Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

The applicability of Divide-and-Conquer: Graph Traversal in parallel on a computational grid

Patrick Goddijn

June 8, 2017

Supervisor(s): Rob van Nieuwpoort

Signed:

Abstract

The Divide-and-Conquer based *Satin* system was developed to make grid computing accessible and efficient. In the past, Satin was proven to be portable and efficient for several classes of applications. It is interesting to examine how extensive Satin's applicability is. Therefore, we wish to investigate the expressiveness of the programming model. To test this the Berkeley Dwarfs are used. These Dwarfs capture the thirteen most important computational and communication patterns used in applications. This thesis focuses on one specific Dwarf: graph traversals. We test if graph traversals can be defined using the Divide-and-Conquer model and if this results in an efficient Satin application. To test the Satin application, several implementations are made to demonstrate some Satin functionalities. These implementations are compared with each other and with sequential applications. The application described in this thesis is a quite elegant way to write graph traversal in a Divide-and-Conquer fashion. Therefore, we are positive on the expressiveness of Satin and the Divide-and-Conquer model in general for this type of algorithm. However, it does not result in an efficient application. This can be attributed to each job being too small and easily doable sequentially. The Divide-and-Conquer application results primarily in parallel overhead. This means that it is possible, but not useful to create graph traversal applications using Satin.

Contents

CHAPTER 1

Introduction

This chapter gives an introduction to this paper, and explains the goals it hopes to achieve. It also gives a layout of the rest of this thesis.

1.1 Context

In the world of computer science there always is a demand for faster applications. Therefore better hardware has to be developed, and for this hardware efficient software should be written. In the last decade, the research into faster processing units has slowed down because the so called *power wall* has been hit. This gave the necessity for looking into parallelism by means of many (micro) processors [3]. To keep enjoying large performance growths the focus of the industry has shifted to developing parallel hardware. One of the main problems with parallel computing is the need for parallel software. Many programmers face problems when coding parallel applications, because it is difficult due to problems like load balancing, race conditions, deadlocks and increased pressure on shared resources like caches. Also, most compilers and operating systems are not conform to it [3]. There is an ever-growing need for innovation in this field.

1.2 Subject

One method for parallelisation is grid computing. A grid is a group of computational resources working together. These machines are all fully functional. The full definition as given by M.L. Bote-Lorenzo et al. [7] is:

> *a large-scale geographically distributed hardware and software infra-structure composed of heterogeneous networked resources owned and shared by multiple administrative organisations which are coordinated to provide transparent, dependable, pervasive and consistent computing support to a wide range of applications. These applications can perform either distributed computing, high throughput computing, on-demand computing, data-intensive computing, collaborative computing or multimedia computing.*

A research group at the VU (Free University of Amsterdam) had come to the conclusion that large-scale grid computing has a much broader application than currently is utilised. Grid computing is used for running independent jobs in parallel or trivial master/worker programs. According to the research group however, most parallel applications can be efficiently run on grids. Yet, most parallel programming models are not a good fit. To resolve this problem, the group has implemented a new system called **Satin**, based on the *extended Divide-and-Conquer model* [19]. This system can be used for a broad range of parallel applications. It also adds an abstraction layer for grid computing, making it easier to use for programmers. With Satin they hope to open up parallel grid computing by showing its possibilities and making it easier to use.

1.3 Desire

For Satin and parallel grid computing to be viable it should be possible to write most applications in the Divide-and-Conquer paradigm. It is the hypotheses of the aforementioned research group that almost all applications can be written in a this style. However, there is no easy way to prove it. Furthermore, even if almost all applications can be implemented in Satin, it does not mean they are *efficient* and *scalable*. To test the hypothesis and to show which kinds of applications can be made efficient and scalable, the Berkeley Dwarfs come into play.

Researchers from the Berkeley University have come up with thirteen algorithmic methods which each describe a pattern of computation and communication used by the most important and common applications [2]. When it is proven for all these Dwarfs that they can be implemented in Satin and can be run efficiently on a grid, this will apply to a large number of applications.

1.4 Research questions

Many of the Dwarfs have already been implemented and tested, and the time set for this research is limited. Therefore the scope of this thesis has been limited to one of the thirteen Dwarfs, which has not yet been implemented in Satin. This Dwarf is *Graph Traversal*. The goal of this paper is to check the hypothesis that graph traversal algorithms can be written in a Divide-and-Conquer style. This leads to the research question:

Can, with the use of Satin, a Divide-and-Conquer application be implemented for the Graph Traversal Berkeley Dwarf? (1)

It also should be checked whether or not this implementation can be efficiently run on a grid, and if it is sufficiently scalable. This goal has resulted in the following research question:

Does this application run efficiently on a computational grid in a scalable way? (2)

1.5 Thesis outline

To formulate answers to these research questions this paper is structured in the following way. Chapter 2 gives more theoretical background on Satin, the Berkeley Dwarfs and specifically the Graph Traversal Dwarf. Chapter 3 will illustrate the design of the graph traversal application, both the sequential and the Satin versions. Chapter 4 will explain the setup of the experiments to test the applications, and which kind of results these experiments will produce. Chapter 5 will display the results received from the experiments. Chapter 6 contains the conclusions drawn from the results of the experiments and the outcome of this research.

CHAPTER 2

Theoretical background

This chapter provides some theoretical background about Satin, the Berkeley Dwarfs and specifically the Graph Traversal Dwarf.

2.1 Satin

As mentioned before, Satin is a programming system for creating parallel Divide-and-Conquer applications on a computational grid. It was developed at the VU by R. van Nieuwpoort et al. [16]. Satin implements a high-level programming system, to take nearly all the parallelisation trouble out of the hands of the programmer. Some design choices, features, and results of the system will now be discussed here. Take note that only information deemed important for this thesis is discussed, for all the information refer to the Satin papers [16–19] .

2.1.1 Divide-and-Conquer

Satin is specifically designed for Divide-and-Conquer applications. The Divide-and-Conquer paradigm is an computational paradigm which breaks down a problem in similar sub-problems. Every sub-problem then is recursively split again. This continues until every sub-problem is trivial enough to solve directly. These sub-solutions are then combined to the full solution of the problem. Hence divide (into sub-problems) and conquer (create solution). One of the most famous examples of a Divide-and-Conquer algorithm is Mergesort. Mergesort sorts a list by splitting the list into two sub-lists and calling Mergesort recursively over the two new lists. When the most trivial sub-problem, an empty list or a list with one item, is reached it is seen as sorted. Then the sub-lists are merged together while sorting the items from the sub-lists. When the algorithm is done, the list is complete again, as well as sorted.

One of the reasons Satin is tailored to Divide-and-Conquer applications is because it is inherently parallelizable. Each sub-problem can naturally be solved on a different processor. Divide-and-Conquer is also very hierarchical which fits the hierarchical nature of computational grids very well. However, Divide-and-Conquer is a quite specific model. To make it more generally usable it was extended by some features, to create the *Extended Divide-and-Conquer* model. These features, shared data and speculative parallelism, also contribute to a better scalability and speedup.

Even though Satin is based on this Extended Divide-and-Conquer model, it is also possible to create master-worker applications with it. These applications are regarded as a subclass of Divide-and-Conquer with only one level of division. However, tests have shown these applications to be less efficient than Divide-and-Conquer applications, because there is a single central bottleneck (the master) [18].

2.1.2 Design choices

Satin is specifically designed to run on computational grids. This explains some of its design choices. For instance the choice for the focus on Divide-and-Conquer applications, as mentioned above, was made because both the Divide-and-Conquer model and grids are hierarchical. Therefore, the applications map well on a hierarchical grid. Grids also have the attribute that they are heterogeneous. This means that they consist of different software and hardware resources. These resources can consist of different types of machines, with different CPU's, operating systems and installed software. This makes it hard to create a program which can be run on all nodes. This was solved by making Satin based on Java, which makes it possible to run a Satin application on any machine with a Java Virtual Machine. This usage of Java's *write-once run everywhere* benefit makes sure any Satin program can be run on any part of a grid, without problems and extra work for the programmer.

The design choice to extend the pure Divide-and-Conquer model for Satin, as mentioned, stemmed from the desire to make Satin as wildly applicable as possible. The most important addition to the Divide-and-Conquer model is the possibility of shared data. In pure Divide-and-Conquer, the only data-sharing is using the input parameters and the returned results. In Satin shared-objects are added to this. These shared-objects are implemented weakly consistent. This means that every update to the shared-object spreads across the grid. It however does not wait for every update to reach all grid-parts, unless explicitly specified by the programmer. Therefore not all processors have a consistent view of the object. This decision was made to make the shared-object work efficiently on the grid, which otherwise would not have been possible [19].

The other main addition to the Divide-and-Conquer model is support for speculative parallelism. This is useful when creating parallel work that might turn out not to be necessary. The work can then be done in parallel, but once it is found not to be needed anymore, it can be aborted. One of the main usages for this is search algorithms, where once the result is found, all the other work can be stopped. Speculative parallelism is added using an inlet/abort system. The abort-messages are also weakly-consistent, for the same reason as the shared-objects. This means not all processors will get the abort-message at the same time. However, this does not matter for the correctness of the system [19].

The goal of Satin is to run all the, by Divide-and-Conquer created, sub-routines in parallel. To schedule all these sub-routines over multiple processors a work stealing strategy is used [17]. This is done by keeping a queue of all jobs that need to be done. Whenever a processor has run out of work in its own queue, it will attempt to steal work from the queues of other processors. If it is successful, the job will be removed from the other processor's queue and added to the queue of this one. To utilise the hierarchical nature of Divide-and-Conquer, steals will take jobs from the bottom of the queue. The reason behind this is that those are, most of the time, larger jobs. Satin uses a special kind of work stealing, called CRS (Cluster-aware Random Stealing) [18]. This means that Satin will always attempt to steal jobs from processors on another cluster. The job-stealing is important, because the more successful steal attempts are made, the more in parallel an application runs.

2.1.3 The model

Satin is based on Java, and can be compiled into valid Java-code using the Satin pre-compiler. The resulting model extends the single-threaded Java model, which means there is no need to make use of other parallel methods of Java. The extensions added to the Java language contain some primitives. The two pure Divide-and-Conquer primitives are *spawn* and *sync*. A spawn is a special invocation of a so called Satin method. This method will run parallel to the method which spawned it. The sync waits for all the spawned calls in the method to finish. Only then the return value of all methods can be used. Spawn and sync know equivalents in many parallel systems. An example can be found in Listing 2.1 (Example code taken from [17]).

```java
// Mark the fib method as a spawn operation.
interface FSpawns extends satin.Spawnable {
    int fib(int n);
}

```

```java
class Fibonacci extends satin.SatinObject implements FSpawns {
    // Spawnable method. Calling it can be done normally.
    public int fib(int n) {
        if(n < 2) return n;
        int x = fib(n - 1);  // Spawn a new method
        int y = fib(n - 2);  // And spawn another one
        sync();  // Wait for both spawned methods to finish
        return x + y;
    }

    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        int result = f.fib(10); // Spawn method
        sync(); // Wait for spawned method to finish
        System.out.println("Fib 10 = " + result);
    }
}
```

Listing 2.1: Example of a Satin program with spawns and syncs (Fibonnaci calculator)

As mentioned before, Satin extends pure Divide-and-Conquer by allowing sharing data between subroutines. This happens by replicating objects on all processors that can access the object. These shared objects have global methods which are applied to all replicates, and local functions which are only applied to the local copy. A shared-object-class extends the `satin.SharedObject` class and all functions extended by the `satin.GlobalMethods` interface are global. If the programmer wishes to assure consistency, the guard consistency model is used. This can be used to check the state of a shared object. If the guard method finds the object not in the right state it will wait for an update to be pushed, or it fetches a newer replica. A guard is specified by adding a public boolean function to the class which uses the shared-object. This function should be called `guard_<name_of_spawnable_function>`. The spawn of the guarded function only continues if the guard returns true.

Satin also has functionality for speculative parallelism, called the inlet/abort mechanism. Inlets, set by making a class which extends `satin.Inlet`, execute code as soon as a certain task has finished. Inlets are implemented in a Java exception handler, which catches the ending of the task, allowing the inlet to execute. The spawn-able method throws this inlet when it is done, and the caller one can catch it to abort the remaining spawns, or do something else with it.

2.1.4 Results

Satin was tested on different clusters, including the DAS 3. The results will be shortly touched upon here, but all tests and results can be found in the Satin paper [19].

A few real-world applications were tested on a single cluster. This showed that for different applications Divide-and-Conquer can be a bit to a lot better compared to master-worker applications. It also showed the worth of the shared objects and the abort mechanism for some applications. The speedups achieved show the worth of Satin, even on a single cluster.

The shared objects versions of the applications mentioned above were also tested on a wide area grid with five clusters, with varying CPU speeds and numbers of cores (DAS 3). This showed that Satin can scale really well to larger networks of machines and clusters. It showed efficiency of 72 to 82 percent, where 100 percent efficiency is a linear speedup of the number of cores added.

2.2 Berkeley Dwarfs

More than ten years ago, a group Berkeley researchers predicted the effects of the steps computing has taken towards parallelism [2]. They found that one of the biggest obstacles to innovation in parallel computing is that it is unclear how to test new ideas in a good way. It is difficult to set a benchmark suite of existing programs to drive the innovation in parallel programming, even though it is the conventional way of evaluating innovations. They propose a more abstract way of setting application requirements, which are not bound to specific applications or hardware optimisations. To achieve this, thirteen Dwarfs have been defined, where each captures a different computational pattern [2]. These thirteen Dwarfs can be found with a short description in Table 2.1 on page 13.

The Dwarfs are defined at a high abstraction level, disconnected of implementation specifics. This keeps them relevant as implementations change and innovate. Together they span the core computation and communication for a large variety of the most important applications. Because they span so many patterns, they are extremely useful to evaluate new innovations in the fields of architecture and programming models. When a parallel innovation gets good performance for all the thirteen Dwarfs, the expectation is that it will generally perform well on applications in the future. That is why the creation and testing of Satin implementations of these Dwarfs is interesting.

2.2.1 Graph traversal

The scope of this thesis contains the testing of only one of the thirteen Berkeley Dwarfs. The subject is the ninth Dwarf, Graph Traversal. Graph traversal is the traversal over a number of objects, or nodes, examining characteristics of these objects and then following links to other objects. According to the researchers from Berkeley [2]: *It typically involves indirect table lookups and little computation.* This makes it quite hard to make efficient in parallel.

This Dwarf was found by the Berkeley researchers when examining collections of benchmarks, specifically in the EEMBC benchmark for embedded computing. The Graph Traversal Dwarf was also found in all three application domains they looked at afterwards. These three domains are machine learning applications (Decision trees and Bayesian networks), computer graphics and games (collision detection, depth sorting and hidden surface removal), and databases (transitive closure). This was enough to prove that Graph Traversal is a specific algorithmic pattern which occurs in many applications. This justified making it one of the Berkeley Dwarfs, and subsequently made it interesting to test out with Satin.

Dwarf	**Description**
1. Dense Linear Algebra	Data are dense matrices or vectors. Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns.
2. Sparse Linear Algebra	Data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all of the nonzero values. Because of the compressed formats, data is generally accessed with indexed loads and stores.
3. Spectral Methods	Data are in the frequency domain, as opposed to time or spatial domains. Typically, spectral methods use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others.
4. N-Body Methods (e.g., Barnes-Hut)	Depends on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an O(N2) calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to O(N log N) or O(N).
5. Structured Grids	Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between 2 versions of the grid. The grid maybe subdivided into finer grids in areas of interest (Adaptive Mesh Refinement); and the transition between granularities may happen dynamically.
6. Unstructured Grids	An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those neighboring points.
7. Monte Carlo	Calculations depend on statistical results of repeated random trials. Considered embarrassingly parallel.
8. Combinational Logic (e.g., encryption)	Functions that are implemented with logical functions and stored state.
9. Graph traversal (e.g., Quicksort)	Visits many nodes in a graph by following successive edges. These applications typically involve many levels of indirection, and a relatively small amount of computation.
10. Dynamic Programming	Computes a solution by solving simpler overlapping subproblems. Particularly useful in optimization problems with a large set of feasible solutions.
11. Backtrack and Branch+Bound	Finds an optimal solution by recursively dividing the feasible region into subdomains, and then pruning subproblems that are suboptimal.
12. Construct Graphical Models	Constructs graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models.
13. Finite State Machine	A system whose behavior is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states.

Table 2.1: All 13 Berkeley Dwarfs with description, as given in by Asanovic et al. [2].

CHAPTER 3

Design

In this chapter, we explain the sequential and parallel graph traversal algorithms implemented. There is a special focus on the different versions of the Satin implementation.

3.1 Sequential implementation

To find if a parallel version works, first a sequential version of the same application has to be created. For this research, both a breadth first search (BFS) and a depth first search (DFS) implementation have been made in ordinary sequential Java. Both implementations try to find a target node in a graph as fast as possible. A start node, or root node, is given. From this node, new adjacent nodes have to be traversed to find the target node. The BFS version keeps a queue with all the adjacent nodes that can be traversed, while the DFS version keeps a stack. This means that the BFS version always traverses to the node which it found first, while the DFS version traverses to the node found last. The queue and stack are implemented by a `LinkedList` Java data-structure. The queue uses `add` and `poll` to add and remove nodes from it. The stack uses `push` and `pop` for the same actions.

The graph is represented in Java by an object of the class `Graph`. This `Graph` has as attributes the number of nodes and an adjacency list of the graph. This is an array with an entry for all nodes in the graph, where each entry consists of a list of all nodes that are accessible from this node. To clarify, an example for the graph in Figure 3.1 is given in Table 3.1. The adjacency list in `Graph` allows for a relatively compressed representation of a graph. It is implemented during the loading of the graph as an Arraylist-array, which is converted to a 2D-array after the graph is completely read. This is because Arraylist are easier to add to, while arrays are smaller and faster to read in memory. We also keep a list which stores for each node if it is visited already. This avoids double work and guarantees eventual termination.

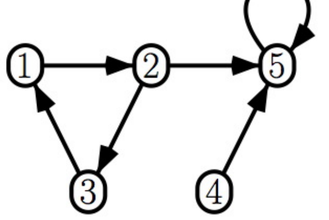

Figure 3.1: A small directed graph [1]

Node	Adjacent nodes
1	2
2	3, 5
3	1
4	5
5	5

Table 3.1: The Adjacency list of this graph

[1]Source: Wikimedia Commons

3.2 Divide-and-Conquer algorithms

To create a Divide-and-Conquer application in Satin, some elements of the two sequential implementations can be re-used. The `Graph` class and visited list can be the same. The stack and queue elements are removed to allow for parallel traversal. It seemed that the most natural Divide-and-Conquer manner of traversal is the splitting of the graph in sub-graphs, where each process can traverse one sub-graph. Unfortunately, the splitting of a graph in sub-graphs is a very hard and intensive problem. Therefore, the algorithm created simply traverses a node, and *spawns* a new process for every adjacent node. This way, the graph gets effectively separated in sub-graphs, one for each new spawned method. These sub-graphs unfortunately cannot be separated into new graph objects. They also will have huge overlapping parts, because for large graphs most nodes can be traversed to from most other nodes. This however will not be a problem, because these overlapping nodes will be processed by the first process that gets to them. Afterwards they will be set as visited, which will withhold other processes from gaining access to them, effectively removing them from these other sub-graphs.

This algorithm spawns a great number of processes. This would really eat into the stack, probably making it run out of memory. Therefore every process turns into an sequential BFS or DFS algorithm when enough sub-processes have been spawned. This is measured by keeping track of the distance traversed from the root-node. This assures that enough processes will be spawned, but not too many. This is demonstrated in a upcoming chapter in Table 5.3.

3.2.1 Different versions

The Divide-and-Conquer algorithm for graph traversal shown above has still a couple of flaws. These flaws are solved by using the previously mentioned extended features in Satin. One of the flaws in the algorithm is that it does not terminate until all nodes in a graph have been visited. This leads to an implementation which always takes the worst-case time to complete. To solve this, we made a new implementation which makes use of Satin's Inlet/Abort mechanism. When the target node is found the new `SearchResultFound` inlet is activated and every recursive parent throws it. Every other sub-process is then aborted, hopefully leading to a faster termination which leaves nodes not visited. The worst-case version of this implementation still traverses every node, but this will not happen unless the target-node is not in the graph or not accessible from the root-node.

This implementation now works fine on a machine with shared memory. However, when using more than one machine, which is more or less the point of working with grids, there will be a lot of double work. The reason for this is that the visited list is copied to each new machine, but updates are not propagated to remote machines. Therefore the protection the visited list offers to the sub-graph model falls away, leading to a lot of needless work. This is where the shared data mechanism of Satin comes in. The visited list is replaced by a visited object, which extends `satin.SharedObject`. It has an update function which sets a node in the list to visited. This update function can be called every time a node is visited, to sent this update to all other machines. This way the amount of double work gets minimised. Because each update is a message that has to be shared between all processes, doing this for every visited node can be heavy on the network. Therefore, this implementation also has a local update and a multiple-node-update. Thus every new node that has not been visited gets locally updated, and every once in a while all new updates get broadcast to the other processes. This way, the best of both worlds is achieved, because processes do not do to much double work, but the application does not slow down too much from all the update-messages.

To evaluate how much the extended features of Satin add to this algorithm, we made a different version for each. We can compare these with each other to show what implementation works the best. The following four versions are made to do this. A version that is just normal search without any extensions, a version with only Inlet/Aborts, a version with Shared Objects and a version with both added features. Henceforth, these will be referenced by Search, Search IA, Search SO and Search IA SO.

3.3 Implementation

The four versions all contain a spawnable search function. This function does the traversing of a graph, while looking for the target node. It keeps a list of all the visited nodes. To illustrate, the Listing 3.1 contains the code used for the standard Search. The interface marks the search function as spawnable, and the class `Search` is a Satin object. Search gets called on a node in the graph, and first checks if this is the target node. Then it checks if this branch has produced enough parallelism. If so, it continues sequentially. Else, it spawns a new search for every adjacent node that was not visited before, and waits for all of them to complete. This process produces the Divide-and-Conquer style recursion.

```java
// Mark the search method as a spawn operation.
interface SearchInterface extends ibis.satin.Spawnable {
    public void search(Graph g, int node, int target, int c, boolean[] visited);
}

// Search is a Satin object, and uses the SearchInterface.
public final class Search extends ibis.satin.SatinObject implements
    SearchInterface, Serializable {

    // Do Search traversal.
    public void search(Graph g, int node, int target, int c, boolean[] visited) {
        // Mark the current node as visited.
        visited[node] = true;

        // Check if the target is found.
        if(node == target) {
            System.out.println("FOUND " + target);
            return;
        }

        // If there is enough parallelism, continue sequentially.
        if(c > MAX_SPAWN_DEPTH) {
            ...
        }

        // For every, non visited, adjacent node: spawn a search.
        for (int i : g.adj[node]) {
            if (!visited[i]) {
                search(g, i, target, c + 1, visited);
            }
        }
        // Wait for all spawned processes to complete.
        sync();
    }
}
```

Listing 3.1: Implementation of the Satin search function

The Inlet/Abort system used in the Search IA and Search IA SO versions, does not change a lot about this implementation. The search function has to throw a `SearchResultFound`. This has to be caught when this function is called externally. The implementation of Search IA is shown in Listing 3.2.

```java
// Mark the search method as a spawn operation.
interface Search_IA_Interface extends ibis.satin.Spawnable {
    public void search(Graph g, int node, int target, int c, boolean[] visited)
    throws SearchResultFound;
}

// The class of the thrown object, which extends the Satin Inlet.
public class SearchResultFound extends ibis.satin.Inlet {
    int count;

    SearchResultFound(int count) {
        this.count = count;
    }
}

```

```java
// Search is a Satin object, which throws an Inlet.
public final class Search_IA extends ibis.satin.SatinObject implements
    Search_IA_Interface, Serializable {

    // Do Search traversal.
    public void search(Graph g, int node, int target, int c, boolean[] visited)
    throws SearchResultFound {
        ...

        // Check if the target is found.
        if(node == target) {
            System.out.println("FOUND " + target);
            throw new SearchResultFound(c);
        }

        ...
    }

    public static void main(String args[]) {
        // Prepare Search.
        ...

        // Run the Search, and abort when the found-inlet is thrown.
        try {
            s.search(g, root, end, 0, visited);
            s.sync();
        } catch (SearchResultFound x) {
            s.abort(); // kill other jobs that might still be running.
            return; // return needed because inlet is handled in separate threat.
        }
}
```

Listing 3.2: Implementation of the Satin search function, with IA

The versions with shared objects are nearly the same as the ordinary Search, but the visited lists are replaced by `Visited` objects. These have the visited list as variable. Because the object is a shared object, it can have write methods, which broadcast the writes to all processors. These write methods are given in the interface. All other methods only have an effect on the local instance. Listing 3.3 shows the implementation of the shared object.

```java
// Mark the update methods, which should be broadcast.
interface VisitedInterface extends ibis.satin.WriteMethodsInterface {
    public void update(int index);
    public void update_list(int[] updates);
}

// The Visited class is a shared object with a visited list.
final class Visited extends ibis.satin.SharedObject implements VisitedInterface,
    Serializable {
    private boolean[] visited;

    public void set(boolean[] v) { visited = v; }

    public boolean get(int i) { return visited[i]; }

    public void update(int index) { localUpdate(index); }  // write method

    public void localUpdate(int index) { visited[index] = true; }

    public void update_list(int[] updates) {  // write method
        for(int i : updates) {
            visited[i] = true;
        }
    }
}
```

Listing 3.3: Implementation of the shared objects

CHAPTER 4

Experiments

This chapter explains how the experiments are conducted. It explains the setup, and which gauges are measured.

4.1 Setup

All following experiments are conducted on the DAS-5 (Distributed ASCI Supercomputer 5) [5]. This cluster-based supercomputer has 5 geographical sites where it has a total of 6 clusters set up. An overview of the specifics of these clusters is given in Table 4.1. The cluster used in the experiments is the one located at the University of Amsterdam.

Cluster	**Nodes**	**Type**	**Total Cores**	**Speed**	**Memory**
VU	68	dual 8-cores	1088	2.4 Ghz	64 GB
LU	24	dual 8-cores	384	2.4 Ghz	64 GB
UvA	18	dual 8-cores	288	2.4 Ghz	64 GB
TUD	48	dual 8-cores	768	2.4 Ghz	64 GB
UvA-MN	31	dual 8-cores	496	2.4 Ghz	64 GB
ASTRON	9	dual 8/10/14-cores	228	2.6 Ghz	128/512 GB

Table 4.1: The clusters of the DAS-5

To test the graph traversal implementations described in Section 3.2, some graphs have to be created or acquired. To get some real world examples of graphs, the SNAP Dataset (Stanford Network Analysis Project) [11] and KONECT [9] were utilised. The used graphs are presented in Table 4.2. Both graphs are really large, but can still be read from file and build in a relatively small amount of time. This is due to the reformatting of the graph-files from the standard ASCII file format to a straight binary one, to reduce the file sizes and read times.

Name	**Directed**	**Nodes**	**Edges**	**Diameter**	**Avg. Degree**	**Read time**	**Citations**
soc-Livejournal1	Directed	4,847,571	68,993,773	16	14.23	30s	[12] [4]
orkut-links	Undirected	3,072,441	117,184,899	10	76,281	110s	[1] [14]

Table 4.2: Statistics of the graphs used in the experiments

4.2 Desired results

The goal of the experiments is to find whether the algorithm can make graph traversal on grids using Satin *efficient*. To test this, a number of different experiments have to be done. First of all, the worst-case scenario can be compared. This is when the target-node is not in the graph, and leads to the entire traversal of the graph. This is important because it gives a lower bound on the achievable performance. Second, a check on how effective the actual search is. This is done by executing 64 searches on the graph and taking the averages of these 64. The searches are all from a random root-node to a random target-node. These random nodes are checked to have at least a degree of one, meaning they have always an adjacent node. This test is modelled after the Graph500 benchmark [15]. The last experiment conducted is to find how efficient more than one graph traversal can be done in parallel. It is the same test as the previous one, except for 128 processes and they can run in parallel, if Satin decides to do so.

For all these tests, the different Satin versions are compared to see which additional features add to the model, and which do not. They are also compared to the sequential versions, to demonstrate the effectiveness of the parallelisation.

Comparisons will be done on a number of measurements. For the sequential versus parallel comparison these include the overhead with the sequential versions when run on a single node. Also the execution time and the average time per search are compared. These illustrate the efficiency of the parallelism, and whether or not it is a viable solution. To illustrate the differences among the parallel versions, these same measurements are useful, but they can be extended with some interesting Satin statistics. These include the number of spawns and syncs executed, the message sizes send, and the number of steal attempts and successes made by the work stealing scheduler. For the versions with Inlet/Aborts the number of aborts and aborted jobs are also interesting, and for the versions with Shared Objects the number of SO calls and the total size of these messages are meaningful. These can also be used to show whether or not the parallel versions are effective or not.

CHAPTER 5

Results

In this chapter the results of the conducted experiments are provided and clarified.

5.1 Overhead

To find whether a parallel application is efficient, first the overhead it yields when run on one processor should be investigated. Table 5.1 shows the search times for the sequential BFS and DFS versions, for both the full traversal of a graph and 64 random searches on the Livejournal graph. It also contains the search-times when run on just one processor for the four Satin search implementations, Search, Search_IA, Search_so and Search_ia_so. The overhead of all these implementations is show, with a relation to the two sequential versions. The lower the overhead the better. Table 5.2 has exactly the same layout, but the overhead is in percentages instead of seconds

	Full traversal			64 Random searches		
Application	**Time (s)**	**Overhead to BFS (s)**	**Overhead to DFS (s)**	**Avg time (s)**	**Overhead to BFS (s)**	**Overhead to DFS (s)**
Seq BFS	0.8940			0.6674		
Seq DFS	1.232			0.5944		
Search	1.243	0.3487	0.01125	0.6715	0.004124	0.07709
Search IA	1.066	0.1724	-0.1651	0.6511	-0.01627	0.05669
Search SO	1.208	0.3143	-0.02320	0.8385	0.1711	0.2440
Search IA SO	1.209	0.3149	-0.0226	0.7658	0.09841	0.1714

Table 5.1: Runtimes and overhead (in s) for all traversal versions, for a full traversal of the graph and for 64 random searches.

	Full traversal			64 Random searches		
Application	**Time (s)**	**Overhead to BFS**	**Overhead to DFS**	**Avg time (s)**	**Overhead to BFS**	**Overhead to DFS**
Seq BFS	0.8940			0.6674		
Seq DFS	1.232			0.5944		
Search	1.243	39.0%	0.9%	0.6715	0.6%	13.0%
Search IA	1.066	19.3%	-13.4%	0.6511	-2.4%	9.5%
Search SO	1.208	35.2%	-1.9%	0.8385	25.6%	41.1%
Search IA SO	1.209	35.2%	-1.8%	0.7658	14.7%	28.9%

Table 5.2: Runtimes and overhead (%) for all traversal versions, for a full traversal of the graph and for 64 random searches.

5.1.1 Evaluation

The overheads seems volatile, with some small and even negative overheads, but also some larger ones. Particularly the Search IA and to a lesser extend the normal Search do have a minimal amount of overhead with regards to the sequential versions. On the random searches the Search SO has notably the worst overhead. On the full traversal the BFS version is very quick, leading to a lot of overhead towards it from all Satin versions. It all seems to be logical, and the overhead on one node does not seem to be that bad.

5.2 Efficiency

Now, to see if the implementations can actually efficiently traverse graphs in parallel. Both the full traversal and the 64 random searches assignments are tested again, for all applications on the Livejournal graph. The diagrams below show the run-times and average search-times for these tests on the y-axis, with the number of cores on the x-axis.

5.2.1 One node, multiple processors

These tests are done for an ascending number of processors on one node. This shows the efficiency when there is shared memory between processes. The results can be found in the Figures 5.1 and 5.2. In these figures, a smaller execution time is good, so lower bars are better.

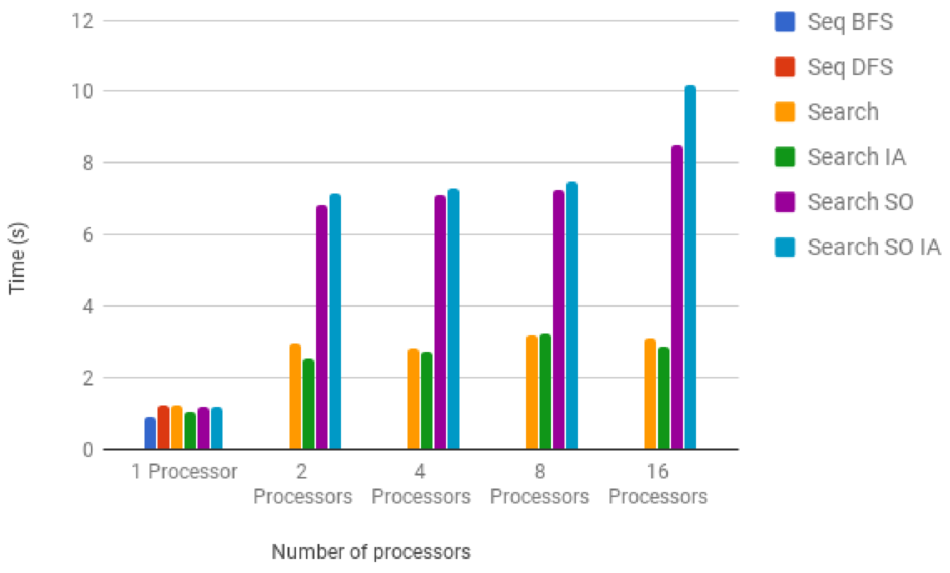

Figure 5.1: Execution times for the full traversal of a graph, on one node with an ascending number of processors.

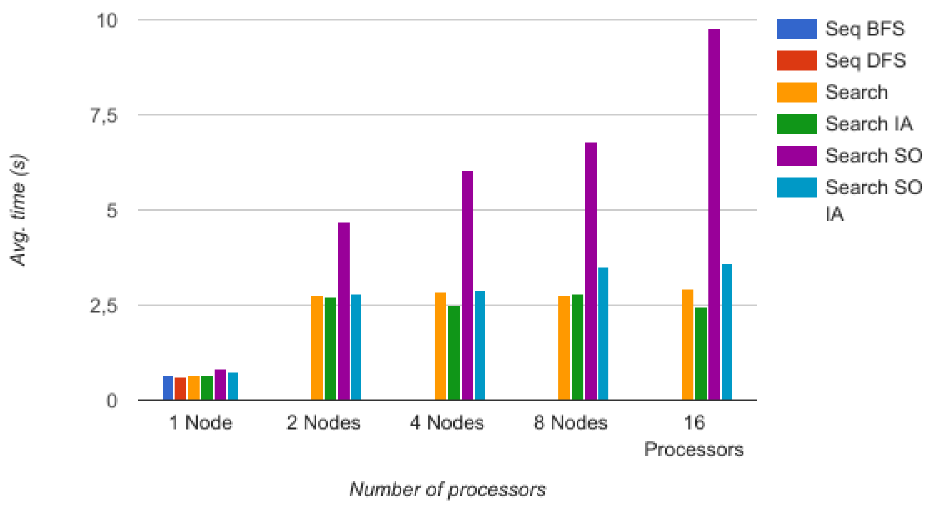

Figure 5.2: Execution times for 64 random searches on a graph, on one node with an ascending number of processors.

5.2.2 Multiple nodes, one processor

These tests are repeated for an ascending number of nodes, with only one processor per node used. This shows the efficiency when there is no shared memory between processes. The results can be found in the Figures 5.3 and 5.4. In these figures, a smaller execution time is good, so lower bars are better.

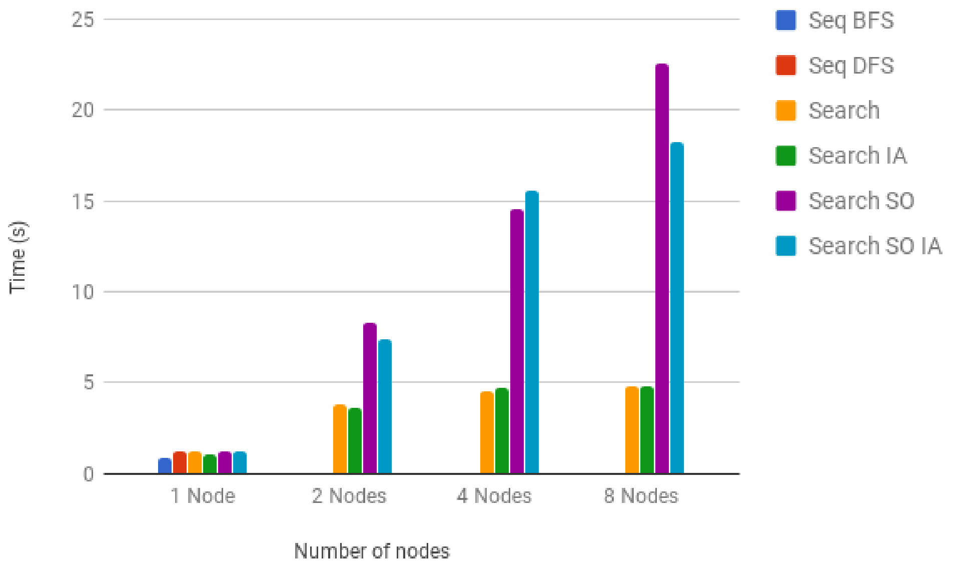

Figure 5.3: Execution times for the full traversal of a graph, on an ascending number of nodes with one processor per node.

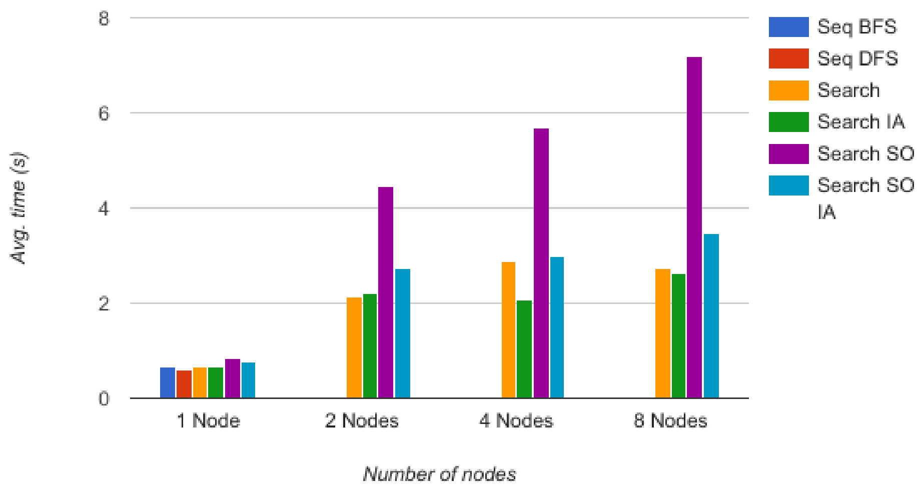

Figure 5.4: Execution times for 64 random searches on a graph, on an ascending number of nodes with one processor per node.

5.2.3 Evaluation

In these figures all parallel applications have a speed-down instead of a speed-up. This is in comparison to both the sequential versions and the parallel versions when ran on just one core. The addition of more and more cores does not seem to have a positive effect. For the Search and Search IA this does not seem to have an effect at all. For both versions with shared objects it even has a negative effect, resulting in even worse times. For the random searches the aborts save the Search IA SO version from this fate, however.

The tests on one node with multiple processors seem to give results that are a bit better, when compared to the tests on multiple nodes with one core. This seems reasonable, because of the shared memory on one node. However it also applies to the versions with shared-objects. This is the exact opposite of the expectation, because the versions with shared objects were specifically designed to improve the results when there is no shared memory. For this application, the overhead of broadcasting the write methods to all machines is too high to achieve speedups. This means that Shared Objects do not produce positive results, and show no sign of advantages. Inlet/Aborts do a bit of positive work when there is a result that can be found (like in the 64 random tests), and take a lot of the negative effects of Shared Objects away.

5.3 Search overhead

The speed-down of the parallel versions warrants further investigation. It could be the result of search overhead: due to the speculative nature of the search, the parallel version may visit more nodes than the sequential code. Visiting more nodes leads to a slower application. To show whether or not this is the main reason, the number of travelled nodes can be found for all versions for 64 random traversals on one node with an ascending number of processors.

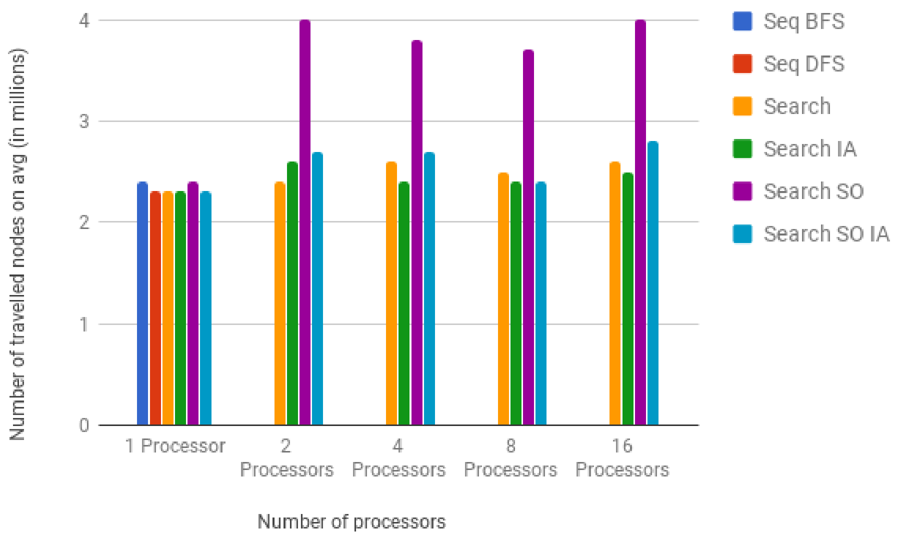

Figure 5.5: Avg number of travelled nodes for 64 random searches on one node with an ascending number of processors.

5.3.1 Evaluation

Most Satin versions have numbers of travelled nodes similar to those of the sequential versions. The Inlet/Aborts seem to have but a small effect, except when used in combination with shared objects. The shared objects version (without aborts) traverses a lot more nodes, nearly all the nodes in the graph on average. This results in a lot of search overhead.

The difference in the number of travelled nodes between parallel and sequential versions is quite small, especially when compared to the difference in execution times. This points to the search overhead not being the main reason for the speed-down. However it still may have some effect. For the version with shared objects it could be one of the larger contributing factors, as the difference in traversed nodes is more substantial. However, the number of travelled nodes only grows from one to two processors, and not from two to four or eight or sixteen. The execution times did show this behaviour, which points at search overhead also not being the largest speed-down factor in this case.

5.4 Efficiency problem: statistics

To find another explanation why the parallel applications do not gain any parallel efficiency and adding more and more cores even results in speed-down, some statistics produced by Satin are analysed. In the tables below some of these statistics are presented.

In the Satin statistics of the full-traversal on one node with 16 cores (Table 5.3), there is no observable difference between Search and Search IA. This can be attributed to the lack of aborts because in a full traversal these will not be executed. The overhead the inlets have is negligible. Both have more than enough spawns, meaning there are enough threats that can be stolen. This is attempted not that many times, and even fewer attempts succeed. This leads to nearly no parallel work.

The two versions with Shared Objects differ not that much from each other. Both have a lot more steal attempts than the other two versions, but relatively even fewer successful steals. They both use a lot more data and send some of it using the Shared Objects. This creates more work, with no advantages.

When looking at the Satin statistics of the random searches on one node with 16 cores (Table 5.4), all numbers have the be divided by 64, when comparing them to the previous table. These are the totals for 64 searches, not the averages.

Application	**Spawns**	**Syncs**	**Attempted steals**	**Steals**	**Messages**	**Total data**	**SO calls**	**SO data size**
Search	1203	1178	48	1	117	329 MB		
Search IA	1203	1178	49	1	127	320 MB		
Search SO	1901	1868	303	3	633	945 MB	7818	24 MB
Search IA SO	1890	1861	818	3	1666	945 MB	7830	24 MB

Table 5.3: Some Satin statistics for the full traversal, on one node with 16 processors.

These statistics show again a large amount of spawns, this time with a lot more steal attempts, but a very small amount of actual steals. The aborts do their work aborting a lot of processes, which leads to fewer syncs. In this test the Search SO does by far the most work, with a lot more spawns, syncs, attempted steals and SO data send. This probably is the reason why it is so much worse in the figures from Section 5.2. The Search IA SO is a lot more like the two versions without Shared Objects than the one with them. This can be attributed to the Inlet/Abort mechanism, which for these tests is effective.

Application	**Spawns**	**Syncs**	**Attempted steals**	**Steals**	**Aborts**	**Aborted jobs**	**SO calls**	**SO data size**
Search	146511	145094	20344	62				
Search IA	101573	12943	7079	62	117	95090		
Search SO	204599	202381	356953	175			546448	1,4 GB
Search IA SO	99613	18055	39678	81	123	87690	250374	949 MB

Table 5.4: Some Satin statistics for the 64 random searches, on one node with 16 processors.

5.4.1 Evaluation

It would seem that the number of parallel spawns is on the high side, but the amount of actual parallel work is negligible. This accounts for the speed-down perceived in Section 5.2. All signs point to the work that should be done by the Satin implementations in parallel, as not actually being done in parallel. This also explains why there is a speed-down from 1 core to 2, but no speed-down when going from 2 to 4 or 8 or 16. The exception here is the implementations with shared objects, because with each new core there has to be more communication. This means that the one working processor does nearly all the work itself, but is constantly slowed down by other processors asking for work, and for the shared objects versions, by having to broadcast to all other processors all the time.

It would seem that each job is too small to effectively give over, because the visiting of one node is nothing more than some memory look-ups. One processor can do it so quickly that the parallel actions are not needed and even bad for performance. From these observations it looks like the only thing these applications create is parallel overhead, and nearly no parallel work at all.

5.5 Multiple parallel traversals

As shown above, the parallel graph traversal implementations do not produce any speed-up in the tests, but instead a lot of speed-down. This is very unfortunate, because it means that the applications are not efficient and scalable. However, they still could be useful when more than one graph traversal has to be done simultaneous. The sequential versions would have to be run one after another, so this could be done more efficiently in parallel, in theory. This would give the Satin version a bit of an edge over the sequential versions in this regard. It would mean that the Satin versions can utilise multiple cores to do multiple traversals, and with more cores, solve them more quickly.

To test this, the previously mentioned parallel test was designed. Hereby 128 random searches are done, with Satin allowed to run them in parallel. This test was conducted for the Search and

Search IA versions and on an ascending number of total cores, eventually using 8 nodes with node 16 cores per node. The resulting execution times are shown in Figure 5.6. To illustrate the effectiveness of the Satin versions in utilising a growing number of cores, the speedup is shown in Figure 5.7, relative to a linear speedup.

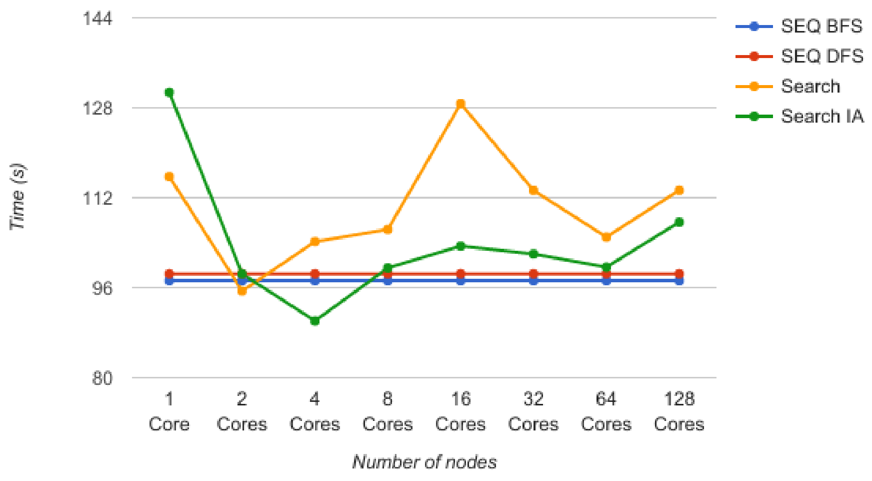

Figure 5.6: Execution times for 128 parallel random traversals of a graph, on an ascending number of cores.

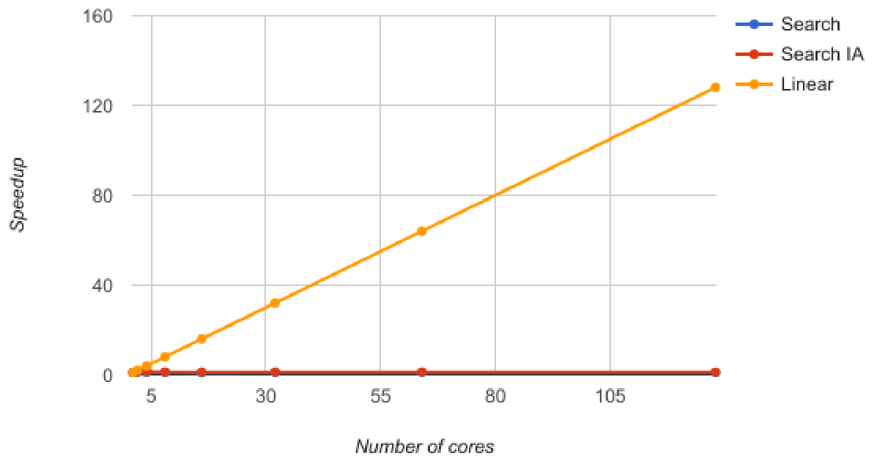

Figure 5.7: Speedup of the 128 parallel random traversal, on an ascending number of cores, relative to linear.

These tests were run on the orkut-links graph, while all previous tests were run on the Livejournal graph. This was to show that the applications work on more than one graph, and that the lack of efficiency is not because of a specific graph.

5.5.1 Evaluation

As seen in Figure 5.6 both Satin versions have some overhead as opposed to the sequential versions when run on one core. When run on a few, two or four, cores both behave better than the sequential versions. However, instead of scaling further when adding more cores, the implementations instead create more overhead and perform even worse than the sequential versions on more cores. This trend is interesting, because the addition of a few cores show that in principle the versions allow speedup when running a lot of traversals in parallel, but then when more are added, this trend stops and the performance gets worse again. This disappoints, because this appeared to be the one way for the Satin implementation to gain some form of scalability.

Figure 5.7 illustrates just how non-scalable the applications are. The speedup does not even show in comparison to a linear speedup, which is the best-case scenario for speedup. This means that the applications can also not be efficiently used for multiple graph traversals.

CHAPTER 6

Conclusions

The goal of this thesis was to make the scope of possibilities of Satin clearer. The way to do this has been to investigate whether or not the by Berkeley defined *Graph Traversal* Dwarf can be written in a Divide-and-Conquer manner in Satin. To make this possible the traversal of a graph was defined as visiting a node, and from there dividing the graph in sub-graphs. These sub-graphs are not defined specifically, but only by their root node, one of the nodes adjacent to the current node. For all these new sub-graphs the same method is used recursively, effectively traversing ever smaller sub-graphs in parallel. To avoid overlap a list of all visited nodes is kept, to effectively divide the sub-graphs. This developed method is quite elegant, and the best way we know of to write graph traversal in a Divide-and-Conquer fashion.

Implementing this algorithm in Satin was very doable. Satin is specifically designed for Divide-and-Conquer applications, and it adds an abstraction layer over the grid to allow for a pleasant parallel programming experience. The implementation of this algorithm in Satin led to four different versions, each making use of different features of Satin to strive for an application that is as efficient as possible. These versions added speculative searches and shared visited lists to the implementations. These aim for better than worst-case results, and for better use of parallel nodes without shared memory.

The tests of these implementations in Satin showed that they did not function in the way that was hoped. When compared to sequential graph traversal algorithms like BFS and DFS the Satin versions where a lot slower. Adding more cores to them did not resolve this. Especially the versions with shared objects seemed to experience a case of speed-down, when the number of cores ascended. The parallel implementations do not take any advantage of the parallel structure, and instead fabricate parallel overhead due to the broadcasting of the visited nodes list. The speculative parallelism mechanism removed some of the overhead, by aborting when the result was found, but not nearly enough to make the applications viable.

The reason for the lack in speedup by the Satin implementations is that it does nearly nothing actually in parallel. There are enough jobs, but idle processors do not succeed in stealing them. The conclusion has to be drawn that the jobs spawned can not be done in parallel, before the sequential version does them. This can be attributed to each job being too small, because each just consists only of a few memory look-ups. Therefore, it results in nearly all being done sequentially. The only result of Satin's attempts at parallelisation is overhead. This means that the applications created by implementing the Divide-and-Conquer graph traversal algorithm are not efficient and are in no way better than the sequential versions, except when computing multiple traversals in parallel on a very small number of machines (i.e., 4 nodes), as opposed to sequential.

A note on the disappointing results is that, because of the short time-frame and subsequently small scope of this research, it has been contained to relatively small graphs. The load time already is dominant, but larger graphs could still scale better than the tested graphs. However, this could not be tested in this thesis. Also, graph traversal is notoriously difficult to parallelize efficiently and difficult to scale [13]. The bad results are not a real shock.

Now we can give answers to the research questions drafted in the introduction of this thesis.

The first one was:

Can, with the use of Satin, a Divide-and-Conquer application be implemented for the Graph Traversal Berkeley Dwarf? (3)

The answer is that it is possible to create a Divide-and-Conquer graph traversal algorithm, and implement this in Satin. The designed algorithm was a good representation of graph traversal in a Divide-and-Conquer style. It could, without any problems, be implemented into Satin code in an elegant manner. The second question was:

Does this application run efficiently on a computational grid in a scalable way? (4)

This has been shown not to be the case, at least for the graph problems we could investigate. All four the applications did not function sufficiently on a grid, and were neither more efficient than sequential implementations nor demonstrated speedup when run on more and more nodes.

The conclusion is that the graph traversal Dwarf can be made in Satin, but it will not work efficiently or scalable.

6.1 Future work

The research to the bounds of Satin's possibilities can be continued in the future. There are still a few Dwarfs not created in Satin, or not researched enough. These include, as far as we could find, the unstructured grids, dynamic programming and finite state machines. There also has been no overarching research into what Satin can do for each specific Dwarf. It would be interesting to list all dwarfs and their ability to be written in a Divide-and-Conquer fashion and implemented in Satin.

Another interesting direction future researchers can take is looking into non Divide-and-Conquer algorithms for graph traversal in Satin. It has been shown that a truly Divide-and-Conquer application does not work efficiently. However, there are enough graph traversal algorithms that are not strictly Divide-and-Conquer [6, 8, 10]. Some do however, use a form of Divide-and-Conquer to build the to-visit-nodes data-structure, like the queue or stack in BFS and DFS. It is interesting to see if graph traversal can be efficiently implemented in Satin, by using a non or semi Divide-and-Conquer method (e.g. a master-worker approach that is expressible in Satin.). However, an attempt at such an implementation was already made during this research, and it did also have disappointing results. This is the reason why it was dropped from the scope of this thesis altogether.

The last interesting avenue to venture on is more extensive testing of the current applications. Larger graphs could prove to at least produce more speedup on multiple traversals. We could also do tests on the perfect time to switch from spawning parallel jobs to switching to sequential traversal. Spawning more or less jobs could offer some improvements. However, future researchers into these subjects should not set their expectations too high.

Bibliography

[1] Orkut network dataset – KONECT, May 2017.

[2] Krste Asanovic, Rastislav Bodik, Bryan C Catanzaro, Joseph J Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William L Plishker, John Shalf, Samuel W Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

[5] Henri E Bal, Dick Epema, Cees de Laat, Rob V van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[6] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[7] Miguel L Bote-Lorenzo, Yannis A Dimitriadis, and Eduardo Gómez-Sánchez. Grid characteristics and uses: a grid definition. In *Grid Computing*, pages 291–298. Springer, 2004.

[8] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[9] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[10] Charles E Leiserson and Tao B Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 303–314. ACM, 2010.

[11] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, May 2017.

[12] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[13] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[14] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.*, 2007.

[15] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[16] Rob V van Nieuwpoort. *Efficient Java-Centric Grid Computing.* Rob van Nieuwpoort, 2003.

[17] Rob V van Nieuwpoort, Thilo Kielmann, and Henri E Bal. Satin: Efficient parallel divide-and-conquer in java. In *Euro-Par 2000 Parallel Processing*, pages 690–699. Springer, 2000.

[18] Rob V Van Nieuwpoort, Thilo Kielmann, and Henri E Bal. Efficient load balancing for wide-area divide-and-conquer applications. *ACM SIGPLAN Notices*, 36(7):34–43, 2001.

[19] Rob V Van Nieuwpoort, Gosia Wrzesińska, Ceriel JH Jacobs, and Henri E Bal. Satin: A high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):9, 2010.