n})$, where n stands for the number of processors, a complexity lower than that of the DFS topological sort algorithm.

However the results of the speed tests contradict this premise. The runtime of every sequential DFS run on a graph was faster than the parallel version with four threads on four cores. This can be caused by a combination of factors:

Overhead. The termination detection used to control the parallel algorithm may introduce overhead that can cause the algorithm to run slower. The introduced overhead may be investigated by comparing the run times of the parallel algorithm, using one thread, with the pure, sequential implementation of Kahn's algorithm.

Different algorithms. The parallel algorithm is not an adaptation of the sequential DFS algorithm. It is a possibility that the algorithm devised by Kahn naturally runs slower than the DFS topological sort. This difference may be explored by comparing the run times of the sequential implementation of Khan's algorithm with the run times of the depth-first search topological sort algorithm.

Shared resources. The parallel algorithm uses a shared queue and result stack. These structures form critical points. These critical points can cause a serious impact on the performance when accessed by multiple threads at the same time. This may be solved by devising a solution that has separate stacks for every thread.

Low task parallelism. When new tasks (in this case vertices) become available at a slow rate, then most of the threads will mostly be waiting. This diminishes the capacity to run in parallel and impacts the performance. This may be addressed by further parallelizing sub-tasks performed within the *doTopSortProcess()* method.

The performed tests conclude that the sequential algorithm is the superior algorithm for all the tested graphs. However they do show that as the graph becomes bigger, the proportional difference becomes smaller, as displayed in table 2. The displayed ratios are calculated by dividing the runtime of the parallel algorithm executed with four threads by runtime of the sequential algorithm.

size/complex	0,001	0,005	0,01
500	34,6	157,1	126,7
1000	153,8	254,8	104,6
2000	1432,4	45,7	17,6
5000	8,4	9,8	5,6
10000	7,8	3,0	1,9

Table 2. Speed ratios (4 cores/DFS)

With the exception of graph of size 1000 with complexity of 0,001 and 0,005, all proportional speed differences become smaller as the graph becomes bigger or more complex. This leads to believe that for a large enough graph, the parallel algorithm may be faster then the sequential one.

9. RELATED WORK

9.1 DFS topological sort

This algorithm is used as the sequential implementation of the topological sort algorithm, so that it might be compared in terms of speed. For understanding of the reader, the DFS topological sort algorithm is described below.

The sequential topological sort algorithm is a fairly plain adaptation of the depth-first search traversal of a graph. The nodes are visited via a post-order traversal. This algorithm was first described by Tarjan [12]. The pseudocode algorithm of the implemented sequential topological sort algorithm is listed below in listing 6.

```
 1 bool nDAG = false;
 2 bool[] tempmark = bool[nrVertices];
 3 bool[] permmark = bool[nrVertices];
 4 Stack result = new Stack();
 5
 6 Stack topSort(Graph g){
 7     for(v in g.vertices){
 8         if(!permmark[v.id]) {
 9             processVertex(v);
10         }
11     }
12     if(nDAG){
13         return null;
14     else
15         return result;
16 }
17
18 void processVertex(Vertex v){
19     nDAG = tempmark[v.id];
20     if(!nDAG && !permmark[v.id]){
21         tempmark[v.id] = true;
22         for(e in v.outgoingEdges){
23             processVertex(e.endVertex);
24         }
25         result.push(v);
26         tempmark[v.id]=false;
27         permmark[v.id]=true;
28     }
29 }
```

Listing 6. Depth-first-search topological sort

The main method of the algorithm is the *topSort*(*Graph g*) method. It takes the graph to be sorted as an argument and the result is a stack containing the sorted graph or null when the graph is not a directed, acyclic graph. The processing algorithm is the *processVertex(Vertex v)* method. It works as follows.

- It takes a vertex as argument and recursively calls the processVertex method on its child vertices (lines 22-24), before adding *v* to the resulting stack (line 25). This results in a post-order ordering of the tree with *v* as root.
- Along the way the algorithm sets the values of *tempmark* and *permmark*, to mark a vertex as temporarily or permanently visited (lines 21-27).
- The *tempmark* array is used to detect a cycle in the graph (line 20), detecting that the graph is not a directed, acyclic graph and sets the value of *nDAG* to true.
- The *permmark* array is used to detect that the calculations on a vertex and its subtrees has already been completed, so the algorithm can skip that node.

When the processing has finished, the thread in the main method checks if the *nDAG* flag has been set. When it has not, it returns a stack containing the topological ordering of the graph. When it does it returns *null*. [3]

9.2 M.C. Er

An alternative algorithm for topological sort was devised by M.C. Er [6]. The algorithm relies on value iteration to deduce a correct topological ordering of a graph. The algorithm starts at the nodes that have no incoming edges and then follows the edges, assigning every encountered vertex an integer value. With every vertex the integer value assigned is the value of the vertex visited before incremented by one. When a thread encounters a vertex that has already been assigned a higher value than the thread wants to assign to it, the value of the vertex is left unaltered. However, when a thread wants to assign a value higher then the current value of a vertex, the value is updated and the thread will continue along the paths originating in that node, updating the values of the nodes accordingly.

Er argues that the time complexity of this algorithm is the maximum number of edges between any of the start and end nodes. This however requires the amount of processors to be equal to the maximum amount of nodes being processed in parallel. The problem is that this will probably not be the case with large graphs. When the amount of processors is not large enough, then the time complexity is $O(|V|^2)$. This is due to the fact that it may be necessary for a path to be evaluated multiple times. This is the reason why this algorithm was not chosen for the implementation of the parallel topological sort algorithm.

10. FUTURE WORK

Here some topics will be discussed to suggest contributions that can be made in the future.

10.1 Improving the parallel topological sort

The parallel topological sort algorithm has been concluded to run very slowly compared to the sequential DFS topological sort. Research could be performed to examine the weak points of the algorithm and improve it. As mentioned before in section 8.3 the algorithms may have some weak points regarding the speed when run. The algorithm may contain code segments that do not run in parallel or have multiple threads that depend on a single critical point or resource. Measures might be taken to reduce the impact these factors have on the runtime efficiency of the algorithms.

10.2 Extending the library

This paper has extended and improved the existing library. However, still many graph algorithms, both simple and more complex algorithms, need an implementation in the library. Well known examples of these algorithms are:

- Prim's minimum spanning tree
- Dijkstra's shortest paths
- Graph colouring

The computer science community may benefit from such parallel algorithms implemented in Java.

11. CONCLUSIONS

In this section the advancement to reaching the final goal of this research area will be discussed. This will be done by answering the research questions this research set out out to answer.

Are the control sequences of the library correct?
During this research the termination detection was discovered to be deficient. The algorithm did in fact permit the program to terminate when it should not. However, a new termination detection algorithm was devised. This algorithm used the concepts of the previous termination detection algorithm, but has improved it by making the necessary code segments more atomically. These adjustments resulted in a termination detection algorithm that does correctly detect termination. This was proven by formulating a formal proof, demonstrating the correct procedures of the algorithm. In conclusion: yes, the control sequences of the library are now correct.

Is a parallel version of the topological sort algorithm possible?
The primary solution for the topological sort problem is the depth-first search algorithm, resulting in a correct, post-order traversal of the graph. However most parallel depth-first search algorithms do not guarantee a correct, post-order traversal. Therefore a different algorithm was used. This algorithm operates by placing a vertex in the list when it has no more incoming edges. This algorithm needed a few adjustments to make the necessary steps execute atomically and to ensure that a vertex was not inserted multiple times, but did eventually result in a correct parallel algorithm for topological sorting. In conclusion: yes, a parallel version of the topological sort algorithm is in fact possible.

How do the parallel algorithms perform when compared to their sequential counterparts?
The parallel algorithm is correct and should theoretically be faster then the sequential version, because the complexity of of the parallel algorithm is lower ($O(\frac{|V|+|E|}{n})$) than that of the sequential algorithm ($O(|V|+|E|)$). However the parallel algorithm was not able to exceed the execution time of the sequential depth-first search topological sort algorithm. There were too many factors slowing down the algorithm in order to process a graph in reasonable time. Therefore more research should be performed to improve the algorithm or to investigate if there are conditions in which the parallel algorithm does perform better then the sequential algorithm. In conclusion: the implementation of the parallel topological sort algorithm is not efficient enough to exceed the sequential depth-first search algorithm.

12. REFERENCES

[1] Barak Naveh and Contributors. JGraphT. `http://jgrapht.org/`. 2012.

[2] Bell Labs. On-the-fly, LTL model checking with Spin. `http://spinroot.com/spin/whatispin.html`. 2012.

[3] Charles E. Leiserson & Thomas H. Cormen. *Introduction to Algorithms*, page 612. MIT Press, third edition, 2009.

[4] de Heus, Marije. Towards a Library of Parallel Graph Algorithms in Java. In *14th Twente Student conference on IT January 21st*, 2011.

[5] DedaSys. Programming Language Popularity. `http://www.langpop.com/`. 2011.

[6] Er, M.C. A parallel computation approach to topological sorting. *The Computer Journal*, 26(4):293–295, 1983.

[7] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[8] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[9] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[10] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri Bal. HipG: Parallel processing of large-scale graphs. *ACM SIGOPS Operating Systems Review*, 45(2):3–13, 2011.

[11] Oracle and/or its affiliates. Package java.util.concurrent.atomic. `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html`. 2011.

[12] Tarjan, Robert Endre. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[13] Paul Tyma. Why are we using Java again? *Communications of the ACM*, 41(6):38–42, 1998.