Towards a Library of Parallel Graph Algorithms in Java

Marije de Heus
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.deheus@student.utwente.nl

ABSTRACT

Despite the wide availability of multi-core processors and the popularity of Java, there is currently no library for parallel graph algorithms in Java available. Such a library would enable all Java programmers, especially those who work on the verification of programs and model checking to efficiently use graph algorithms. To find out what would be a good design for this library, we have made a start on it. We have created a general design for algorithms and graphs in the library. We have also implemented the reachability and connected components algorithms, both sequential and parallel. In this paper we describe the algorithms we have implemented and our design of the library, focusing on the design choices we made.

Keywords

Concurrency, graph algorithms, Java, termination detection.

1. INTRODUCTION

Graphs have a widespread use. Computer networks, chemistry, biology and physics all benefit from these models of pairwise relations between objects. The problems that arise from the use of graphs are often time consuming because of the complexity of most graph algorithms and the great size of the graphs they work on. Luckily, the increasing number of multi-core processors enables algorithms to execute quicker. However, until now no library for parallel graph algorithms exists in Java. We have made a start on a generic library for parallel graph algorithms.

The 6 sections of this paper are divided as following: this is the introduction which provides a background for our research; section 2 describes our research approach; section 3 is about the graph libraries that are currently available, these are either not in Java or sequential; in section 4 we descibe the design of our library; section 5 is about our implementations of the reachability and connected components algorithms, and section 6 contains our conclusion and suggestions for future work.

In this section we will introduce you the concepts of graphs, describe some frequently used graph algorithms and parallel computing. The reader who is already familiar with those concepts may want to skip the introduction and continue with section 2.

1.1 Graphs

Graphs consist of a number of nodes (vertices) and edges between pairs of nodes. The edges may be given a value or they may all be uniform. Directed edges have a start and end vertex and the relationship between them is only true in that direction. Undirected edges do not distinguish between a start and end vertex and the relationship is true in both directions. If the graph consists of directed edges, it is a directed graph and vice versa. An edge may start and end at the same node, which is called a loop [5].

Graphs may be used to represent anything. The nodes represent objects and edges represent relations between objects. Some examples of possible applications of graphs are:

Social networks Graphs can be used to represent networks of people. Nodes would be different people, and edges could be relationships between the people. This could for example be used in a family tree or for a social network like LinkedIn or Facebook.

Computer networks Nodes could be routers or computers and edges could be connections between the nodes. The edges could be given a weight to represent the distance between the nodes or the time needed for a bit to travel between the nodes.

Maps Here nodes could be cities and edges could represent roads between the cities. The edges could be given a weight to represent the distance between the cities.

Also, graphs are of great importance in the field of formal methods, for example in model checking and the verification of programs [1] [6].

1.2 Graph algorithms

There exist numerous useful algorithms for graphs. There are algorithms available for searching a graph (depth first search, breadth first search, uniform cost search), sorting a graph (topological sorting, pancake sorting), coloring graphs, routing, et cetera.

In the rest of this section the terms *sequential* and *parallel* will be used a lot. Sequential means the algorithm is executed on one processor, one instruction at a time. Parallel means the instructions will be distributed among several processors. Most graph algorithms currently available are sequential, and even if they are parallel, they are in other languages than Java, like C++. Below we will look deeper into some of the most used algorithms.

1.2.1 Reachability
Reachability means being able to get from one vertex in a graph to another one by following an edge. This is a simple type of algorithm but still often used in practice. Reachability can be implemented using depth first search (DFS) or breadth first search (BFS). Our implementation of reachability uses a type of DFS.

1.2.2 Depth first search
This type of search is done by starting at the root of the graph, explore its first child, again explore its first child, until a goal node is hit or there are no more children to explore. Then, backtracking is used, returning to the last node that is not yet completely explored.

It is not certain if depth first search is suitable for use on multi-core processors. Some say that depth first search is inherently sequential [25], while others have provided implementations of parallel DFS that are more efficient than their sequential variants [24, 19].

1.2.3 Breadth first search
Breadth first search also starts at the root node, but then it first explores all of its children, then all of their children, et cetera. This type of search could benefit more from parallel computers. The exploration of the children could be done by different processors. After each level the new children will be divided among the processors again, so the work can always be divided equally.

There is a lot of information available on BFS. Most algorithms that use BFS are implementations of Dijkstra's shortest path algorithm [29, 11, 9, 17] but BFS has many more purposes. We have not implemented BFS but this would probably be a good next step.

1.2.4 Strongly connected components
A part of a graph is called strongly connected if there is a path from each of the nodes to all other nodes in that subgraph. If the entire graph has this property, it is a strongly connected graph.

There are a few different implementations of the (sequential) strongly connected components (SCC) algorithms. Kosajuru's algorithm, Tarjan's algorithm and Garbow's algorithm all compute the SCC's but the latter two only require one DFS whereas the first requires two. There has been a lot of research on how to do this in parallel [21, 16, 12]. We have implemented connected components, which is a type of strongly connected components that only works on undirected graphs.

1.2.5 Topological sorting
Topological sorting is a linear ordering of the nodes of a directed acyclic graph. Each node has to go before all nodes to which it has outbound edges. For example, when getting dressed, one would have to put on socks before putting on shoes, and not the other way around.

This algorithm usually has a good complexity: its running time is linear in the number of nodes + the number of edges. Kahn first described such an algorithm in 1962 [18]. Most other graph algorithms (at least BFS and DFS) usually have a polynomial complexity. We did not have enough time to implement this algorithm, but it would be nice if this was added to the library later on.

1.3 Parallel computing
Parallel computing is the simultaneous execution of the same task on multiple cores, processors or computers in order to obtain results faster. There are several types of parallel computing, like distributed-memory computers, partitioned global address-space computers and shared-memory computers. Java includes APIs for multithreading which are useful for shared-memory computers with multi-core processors. We have designed our library to suit multi-core processors.

2. PROBLEM STATEMENT
There is no (generic) library for parallel graph algorithms in Java available, even though Java is one of the most popular programming languages. According to several measures used by Langpop, Java is in the top 5 of most used programming languages in June 2010 [10] . In the past, when Java developers want to use a parallel graph algorithm they need to use an algorithm in another language, or design one themselves. Both of these options require a lot of work. We have designed a generic library for parallel graph algorithms in Java that will solve this problem.

2.1 Goal
Our goal was to enable Java programmers to use graph algorithms that benefit from the multi-core processors without having to reimplement any of the parallel algorithms or changing existing core data structures. The algorithms are all generic with the vertices and edges as parameters in order to let everyone use their own data structures. The library was written in Java for Java programs to use it.

2.2 Research question
Our main research question is:

Does a generic library of parallel graph algorithms in Java have practical merit?

The library we have made may have a merit if its speed is better than sequential algorithms or algorithms in other languages. However, due to the time constraints we have not been able to benchmark our library, so we cannot compare it to algorithms in other languages. We can still argue if our library has practical merit based on some beneficial characterisitics of our library and Java, and we will do so in our conclusion and discussion. In order to answer this question there is a subquestion that also needs to be answered:

1. *What is a clean design of a parallel graph library in Java?*

We expect that some algorithms will be easier to implement in parallel than others. It is difficult for some algorithms to divide the work equally over several processors. Algorithms like DFS are known to have such problems [25] and will therefore be difficult to implement. Reachability is a simple algorithm and we expect it will be easiest to parallelize. Therefore, we have chosen to implement reachability. Connected components requires only a little more work than reachability because it can take advantage of reachability's implementation and that is the reason why we have also implemented that algorithm.

2.3 Research approach
To answer the research question, we took the following steps:

2.3.1 Literature study
An extensive literature study was done on the subject of parallel graph algorithms to get some ideas of what properties our library should have. Since there are no such libraries available in Java, we have studied the algorithms

that are available in other languages. We also studied parallelism and threads in Java. Most of the results of our studies can be found in the introduction and related work sections.

2.3.2 Implementing part of the library

We selected two algorithms to implement: reachability and connected components. We have created the design for a parallel graph algorithm library and implemented sequential and parallel versions of both algorithms. By doing so, we got first hand experience of the advantages and disadvantages of implementing the algorithms using the design we made. Also, our research question is about the *practical* merit of the library and we would not have been able to answer it without using it ourselves. We did not just implement the algorithms but also created a design for other algorithms to use so other people will be able to extend this library.

2.4 Evaluation

We have written Reachability and Connected Components in Java, both sequential and parallel. In this paper we will shown the design of the library and we will evaluate the design choices we made. We cannot give a yes or no answer to our main research question, but we can name the advantages and disadvantages. We do this in our discussion, which can be found in section 6.1.

3. EXISTING GRAPH LIBRARIES

While no other parallel graph library in Java exist, there are sequential graph libraries for Java (JgraphT, yFiles), graph libraries for other languages like C++ and Objective Caml (STAPL and Ocamlgraph) and perhaps most interestingly, an open source generic parallel graph library in C++ (Parallel Boost Graph Library). Here we describe these libraries, as they might be useful when we need to make design decisions for our library. Perhaps we could even use an available library in Java like JGraphT to build our own library on.

3.1 Parallel Boost Graph Library

The Parallel Boost Graph Library (PBGL) is developed by a part of the Pervasive Technologies Institute at Indiana University. There are numerous graph algorithms available as well as some data structures for graphs, but since it is a generic library one could use their own data structures [15]. The library is built on a sequential library and they have explained how to build a parallel library on a sequential one in [14].

3.2 JGraphT

JGraphT is a free Java graph library that contains several algorithms and types of graphs. It uses Java generics to enable its users to use their own data structures. They implemented more than 15 algorithms, but all of these are sequential.

One of the choices we had to make is whether we wanted to build our library on an existing library like JGraphT or start from scratch. We have started a completely new library for two reasons. Firstly, it is a lot of work to build on an existing library. Even though it seems like it should make things easier, we expect that we would spend a lot of time on studying the library. And perhaps more importantly, we wanted to make some conscious, well-thought design choices. If we would have used someone else's library we could not have made those choices ourselves.

3.3 Other related (graph) libraries

An overview of the related (graph) libraries is given in Table 1. In this table, there are columns specifying the language, whether it is parallel or sequential, generic or not, and if it contains data structures and graph algorithms. Also, we mention if they are available or not; if the table says it is not available it means we could not download the library at all. Our own library has the following properties: it is written in Java, the algorithms are available parallel and sequential form, the algorithms are generic, and there should be many graph algorithms and some data structures. Furthermore, it should be easy for the user to implement their own algorithms and data structures in our library.

4. DESIGN OF THE LIBRARY

In our research we made a start on a library in Java with parallel graph algorithms. Here we will describe the design of the library and the design choices we made. The (documented) code is available as well.

The two packages that are most relevant to describe are the Graphstructures (section 4.2) and the Algorithms (section 4.1). The Graphstructures package contains mostly interfaces for different types of graphs and one implementation called `SimpleGraph`. Algorithms contains interfaces, abstract classes and default implementations of sequential and parallel reachability. It should be noted that every class mentioned in this section is generic with parameters for the vertices and edges of the graph, and therefore users can easily use this library with their own graphs. The parameters are left out of the class diagrams to make them more readable.

4.1 Design for algorithms

The algorithms package is designed to enable others to extend this library with their own algorithms. We have implemented the reachability and connected components algorithm. A class diagram for the algorithms package can be found in Figure 1. In the class diagrams, the interfaces are represented as $<< Interfacename >>$ and the abstract classes all start with the word *Abstract*.

The interface `Algorithm` contains the methods that should be implemented by any algorithm. An algorithm knows on which graph it works, which vertex its start vertex is and it can change the start vertex. Also, there are the methods `preprocessVertex()` and `postprocessVertex`, which should be called when a vertex is entered respectively has been finished. What happens here will likely vary among the algorithms but these methods are often needed.

`AbstractAlgorithm` implements the methods that are defined by `Algorithm`. The reason why we did not leave out the interface `Algorithm`, is that we want to allow the user to implement their own version of `Algorithm` as well.

The subclasses of `AbstractAlgorithm` are `AbstractReachability` and `AbstractConnectedComponents`. `Reachability` contains the method `defaultReachability()` which is already implemented. This forms the basis of the reachability algorithm. It takes a vertex, calls `preprocessVertex()`, gets the vertices that are reachable from here, calls the `processReachableVertices()` with this set of vertices, calls the `postprocessVertex()` method. For a more indepth explanation of the algorithms we refer to section 5. The important thing here is that only the global structure of the algorithm is defined. One can implement this class and define all of the methods that are called to make it suit any purpose.

Table 1. Related libraries

Name	Language	Parallel/Sequential	Generic	Data structures	Algorithms	Available
yFiles for Java [4]	Java	Sequential	No	Yes	Yes	Commercial
JGraphT [23]	Java	Sequential	Yes	Yes	Yes	Open source
JDSL [28]	Java	Sequential	Yes	Yes	Yes	Open source
PBGL	C++	Parallel	Yes	Yes	Yes	Yes
STAPL [3, 2]	C++	Parallel	Yes	Yes	No	No
GTL [13]	C++	Sequential	No	Yes	Yes	No
LEDA [22]	C++	Sequential	No	Yes	Yes	Open source
BGL [27]	C++	Sequential	Yes	Yes	Yes	Open source
Ocamlgraph [7, 8]	Ocaml	Sequential	Yes	Yes	Yes	Yes
MLRisc [20]	SML	Sequential	No	Yes	No	Open source

`AbstractConnectedComponents`, the other class that implements `AbstractAlgorithm`, is designed in a similar way. It takes a vertex, calls reachability with that vertex and calls the `finishedCC()` method with the connected component that was found by reachability. The `finishedCC()` method can be implemented by the user.

Apart from the interfaces, this library also contains default implementations of the algorithms. Four algorithms have been implemented: sequential reachability, parallel reachability, sequential connected components and parallel connected components. Parallel reachability is probably the most interesting one. It implements the `ParallelAlgorithm` interface, which specifies the basics that are needed to make an algorithm parallel. Because `Reachability` uses a stack, which is not synchronized, there are also synchronized methods needed for access to the stack, so `getNewUnexplored()` is synchronized and pops a vertex from the stack, and `stackEmpty()` is synchronized as well. The instance variables load, waiting and done are all needed for termination detection, which is described in section 5.

4.2 Design for graph structures

The graphs package contains many interfaces containing methods that are used by the algorithms. Apart from the general `Graph` interface and the `UndirectedGraph`, `DirectedGraph` and `AdjustableGraph` interfaces, each algorithm has a corresponding interface that specifies the required methods for this algorithm. A class diagram describing the graphs package can be found in Figure 2.

The `Graph` interface contains the most general methods, that any graph should have, at least for the algorithms that are designed now. Graphs should know for example how many edges and vertices they have, which vertices these are and the vertices that are connected to an edge.

The `DirectedGraph` interface is currently not used but might be used in future algorithms which require a directed graph, like strongly connected components. The methods that are specified return this extra information about the edges.

The `UndirectedGraph` interface does not have any methods yet, but it exists for classes to implement it to make clear that they are undirected. It would not make sense to implement both `DirectedGraph` and `UndirectedGraph`, because a graph cannot be both directed and undirected at the same time.

Like we mentioned above, each algorithm has a corresponding graph interface in our library. `ReachabilityGraph` is such an interface. It contains one method that is used by reachability, namely `reachableVertices()`. A `ReachabilityGraph` may be either undirected or directed.

The `ConnectedComponentsGraph` is a subclass of `ReachabilityGraph` and `UndirectedGraph`. Connected components does not work on a directed graph right now but can be easily adapted to change it to a strongly connected components algorithm. It is a subclass of `ReachabilityGraph` because connected components uses the reachability algorithm, and should therefore have access to the same methods that reachability has access to. There is also an extra method called `getAVertex()` which returns a random vertex, so connected components can be used without specifying a start vertex.

The `AdjustableGraph` interface represents a graph that can be adjusted by adding vertices and edges to it. This can be useful for generators of graphs that add vertices and edges one by one to the graph.

We have implemented a default implementation of a graph that is a `ConnectedComponentsGraph`, and can therefore be used by both algorithms that we implemented. Besides the methods from the implemented interfaces, there are no other methods in `SimpleGraph`. There are five variables to store the edges and vertices. This might seem superfluous because a graph can be stored by just a set of edges, but it makes the methods of the graph faster. For example, it is very easy to get the vertices that are reachable from a certain vertex v, we can just use `verticesWithReachables.get(v)`. This method is used a lot by reachability.

5. ALGORITHM IMPLEMENTATION

In this section we describe the algorithms we have already implemented. This may be helpful for implementing other algorithms to extend the library.

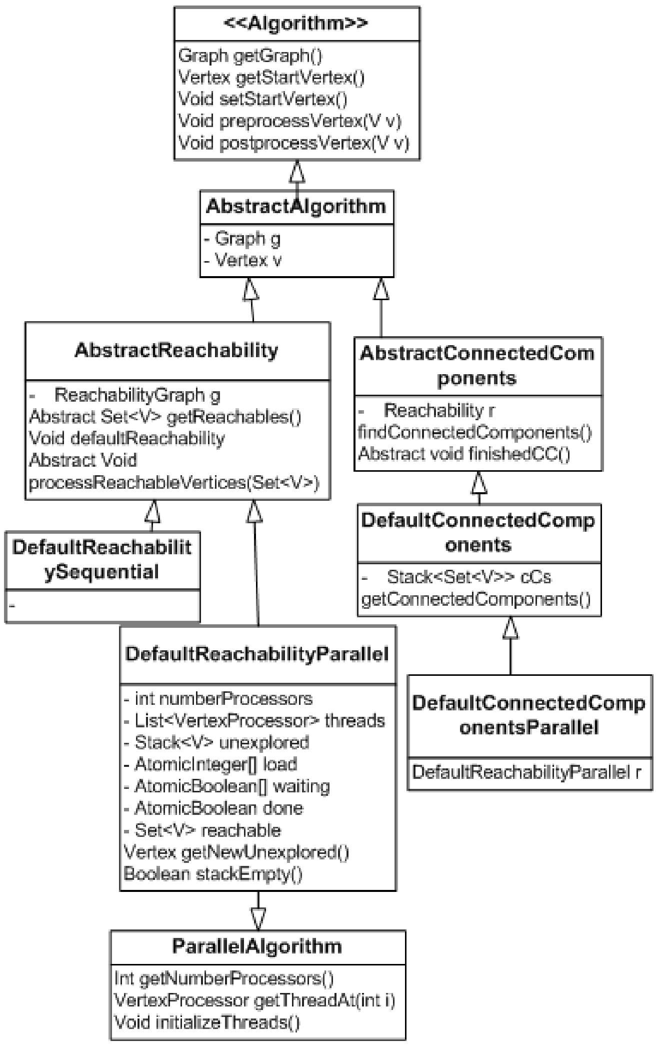
5.1 Sequential reachability

`Reachability` uses a `ReachabilityGraph`, which may be directed or undirected. A start vertex is specified in the constructor. The algorithm then finds all the reachable vertices from that start vertex.

5.1.1 Pseudocode: sequential reachability

```
Stack<Vertex> unexplored;
Vertex v;
ReachabilityGraph g;
unexplored.push(start vertex);

while(!unexplored.empty()){
  v = unexplored.pop();
  preprocessVertex(v);
  processReachableVertices(g.reachableVertices);
  postprocessVertex(v);
}
```



<<Algorithm>>
Graph getGraph()
Vertex getStartVertex()
Void setStartVertex()
Void preprocessVertex(V v)
Void postprocessVertex(V v)
AbstractAlgorithm
- Graph g
- Vertex v
AbstractReachability
- ReachabilityGraph g
Abstract Set<V> getReachables()
Void defaultReachability
Abstract Void processReachableVertices(Set<V>)
AbstractConnectedComponents
- Reachability r
findConnectedComponents()
Abstract void finishedCC()
DefaultReachabilitySequential
-
DefaultConnectedComponents
- Stack<Set<V>> cCs
getConnectedComponents()
DefaultReachabilityParallel
- int numberProcessors
- List<VertexProcessor> threads
- Stack<V> unexplored
- AtomicInteger[] load
- AtomicBoolean[] waiting
- AtomicBoolean done
- Set<V> reachable
Vertex getNewUnexplored()
Boolean stackEmpty()
DefaultConnectedComponentsParallel
DefaultReachabilityParallel r
ParallelAlgorithm
Int getNumberProcessors()
VertexProcessor getThreadAt(int i)
Void initializeThreads()

Figure 1. Algorithms

5.1.2 *How it works*

The reachability algorithm finds all the vertices that are reachable from a certain vertex. That means, it finds the connected component in which that vertex is situated. In this pseudocode, `unexplored` is a stack with unexplored vertices, and `result` is a set of reachable vertices from a certain start vertex.

The algorithm does the following:

1. Take an unexplored vertex
2. Call the `preprocessVertex()`. Here the user can specify what to do. In our default implementation of reachability, the vertex will be added to the set of reachable vertices.
3. The `reachableVertices()` method from the graph is called. This returns all the vertices that are reachable from v. How this is done can be specified by the user; in our implementation of the graph, a `Map <Vertex, Set<Vertex> >` is used to keep the number of calculations needed to a minimum.
4. Call the `processReachableVertices()` with the vertices that were found. Here, vertices that have not yet been explored will be added to `unexplored`.
5. Call the `postprocessVertex()`. The user can define what this method does. Our default reachability implementation does not do anything here.

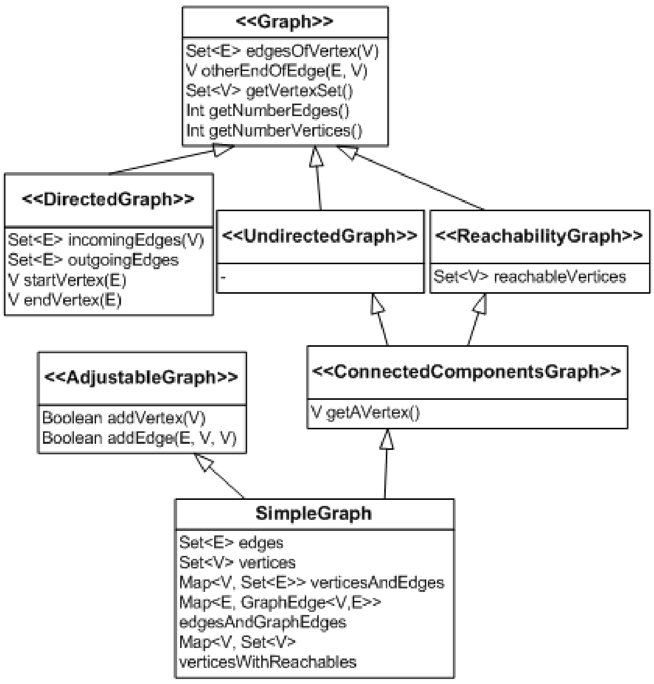


Figure 2. Graphs

It repeats steps 1 to 5 until `unexplored` is empty.

5.2 Sequential connected components

5.2.1 *Pseudocode: sequential connected components*

```
Vertex v;
Reachability r;
Stack<Vertex> unexploredVertices;
Set<Vertex> graphVertices

while(there are unexplored vertices in this
    graph){
  v = unexploredVertices.pop();
  r.setStartVertex(v);
  r.reachability();
  Set<Vertex> reachables = r.getReachables();
  finishedCC(reachables);
  graphVertices.removeAll(reachables);
}
```

5.2.2 *How it works*

This algorithm finds all the connected components of a undirected graph. It uses a reachability instance. This could be our sequential or parallel default implementation of reachability, or an implementation by the user.

The algorithm does the following:

1. Take a new, unexplored vertex;
2. Set this new vertex as the start vertex for reachability.
3. Perform reachability with this start vertex
4. Ask reachability for the connected component

5. Call the `finishedCC()` with this connected component. The user can define what happens here. Our default implementation pushes every connected component on a stack.

6. Remove this connected component from the set of vertices.

It repeates steps 1 to 6 untill all of the vertices of the graph have been explored.

5.3 Parallel reachability

5.3.1 the algorithm

The algorithm consists of two important parts. The constructor of `ParallelReachability` calls `startReachability()`, which then starts all of the threads and waits until they are finished. The `run()` method of the threads perform the termination detection, which in turn calls the reachability algorithm.

Pseudocode: startReachability.

```
int numberProcessors;

for(int i = 0; i < numberProcessors; i++){
  getThreadAt(i).start();
for(int i = 0; i < numberProcessors; i++){
  getThreadAt(i).join();
```

Pseudocode: doReachability.

```
int doReachability(){
  Vertex v = getNewUnexplored();
       if(v!=null){
           preprocessVertex(v);
           processReachableVertices(
               g.reachableVertices);
           postprocessVertex(v);
           return getStackSize();
      } else return 0;
}
```

How it works.

The algorithm does the following:

1. Start all of the threads and wait until they are finished.

2. All of the threads run the termination detection which is described below.

3. Termination detection calls the reachability algorithm described above

4. `doReachability()` tries to get a new vertex, when this is not possible (v == null) 0 will be returned to indicate that the stack is empty and this method exits.

5. If v!=null, the reachability algorithm is executed just like in sequential reachability.

6. The stack size is returned, and this will be used as the new load for this thread.

5.3.2 Termination detection

A termination detection algorithm is needed for the algorithm to terminate normally. A lot of research has been done on termination detection. Users can implement their own version of termination detection since this algorithm works indepent of the reachability algorithm. Of course, one can also use the algorithm we provided. We used the algorithm described by Sanders [26], and altered it slightly to make it work in Java. Here we explain how this algorithm works.

Pseudocode: termination detection [26].

```
AtomicInteger[numberProcessors] load;
AtomicBoolean[numberProcessors] waiting;
AtomicBoolean done;

while(!done){
  load[threadNumber] = reachability();
  for(int i = 1; load[i-1]==0 &&  i <
      numberProcessors; i++){
     if (i== (numberProcessors-1) && load[i]==0)
             done = true;
   }
  if(load[threadNumber]==0)
      waiting[threadNumber]=true;
  else
      for(int i = 0; i < numberProcessors; i++)
          waiting[i]=false;
  while(waiting[threadNumber] && !done) ;
}
```

How it works.

This algorithm is called by every thread performing reachability.

The algorithm does the following:

1. Call reachability. `Reachability` returns the number of vertices that are left on the stack at the end, this will be the new load for this thread.

2. If the load of every single thread is 0, it sets done to true.

3. If the load of this thread is 0, it sets waiting to true for this thread. Else, it sets waiting to false for all other threads. Since there are more vertices on the stack no thread should be waiting.

4. If this thread is waiting and the algorithm is not done yet, it will stay in this while loop.

5. This algorithm keeps calling reachability until one of the threads sets done to true.

5.4 Parallel connected components

Our default parallel connected components algorithm extends the default sequential connected components algorithm. The only thing that is changed is the type of reachability it uses: instead of a `Reachability`, it uses `DefaultReachabilityParallel`.

This is the only way we could make connected components parallel. We cannot, for example, work on several connected components at once, because we do not know beforehand which vertices will be in which connected component. Therefore using the parallel reachability was the only thing we could do to make connected components parallel, but this should be good enough, especially for large graphs.

6. CONCLUSION

We conclude this paper with a discussion and some suggestions for future work.

6.1 Discussion

6.1.1 Design choices

Here we reflect on some of the design choices we made.

Reuse of reachability.

For our connected components algorithm, we use the reachability we have implemented. The parallel connected components uses parallel reachability and does not need any other parallelization. Because of this, very little extra lines of code had to be written for connected components.

Graph structures.

We have designed a graph interface for every type of algorithm. In Figure 2 this can be seen: there is a `ReachabilityGraph` and a `ConnectedComponentsGraph`. This makes explicit which methods in a graph are needed by the algorithm. It is also clear from the class diagram that the connected components algorithm needs an undirected graph, whereas both directed and undirected graphs can be used for reachability. These interfaces make it easier to define a graph structure for a certain algorithm, and we recommend providing a new interface for every new algorithm.

Interfaces and abstract classes.

We have used a lot of interfaces and abstract classes, as can be seen in the class diagrams 1 and 2. This might seem overly complex, but it helps when one wants to implement their own classes. For example, there is an interface `Algorithm` and an abstract class `AbstractAlgorithm`. Without the interface, it would not be able to make another implementation of the methods in `AbstractAlgorithm` without significant code changes. Without the abstract class, there are no default methods that can be used for algorithms that are added in the future, and everyone would have to implement the methods by themselves, creating a lot of duplicate code.

6.1.2 Java programmers

An obvious advantage for Java programmers is that this library saves them a lot of work. Without a parallel graph library in Java they would have to implement their own algorithms or use algorithms in other languages. It would require a lot of work to do so.

6.1.3 Use and maintenance of the library

The advantages of Java over other languages are valid for our library as well. Java is rather easy to understand and makes maintaining and extending programs easy, at least for Java programmers. This is true for our library as well. Also, because the entire library is generic, any the algorithms can be used for any type of graphs.

6.1.4 Speed

One of the things that would help answering our research question, is benchmarking our parallel algorithms and comparing them to parallel algorithms in other languages. If our algorithms are faster, then that would contribute to the practical merit of our library, but of course speed is not the only thing that matters. For now we cannot say anything about the speed (dis)advantages of parallel over sequential algorithms and our library versus libraries in other languages.

6.2 Future work

6.2.1 Benchmarking

Due to the limited time of our research, we have designed the algorithms but have not benchmarked them. Benchmarks can be performed to see how the parallel algorithms compare to their sequential variants. Also, our parallel algorithms could be compared to other parallel algorithms in other languages like C++, to see if a parallel graph library in Java has a speed advantage or disadvantage over one in another language.

6.2.2 Proving termination detection

The termination detection that we used has not formally been proved. We have reason to believe that it works because it ran for many hours without showing errors, and we took an existing algorithm [26] which we slightly altered to make it work in Java. Although we have let it run for many times without errors, a formal proof is needed to be sure that the algorithm always terminates normally and gives the expected results.

6.2.3 Extending the library

We have only implemented two algorithms: reachability and connected components. The library can be extended with other parallel graph algorithms, such as topological sort and breadth first search. The design of our interfaces and algorithms can be used to implement other algorithms. The easiest way to add an algorithm is by extending the `AbstractAlgorithm` class and adding the algorithm specific methods to it. To make a parallel algorithm, our termination detection algorithm can be used. Make sure all access to shared data structures is synchronized. We also suggest adding a graph interface for each algorithm, so it is clear what methods are required in a graph to perform the algorithm.

7. REFERENCES

[1] W.R. Adrion, M.A. Branstad, and J.C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: an adaptive, generic parallel c++ library. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 193–208, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, and N.M. Amato. Stapl: A standard template adaptive parallel c++ library. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, page 10, 2001.

[4] M.Y. Becker and I. Rojas. A graph layout algorithm for drawing metabolic pathways. *Bioinformatics*, 17(5):461, 2001.

[5] J.A. Bondy and U.S.R. Murty. *Graph theory with applications.* MacMillan London, 1976.

[6] J.R. Burch, E.M. Clarke, et al. Symbolic model checking: 1020 States and beyond* 1. *Information and computation*, 98(2):142–170, 1992.

[7] S. Conchon, J.C. Filliâtre, and J. Signoles. OcamlGraph. `http://ocamlgraph.lri.fr/`.

[8] S. Conchon, J.C. Filliâtre, and J. Signoles. Designing a generic graph library using ML functors. *TFP*, 7:124–140, 2008.

[9] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[10] DedaSys. Programming Language Popularity, 2010. `http://www.langpop.com/`.

[11] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library. *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, 2006.

[12] L. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. *Parallel and Distributed Processing*, pages 505–511, 2000.

[13] M. Forster, A. Pick, and M. Raitner. Graph Template Library. `http://www.infosun.fmi.uni-passau.de/GTL/`.

[14] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 437. ACM, 2005.

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

[16] J. Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 16–25. ACM Press, 1994.

[17] M. Hribar, V. Taylor, and D.E. Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *Proceedings from ISUG '97 Conference*, 1997.

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

[19] V. Kumar and V.N. Rao. Parallel depth first search. Part II. analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[20] A. Leung and L. George. MLRISC Annotations. 2007.

[21] S.S. Lumetta, A. Krishnamurthy, and D.E. Culler. Towards modeling the performance of a fast connected components algorithm on parallel machines. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, pages 32–49. ACM Press, 1995.

[22] K. Mehlhorn, S. Näher, and C. Uhrig. The LEDA platform for combinatorial and geometric computing. *Automata, Languages and Programming*, pages 7–16, 1997.

[23] B. Naveh. JGraphT, 2008. `http://jgrapht.sourceforge.net/`.

[24] V.N. Rao and V. Kumar. Parallel depth first search. Part I. implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[25] J.H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[26] P. Sanders. *Load Balancing Algorithms for Parallel Depth First Search*. PhD thesis, University of Karlsruhe, 1997. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag.

[27] J. Siek, L.Q. Lee, A. Lumsdaine, L.Q. Lee, L.S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, M. Heroux, et al. The Boost Graph Library: User Guide and Reference Manual. In *Proceedings of the*, volume 243, pages 112–121. ACM Press, 2002.

[28] R. Tamassia, M.T. Goodrich, L. Vismara, M. Handy, G. Shubina, R. Cohen, B. Hudson, R.S. Baker, N. Gelfand, and U. Barandes. JDSL: The data structures library in Java. *Dr. Dobb's Journal*, 26(4):21–31, 2001.

[29] J. Träff and C. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *Parallel algorithms for irregularly structured problems*, pages 183–194, 1996.