Design, implementation and analysis of algorithms on multi-core systems

Peter Wind & Jonas Hansen

Kongens Lyngby 2008
IMM-B.Sc

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

It was the main task of this thesis to analyze, design and implement a parallel version of a classical sequential algorithms. We have in this thesis selected to parallelize the classical sequential algorithm *Push Relabel*, using the general parallelization procedure described in [11]. Several different implementations, with or without heuristics, has been considered and tested against each other to get the most efficient algorithm for solving the maximum flow problem.

The Push Relabel algorithm has been discussed and parallelized in previous studies (e.g. [2] and [3]), but they utilize the programming language C to implement the actual algorithms. Our thesis differs from previous studies by implementing the algorithms using the widely used object-oriented programming language Java. The main focus of the thesis is therefore not only to develop a parallel implementation of the Push Relabel algorithm but also to study the affect of using the parallelization tools available in Java. Our implementation also differs from earlier studies by being optimized and tested on public available multi-core computers instead of high-end supercomputers.

All developed algorithms has been tested against reference implementations of other well known sequential algorithms, and for testing-purposes, three graph generators has been developed and implemented. The graph generators are inspired by graphs used in previous studies [10] and graphs used in the DIMACS implementation challenges [7].

The developed algorithms performs well in practice, and results in running time decreases proportional to the number of CPU cores. We have, however, discovered that the performance of the algorithms varies a lot depending on the operating system and Java version.

Resumé

Hovedvægten i denne opgave er blevet lagt på, at analysere, designe om implementere en parallel version af en klassisk sekventiel algoritme. Vi har i opgaven valgt at parallisere Maxiumum Flow algoritmen *Push Relabel*, ved at bruge den generelle paralleliserings-metode beskrevet i [11]. Forskellige implementationer, med og uden heuristik, er blevet overvejet og testet op imod hinanden for at finde den mest effektive algoritme for at løse Maximum Flow problemet.

Push Relabel algoritmen er blevet beskrevet og paralleliseret i tidligere undersøgelser (f.eks. [2] and [3]), men disse undersøgelser bruger programmeringssproget C til at implementere algoritmerne. Vores opgave afviger fra tidligere undersøgelser ved at implementere algoritmerne ved brug af det objekt orienterede sprog Java. Hovedvægten i opgaven er derfor ikke blot at udvikle en parallel implementation af Push Relabel-algoritmen men også at undersøge hvilken effekt det giver at bruge de indbyggede paralleliseringsværktøjer i Java. Vores opgave afviger yderligere fra tidligere undersøgelser ved at henvende sig til private flerkernede computere i stedet for avancerede supercomputerede.

Alle de udviklede algoritmer er blevet testet op imod en reference-implementation af andre kendte sekventielle algoritmer og tre graf-generatorer er blevet implementeret for at generere testgrafer. Disse graf-generatorer er inspireret af grafer som er blevet brugt i tidligere undersøgelser [10] og grafer som er blevet brugt i DIMACS [7].

De udviklede algoritmer klarer sig godt i praksis og køretiderne viser sig at aftage proportionelt med antallet af CPU-kerne. Derudover har vi opdaget at algoritmernes ydeevne afhænger meget af det operativsystem som bruges og af den Java-version som bruges.

Preface

This thesis was prepared at the institute of Informatics and Mathematical Modelling, at the Technical University of Denmark in the period of February through June 2008. as part of the requirements for acquiring the B.Sc. degree in engineering.

The thesis deals with different aspects of parallelizing algorithms using the widely used programming language Java, and the main focus is on parallelizing the classical sequential algorithm *Push Relabel*.

We would like to thank our supervisor Asssociate Professor Paul Fischer for agreeing to supervise this project. Paul has great knowledge of algorithmic development and has guided us in choosing the right approaches and analyses to be able to achieve our results. We also thank Paul for his great help in reading and commenting the drafts of the thesis, which has been a great help in improving the final version.

Lyngby, Juni 2008

Jonas Hansen
Peter Wind

Contents

Chapter 1

Introduction

1.1 Motivation and purpose

Until a few years ago, designing parallel algorithms was the task of only a small part of developers, in particular researchers. This was mainly because multi-core computers were not available to the general public, but also because efficient parallel programming tools were not implemented in the widely used object-oriented programming languages. This, however, has changed during the last few years as multi-core processors has become more mainstream. Software and algorithms now needs to be parallelized to take advantage of multiple CPU cores, as this have become the means to improve performance while the actual processor clock frequency is no longer increasing.

The main purpose of this thesis is to parallelize the classical sequential algorithm *Push Relabel*, which is an efficient algorithm for finding the maximum flow in a flow network. Other classical algorithms could have been chosen, but our initial research combined with previous studies [2], showed that the Push Relabel algorithm has a good parallelization potential.

An important part of the thesis is the actual parallelization process and much work has been put into describing this in details. The process begins with a analysis followed by a design phase and the actual implementation. The process

used, is inspired by the process used in [11] and is similar to the procedures used in other known literature.

The analysis phase is very thorough and contains several steps that divides the algorithm into smaller tasks and maps these to the different CPU cores and thereby enables concurrent execution of tasks. The analysis also discusses and plans the synchronization between the smaller tasks as well as the access control to shared data structures.

Following the analysis, the developed algorithms are implemented using the object-oriented programming language *Java*, and their performance is compared to reference implementations of other maximum flow algorithms. One important aspect in which our thesis differs from previous studies is by the use of an object-oriented language instead of the commonly used language *C*, and it is interesting to see what impact this will have on the performance as well as the development process.

The thesis does not require any previous knowledge on neither the maximum flow problem, the push relabel algorithm and/or the parallelization process as all relevant theory is introduces during the thesis.

1.2 Structure and overview

In Chapter 2, we introduce the main concepts of concurrency and parallelization. This includes an introduction to basic concurrency theory which introduces the concepts of *memory models*, properties of concurrent programs and how concurrent programs can be modelled. This is followed by a thorough explanation of the parallelization procedure used in the thesis.

The chapter proceed by introducing the classical synchronization primitives and explains how they are available in Java.

In Chapter 3, we introduce all relevant theory and definitions of the maximum flow problem. This is followed by an introduction to the different maximum flow algorithms used in the thesis.

Chapter 4 describes in details the development of the parallelized version of the Push Relabel algorithm. The chapter contains an analysis following the procedure described in Chapter 2, which leads to a more detailed design of the algorithms. Finally, the chapter contains implementation specific details on each of the implemented algorithms, as well as implementation details on the

underlying graph framework.

To verify and test the developed algorithms, Chapter 5 contains the results from the actual execution of the developed algorithms. The results are compared to reference implementations of other sequential algorithms, and the performance benefits gained from parallelizing the Push Relabel algorithm is discussed.

Finally, our achievements, conclusion and suggestions for future work is presented i Chapter 6.

Chapter 2

Concurrency and parallelization

In this chapter the theory needed to parallelize a sequential algorithms is presented. The chapter contains an introduction to basic concurrency theory as well as a description of the parallelization procedure used in the thesis. This is followed by an introduction to the classical synchronization primitives and how these are available trough Java.

2.1 Concurrency Theory

A possible definition of concurrency in computer science is a property of a system in which several computational processes or tasks are executing at the same time, possibly interacting with each other. These tasks may be implemented as separate programs or threads within a single program.

In this thesis the term *task* is used instead of the more classical term *process* to denote the computational process which takes place. This is done to clearly distinguish it from the term *processor* which denotes a CPU core.

On modern computers all tasks may execute in parallel (*true parallelism*), how-

ever it is most often the case that the number of of processors is less than the number of tasks. As a result of this, parallelism is often obtained by the method of *time-slicing*, where the operating system controls the scheduling of tasks between the available processors. A consequence of this is that the actual execution time of a single task is unknown [16, cha 2.3.2].

As stated, tasks in a concurrent system can interact with each other while they are executing. This, combined with the time-slicing method, results in an highly unpredictable order of execution between the different tasks, and it is often up to the developer to use different techniques to control this execution order. The interaction between tasks is described more formally later in this section.

In general when two or more tasks are interacting with each other, they are said to be "communicating". Communication is however a very vague term and in concurrency theory two different communication strategies has been defined more formally:

- **Shared memory** In a shared memory system, the different processors and memory modules are interconnected, and tasks can therefore share the same memory locations. The communication between tasks is carried out by altering some shared variable which then is visible to other tasks. Because different tasks, or threads as they are most often called in this context, are sharing the same memory locations, consistency issues can occur. These issues can be handled by utilizing some kind of locking mechanism (synchronization primitives), which controls the access to the memory locations and preserved the program invariants. The main problem with the locking solution is, that it is up to the programmer to ensure that the consistency issues are taken care of, which leaves room for errors. We use the synchronization and locking primitives of the JAVA language to the control the consistency issues and they will be introduces in Section 2.4.
- **Distributed memory** In a distributed memory system all processors are still interconnected, but each processor now has its own memory module. This can be achieved within a single computer (a multi-computer), but the distributed memory model is also suited for connections across a network (network system). The communication between the processors is in a distributed system handled by message parsing. The concept of message parsing means that tasks communicate by exchanging messages. The exchange of messages may be done both synchronously (blocking) or asynchronously (non-blocking). There exists a number of models for modeling the behavior of systems using message parsing, one of them ,*rendezvous*, may be used to model blocking implementations, in which the sender blocks until the message is received. Message parsing systems are often

easier to reason about than shared memory systems, and they are often very scalable in size. The downside of distributes memory systems is, that they have a larger communications overhead than shared memory systems and thereby doesn't utilize the processing power as optimal.

An example of a very large distributed memory network system is the SETI@HOME project in which almost 3 million home computers work together to detect intelligent life outside of Earth. [19].

2.1.1 Properties of concurrent programs

"Concurrent programs differs from sequential ones by being reactive, i.e. expressing an (often infinite) activity rather than the computation of a final result."[16, cha 3.1]. Because of this the focus is on behavior, rather than on input/output relations. Properties of a concurrent program can be divided into two informal categories:

- **Safety properties** A safety property is a property stating that *"something bad does never happen"*. A typical example of a safety property is that a program is *Deadlock* free. A deadlock is a situation wherein two or more competing tasks are waiting for the other to finish, and thus neither ever does. The safety properties is therefore usable in securing and actually proving that a program always does what it is supposed to do.
- **Liveness properties** A liveness property is a property stating that a program *"eventually will make progress"*. An important liveness property is that a program should always terminate at some point.

By parring safety and liveness properties it can be implied that a program *"eventually does something good"*. More information on safety and liveness properties can be found in [16].

2.1.2 Modeling concurrent programs

To model and prove properties of concurrent programs different models have been developed. In this section two useful models for modeling concurrent behavior is introduced, namely Petri Nets and the interleaving model. These models also serves as useful models for introducing several of the most important terms of concurrent programs. Of course several other models exists, but they are out of the scope of this thesis.

Petri Nets Formally a Petri Net is a bipartite, directed graph that can be used to describe the dynamic behavior of a system, especially the concurrency and synchronization aspects [16]. A Petri Net graph consists of two kinds of nodes/vertices. *Places* which represents states of a system and *transitions* which represents activities in the system. Places are drawn as circles and transitions are drawn as bars or boxes. Places and transitions are connected via *arcs* that indicate the dependencies between states and activities. Furthermore, places in a Petri Net may contain zero or more *tokens*, and a distribution of tokens across a Petri Net is called a *marking*.

As stated, A Petri Net can describe how a system behaves dynamically over time. This is done by showing how the different states and activities of a system influences each other. The change of state is modelled by letting an initial marking evolve trough "firing" of transitions, meaning that a transition consumes a specified number of tokens from its input places (one token per incoming arc), performs some processing task, and produces a specified number of tokens into each of the output places (one token per outgoing arc). A transition is set to be enabled, if each of its input places contains at least one token, and a firing of a transition, which is performed in a single, non-preemptible step, can only take place if the transition is enabled. Enabled transitions can fire at any time, and happens in a non-deterministic manner, meaning that multiple transitions may fire simultaneously exactly like the execution of concurrent threads.

The mathematical representation of a Petri Net is a tuple $< P, T, F >$ where

P: Is a set of vertices called *places*

T: Is a set of vertices ($P \cap T = \emptyset$) called *transistions*

F: Is a flow relation consisting of a set of arcs, where $F \subseteq (P \times T) \cup (T \times P)$

The state of a Petri Net is represented by the the current marking of the net represented as a vector M, where the initial state is given by the marking M_0. If an enabled transition is fired, the marking M evolves into M', which is denoted by $M_1 \xrightarrow{t} M_2$. Figure 2.1 shows an example of a Petri Net where the current state is given by the vector $M^t = (1, 0, 1, 0)$.

As already mentioned, a transition t can only fire if there exists enough tokens in the input places. This can be seen as a condition of an action and can be used to model many synchronization mechanisms. A mechanism used a lot throughout this thesis is the concept of *Mutual Exclusion*. Mutual exclusion is when certain actions (or sequences of actions) are not allowed to be executed at the same time. Usually this is to prevent the two or more threads access the

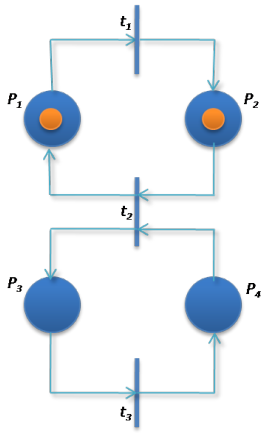

Figure 2.1: Example of a petri net

same resource or data-structure at a time. Mutual exclusion is easily modelled using Petri Nets by using an auxiliary place with a single token that represents the "right" to execute. By letting the transitions, which should not be executed at the same time, claim this single token, the firing rules prevents that more than one of the transitions to be fired at the same time. The auxiliary place with a single token is in most programming languages known as a *lock*. The lock is *acquired* before critical code is executed and *released* when done. A Petri Net with mutual exclusion can be seen in Figure 2.2.

The Interleaving Model If one is not interested in the properties related to whether actions are actually performed in parallel, but solely in the properties related to the sequence of states a task will go trough, one may use the *interleaving model* of the task behavior. For any given task the execution flow can be modelled by a finite or infinite sequence of the form:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$$

where s_0 is the initial state of the task and the a_i's are actions/transitions. The execution of action a_0 will change the state of the task from s_0 to s_1, etc. If the actions of a program always executes without any overlapping in time, the task is said to be *sequential*. In the interleaving model, a concurrent program is composed of two or more sequential tasks overlapping in time. Because of the overlap in time we need to introduce the concept of *atomic actions*. An atomic action is an indivisible action, meaning that no other tasks are able to detect

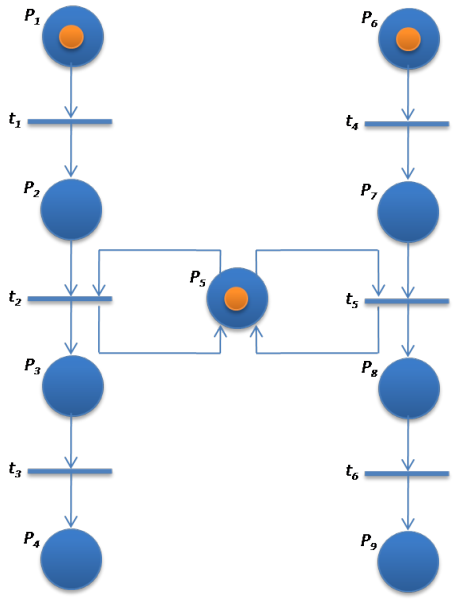

Figure 2.2: Example of a Petri-net representing mutual exclusion

any intermediary states during the execution of the action.

With the introduction of the atomic action, the question is which executions are possible for a program. The execution speed of each action and thereby also each task is unknown, meaning that the execution of a program is given by all possible interleaving of all possible sequences of actions of the tasks. The total number of interleavings of a program can be computed as seen in equation 2.1.

$$\frac{(i_1 \cdot i_2 \cdot i_3 \cdot \ldots \cdot i_n)!}{i_1! \cdot i_2! \cdot i_3! \cdot \ldots \cdot i_n!} \tag{2.1}$$

where i_n denotes the number of atomic actions in the *n'th* task. As this number rapidly increases, it is hard to predict all possible outcomes of a program. This is exactly why it is important to control access to certain areas of code and thereby limiting the amount of possible interleavings.

2.2 Parallelization

The main topic of this thesis is the design of algorithms for modern-day multi-core systems. As multi-core processors becomes more mainstream, software and algorithms needs to be parallelized to take advantage of multiple cores. Until a few years ago most algorithm designers were designing sequential algorithms which is more or less a series of steps for solving a given problem using a serial computer (with a single CPU core). Likewise, a parallel algorithm could be described as the steps involved in solving a given problem on a parallel computer. This however, is an oversimplification, as the development of parallel algorithms involves much more than just describing the steps of the computation. At the very least, a parallel algorithm has the added dimension of concurrency and the algorithm designer must specify the sets of steps that can be executed simultaneously. Simultaneously or concurrent execution is essential for obtaining any performance benefits from the use of a multi-core computer. In practice, designing a nontrivial concurrent algorithm may include some or all of the following steps [11, chapter 3]:

- **Decomposition**: Identifying parts of problem that can be performed concurrently.
- **Mapping**: Mapping the sub-problems to the available threads.
- **Access control** (synchronization): Controlling the access to data shared by multiple threads using synchronization.

The above items is the parallelization technique used in this thesis, but is just one out of many possible ways of parallelize an algorithm. A parallelization technique should not be seen as an exact science and varies depending on the actual problem. Usually the steps also varies depending on both the underlying architecture and the programming paradigm (programming language). It is also important to note that not all sequential algorithms are suited for parallelization, often because each subproblem relies on results from the previous subproblem. This creates a large communication overhead and the benefits of parallel computing is lost.

A sequential algorithm's potential benefit by being converted into a parallel algorithm is theoretically defined by Amdahl's Law[1]. The law predicts that if P is the portion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the portion that cannot be parallelized (remains sequential), then the maximum speedup that can be achieved by using N pro-

cessors is given by equation 2.2.

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{2.2}$$

In the following subsections, the parallelization steps are discussed in further details, followed by a discussion of the challenges met when designing concurrent algorithms. Together with each step is a small example showing the steps in practice.

2.2.1 Decomposition

As mentioned, one of the most important steps needed to solve a problem in parallel is to decompose or split a problem into a set of subtasks which can be executed concurrently. Tasks can be of arbitrary size defined by the programmer, and not all tasks need to be of the same size.

In an ideal case, all tasks should be independent from each other, but most often it is the case that each task depends on the result of other tasks. It is therefore important to resolve how each task is dependent on other tasks. This could be done by using a *task-dependency graph*, which is a directed acyclic graph where vertices represents tasks and the directed edges represents dependencies between tasks. The task corresponding to a node can be executed when all tasks connected to this node by incoming edges have completed. An arbitrary example of task-dependency graph can be seen in figure 2.3. The figure, among other things, shows that task 7 can only be executed when all other tasks have been completed.

Another important problem to have in mind when decomposing a problem is the *granularity* of the decomposition. The granularity refers to the size of the subtasks compared to the main problem. If the decomposition consists of a large number of small tasks it is called *fine-grained*, and if the decomposition consists of fewer larger tasks it is called *coarse-grained*. The granularity can have a big impact on the performance of an algorithm in two ways. If the decomposition is too fine-grained, each task is not very computation heavy and much time is used on communication between the different tasks. If the decomposition on the other hand is too coarse-grained, not enough operations can be done concurrently and some processors may be idle for some time. The key is to find the correct task-size so that each processor is always occupied and only minimal time is spend on communication. This optimal task-size can be hard to figure out and is most often achieved trough trial-and-error. An example of

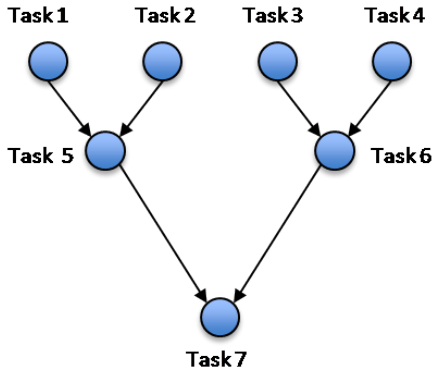

Figure 2.3: Task-dependency graph

decomposition can be seen in figure 2.4, which is about the multiplication of a dense $n \times n$ matrix A with a vector b to yield another vector y. The i'th element $y[i]$ of the product vector is the dot-product of the i'th row of A with the input vector b. The figure illustrates a decomposition into 4 tasks, each responsible of a quarter of the complete computation. This decomposition limits the number of concurrent tasks to 4 as this is the maximum number of independent tasks.

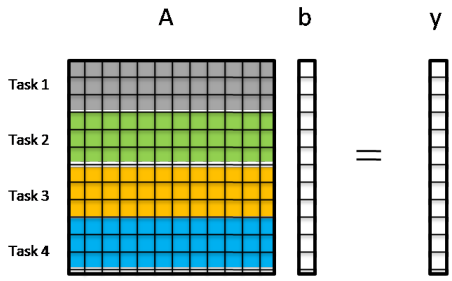

Figure 2.4: Decomposition of a matrix multiplication

There exists a number of different approaches to decompose computational problems and this thesis will only introduce two of them - *recursive decomposition* and *data decomposition*. The decompositions mentioned should however be a good starting point for many computational problems and they can often lead effective decompositions.

Data decomposition In the data decomposition approach, the data associated with the problem is decomposed instead of the problem itself. If possible

the data is divided into smaller pieces of approximately equal size. Next the computation is decomposed, which is typically done by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each comprised of some data and a set of operations on that data. In an ideal case, each task is completely independent from other tasks, but often operations may require data from several other tasks. In these cases communication between tasks is required.

The data decomposed may be either the input to the program, the output of the program or some intermediate data maintained by the program. Several different partitions are often possible based on the data structure, and careful analysis is required to determine the most effective. An effective approach is to focus first on the largest data structure or on the data structure which is accessed most frequently, but it is important to note that a computation could contain more than one phase which requires different decompositions.

The example shown in figure 2.4 utilizes data decomposition in which the input data (The matrix A and the vector b) is decomposed into smaller parts.

Recursive decomposition Recursive decomposition represents a different and complementary way of thinking about problems. In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. This method is useful if the computation can be divided into disjoint tasks which often is the case for algorithm usually solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent subproblems. Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results. The divide-and-conquer strategy results in natural concurrency, as the different subproblems can be solved concurrently. If the subproblems are not completely disjoint (as they are in divide-and-conquer algorithms), the recursive decomposition technique can still be used. However, this requires some communication between the different tasks, which in many cases isn't trivial. If the communication overhead is too large the data decomposition method should be considered instead.

2.2.2 Mapping

When the problem has been decomposed into smaller tasks, each of these tasks should be run on physical processors. This is done by assigning tasks to one or more threads, which is also known as *mapping*. Normally there are more tasks than threads and physical processors, and it is therefore necessary to distribute

the tasks evenly across all threads. To benefit from having more than one physical processor and thereby minimizing the execution, several tasks should run concurrently. Two strategies can be used to achieve this goal:

- Tasks which can be executed concurrently should be mapped to different threads, so as to enhance concurrency.
- Tasks that communicate frequently should be mapped to the same thread, as to minimize the overhead associated with communication and to increase locality.

These two strategies can and will often collide and it is therefore necessary to make some kind of tradeoff. In addition, the number of processors are often limited, which restricts the number of threads which can be run concurrently.

The above mentioned approach only works well if the number of tasks are known in advance. If they varies dynamically during the program execution, one needs to use an algorithm which can distribute new tasks to different processors. This is known as *load-balancing* and several effective algorithms or strategies has been developed to solve this. Two of these strategies is used in this thesis:

- **Manager/Worker Scheme**[1] - The manager/worker scheme consists of a one *Manager* (often represented as a single thread) and several *Workers* (often represented as several threads). The workers repeatedly request and process tasks from the manager which maintains a pool of available tasks. If the tasks size is small the communication overhead between the manager and the workers would be high. This can however often be solved by letting several tasks be distributed at each request (if the order of tasks are not important). The manager/worker scheme is a simple but effective task scheduling scheme for a low to moderate number of physical processors. An illustration of the manager/worker scheme can be seen in figure 2.5.
- **Decentralized Scheme**[2] - In a completely decentralized schemes, there is no central manager. Instead, a separate task pool is maintained for every worker, and if a worker has no more tasks to do, it request tasks from the other workers. In effect, the task pool becomes one large distributed data structure which is accessed by the workers in an asynchronous manner. This scheme doesn't suffer from the communication bottleneck associated with a centralized manager but it loses the ability to control in which order

[1] Also known as the Master/Slave scheme.
[2] Also known as the Distributed scheme.

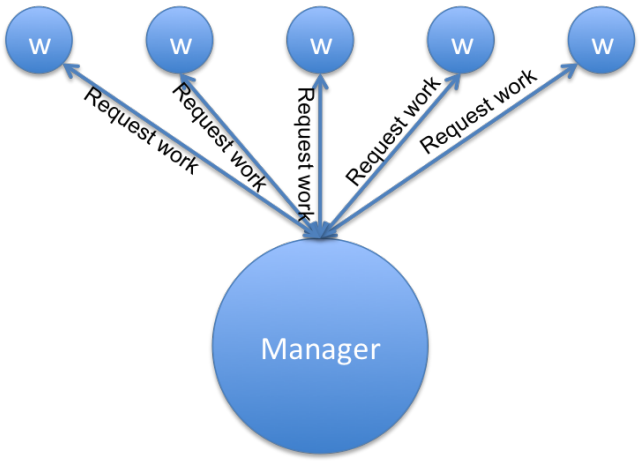

Figure 2.5: Overview of the manager/worker load-balancing strategy

the different tasks should be processed. It is also important to decide how and when the actual communication between the different workers should take place and how to determine when the execution is completed. An illustration of the decentralized scheme can be seen in figure 2.6.

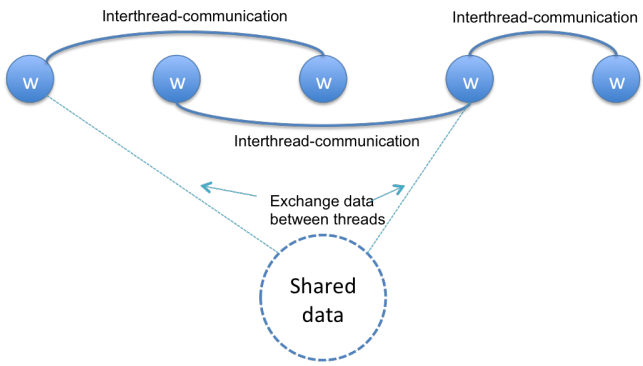

Figure 2.6: Overview of the decentralized load-balancing strategy

2.2.3 Access control

When solving problems concurrently, shared data-structures are often used and the need for controlling access to these structures becomes important. If two threads tries to access and write to the same part of the data structure, inconsistencies can occur. The basic tools used to secure the access control is locks and other synchronization primitives. They will not be discussed in more details in this section as they are the subject of Section 2.3.

2.2.4 Challenges of parallel programming

Developing parallel algorithms consists of many challenges which in some way influences the performance. This section sums up of the most important challenges.

Parallel Overhead Parallel overhead refers to the amount of time required to coordinate parallel tasks, which is at the cost of time of doing some useful computation. Typical parallel overhead includes the time to start/terminate a tasks, the time to pass messages between tasks and other extra computation time. When parallelizing a sequential program, overhead is inevitable, but should be kept at a minimum by a thorough analysis following the steps described in this section. Especially the decomposition and granularity should be kept in mind when trying to decrease the overhead.

Synchronization Synchronization is necessary in concurrent programs e.g. to prevent race conditions among threads. Synchronization limits the parallel efficiency even more than the parallel overhead, as it actually serializes some part of the program and thereby limits the number of concurrent activities. Improper synchronization can, besides slowing down the execution time, also lead to incorrect results as it can be a complex tasks to control access to important parts of the code. Synchronization is hard to automate and it os often the sole job of a developer to ensure the correctness of the program. Mathematical tools can help in proving that the synchronization is behaving correctly, but they are most often very time-consuming and complex to use on larger parts of code.

Load Balance Load balance is important when parallelizing sequential programs because poor load balance causes under-utilization of the processors. Processor idle-time should always be kept at a minimum, but this can be hard if the

number and size of the different subtasks are unknown at the start of the execution. Load balance can be achieved using different algorithms and by carefully analyzing the problem.

2.3 Synchronization primitives

The implementations in this thesis are all based on the shared memory model, and as mentioned in Section 2.1 synchronization primitives are often used to control the access to the shared memory locations. One of the most important techniques to control access to shared memory locations is mutual exclusion, i.e. to let only one thread into a critical region of code at a time. Mutual exclusion is the synchronization technique predominantly used in this thesis, and is therefore the one focused on. Other important techniques worth mentioning are condition synchronization and true synchronization and these are investigated further in [16].

This section will introduce the relevant synchronization primitives in the classical concurrency theory followed by a short introduction to a modern data structure which doesn't utilize the classical synchronization primitives [17].

2.3.1 Locks

Locks are the most simple form of synchronization primitive used to synchronize the communication between different threads. They can be implemented in many different ways depending on the machine architecture but their task is almost always the same - to limit the access to a section of code by providing mutual exclusion. Locks usually consists of two operations - Lock/Acquire and Unlock/Release - and depending on the implementation, a thread which tries to acquire a lock is set to wait, if another thread already has acquired the lock. Locks provides a very fine-grained way of controlling the synchronization between threads, but is also very error prone, because it is very easy to provoke deadlocks and starvation. This is due to the fact, that it is up the developer to make sure, that the lockings are done correctly.

2.3.2 Semaphores

One of the first multiprogrammed operating systems was developed by E. Dijkstra in the mid-1960s [6], and with it came the need for mutual exclusion between treads in a more controlled way than just by using locks. Dijkstra therefore introduced the semaphore concept. A semaphore relies on locks and is a shared variable which is initialized to a non-negative integer count which only can be manipulated by two atomic operations, **P** (from the dutch word "proberen", meaning "to test") and **V** (from the dutch word "verhogen", meaning "to increment"). When a thread executes **P** on the semaphore, the count is tested to see if it is greater than 0. If that is the case, the semaphore decrements the count. If the count is 0, the calling thread is set to wait in the queue. When a thread executes **V** on the semaphore, the semaphore determines if one or more threads are waiting in the queue. If so, the semaphore allows one of those threads to proceed. If no threads are waiting, the count increments.

The classical way of implementing the operations of a semaphore can be seen in listing 2.1.

```
1  P(Semaphore s)
2  {
3    wait until s > 0;
4    s := s - 1; /* must be atomic operation */
5  }
6
7  V(Semaphore s)
8  {
9    s := s+1;   /* must be atomic */
10 }
```

Listing 2.1: Semaphore test

Semaphores can be classified as either counting semaphores or binary semaphores. A semaphore is a counting semaphore if it can assume any non-negative integer value, and if the semaphore can only assume the values 0 or 1, it is known as a binary semaphore. A binary semaphore is in many ways the same as a simple lock.

2.3.3 Monitors

Semaphores and locks are a very effective and fine-grained way of handling the access to share memory locations, but they are also very error-prone for the developers. A simple mistake e,g. by forgetting to acquire semaphores in the correct order, could result in deadlocks making the code unusable. To let the

developers concentrate on other thing than the synchronization, the monitor concept was invented.

In the classical monitor approach, as stated by E. Dijkstra, P. Brinch-Hansen and C.A.R. Hoare ([12]), a monitor object contains a lock which is acquired by the thread that enters the monitor. If a thread tries to enter the monitor while another thread has acquired the lock it is blocked. The semantics of the classical monitor pattern ensures that all threads exits the monitor before they release the lock and thereby ensuring that only one thread can be in the monitor at any given time (and thereby securing mutual exclusion). In the classical monitor approach a thread can exit the monitor by exiting completely or by waiting on a waiting queue (a condition queue). In both cases the lock on the monitor is released, and another thread can acquire the lock. A waiting thread can be woken up by another thread notifying the waiting queue (signaling). In early implementations, notifying a waiting queue caused a waiting thread to receive the lock immediately from the signaler. The signaler was then places on a separate queue waiting to re-acquire to the lock. This method is called Signal-and-wait but due to implementation overhead [16] it has been substituted by the Signal-and-continue method. In the Signal-and-continue method the signaler continues and the signaled thread is woken up and waits to acquire the lock.

2.3.4 Lock-free and wait-free algorithms

Synchronizing the access to a shared data structure can be a complicated matter and a lot of research has been put into the development of data structures that doesn't require the use of the classical synchronization primitives. These algorithms are known as *lock-free* and *wait-free*. Most lock- and wait-free algorithms are developed using atomic primitives provided by the operating system and the underlying hardware. The most notable of these atomic primitives is *compare-and-swap* (often notated "CAS"). Pseudocode of the CAS primitive can be seen in listing 2.2.

```
1 CAS(addr, old, new) = atomic
2                         if *addr = old
3                            then *addr := new ;
4                                 return true
5                         else return false
6                         endif
7                       endatomic
```

Listing 2.2: Compare-and-swap primitive

The CAS takes three arguments: a memory address, an old value, and a new value. If the address contains the old value, it is replaced with the new value,

otherwise it is unchanged. Critically, the hardware guarantees that this "comparison and swap" operation is executed atomically. The success of this operation is then reported back to the program. This allows an algorithm to read a value from memory, modify it, and write it back only if no other thread modified it in the meantime. Usually the CAS primitive is used repeatedly until a thread succeeds in writing to the shared data structure.

2.4 Java Synchronization Primitives

As stated in the previous sections, there exist a number of different synchronization primitives which can be used to control the communication between different tasks/threads. The programming language used in this thesis is *Java* and this section will give an introduction to Java's take on the different synchronization primitives. The synchronization primitives of Java has evolved a lot in the most recent versions and can be found in the `java.util.concurrent` package. Most of the primitives has been in Java since version 1.5, and has been improved ever since.

2.4.1 Semaphore - Locks

Java provides the package `java.util.concurrent.locks` which contains different implementations of the classical lock. The lock follows the classical lock closely, and provides a very flexible and fine-grained way of designing critical regions in code. Locks can intersect each other whereas a monitor in Java only can be contained within another. A critical region in which two locks intersect can be seen in listing 2.3.

```
1
2 method() {
3     A.Lock();
4     B.Lock();
5     //Critical region
6     A.Unlock();
7     B.Unlock();
8 }
```

Listing 2.3: Intersection between locks

Semaphores in Java is located in the package `java.util.concurrent` and is a standard counting semaphore as described in the previous Section 2.3.2. A semaphore in Java maintains a set of permits, which can be acquired and released

by threads. A semaphore in Java accepts a fairness parameter which sets the order in which permits are handed out. If the fainess parameter is set to `false`, no guarantees are made about the order in which threads acquire permits, and if set to `true` the permit order is controlled by a FIFO queue (First-In,First-Out).

A Semaphore in Java also contains a number of auxiliary methods, which can be used to inspect the number of permits, as well as the number of threads waiting. These methods should however not be used to do any synchronization, as they are not always up to date.

2.4.2 Monitors

Java contains the `synchronized` primitive which reassembles the classical monitor idea in many aspects. The Java implementation of the monitor is a bit more flexible as it allows any kind of object to be used as a monitor lock, and like the classical approach, it only allows one thread to have acquired the lock at any given time. In the classical monitor it is possible to have several condition queues which the different threads can wait at, but in Java there is only a single common waiting queue. A Java monitor utilizes the "Signal-and-Continue" method as most other modern programming languages. One thing that a programmer should notice when doing parallel programming in Java is it's ability to do spurious wake-ups. A spurious wake-up is when a thread is woken up even though it hasn't been signaled. It is therefore important to test if a thread that reenter the monitor, is allowed to proceed execution, otherwise it should continue waiting. An example of a monitor which takes spurious wake-ups into account is given in listing 2.4.

```
1
2  public class PreventSpuriousWakeUps {
3
4      private int c = 0; //A counter which only should be incremented by one
           thread at a time
5
6      public synchronized void increment() { //Synchronized method
7
8          //C should only be incremented if it is less than 5.
9          while(c < 5) {wait();} //While is needed instead of if to prevent
               spurious wake-ups.
10
11         c++;
12     }
13 }
```

Listing 2.4: Testing for spurious wake-ups

The `synchronized` primitive can be applied at two levels in Java - either by

applying it to a complete method (Synchronized methods) or by applying it to a single statement (Synchronized statements). This provides the programmer with the possibility to use a more or less fine-grained type of synchronization.

Listing 2.5 provides a simple example of the difference between a synchronized method which only one thread can access at a time, and a synchronized statement which only limits the access to a single statement.

```
1  public class SynchronizedCounter {
2      private int c = 0; //A counter which only should be incremented by one
           thread at a time
3
4      public synchronized void increment() { //Synchronized method
5          c++;
6      }
7
8      public void decrement() {
9
10         //Do something which doesn't require synchronization...
11
12         synchronized(this) { //Syncronized statement
13             c++
14         }
15     }
16 }
```

Listing 2.5: Synchronized method vs. synchronized statement

It is important to mention that by using monitors in Java you have a larger overhead than by using the more simpler locks and semaphores. If a class contains more than one synchronized method only one thread can access a method at a time and this can result in unnecessary waiting.

Chapter 3

Maximum flow algorithms

In this chapter the maximum flow problem is introduced. This includes a general introduction as well as a more mathematical explanation. This is followed by an introduction to the maximum flow algorithms used in the thesis with focus on the Push Relabel algorithm.

3.1 Introduction to the maximum flow problem

The algorithm focused on in this thesis is the *Push Relabel Algorithm*, which belongs to the group of algorithms that addressed the *Maximum Flow Problem*. The problem or task of the maximum flow problem is to find the maximum rate which material or data flows in a *Flow Network*. Just as a *directed graph* can be used to represent roads in a city, a *Flow Network* can too be described by a *directed graph*. An easy way to imagine a *Flow Network* is to think of liquid running through a system of pipes (e.g. a sewer). The pipes are represented as *Directed Edges* in the *Flow Network*, and because pipes in a system can be of different sizes and therefore can transport different amounts of liquid, each directed edge has a *Capacity*. This capacity is the maximum rate at which the material (liquid in the pipe-example) can flow through the pipe. It is not required that the *capacity* of an *edge* is being used to full but it figures as an upper bound.

The pipes in the system are connected via knots where a number of pipes go into and a number of pipes leave. These knots are represented as vertices in the Flow Network. When material (liquid in the pipe-example) runs through a vertex it will not be collected within the vertex (it has no reservoir), which means that the amount of material which goes into a vertex must be equal to the amount of material leaving the vertex. (This will be described later as "*Flow conservation*"). The pipe-system must of course also have a place where the liquid "starts" and a place where it "ends' - this is represented by special source and sink vertices in a Flow Network.

To sum up, the Maximum Flow problem is used to find the maximum feasible flow from the source to the sink in a Flow Network. It both computes the flow for all vertices and the value of the maximum flow.

3.1.1 Theory & Definitions

This section will cover the basic formal theory used to explain and discuss the Maximum Flow Problem. The definitions and theory covered in this section follows the definitions from the section on maximum flow in [5, chapter 26] .

3.1.1.1 Flow network

A Flow Network is a directed graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges. Two vertices in the graph are defined as the source s and sink t. For every edge $(u, v) \in E$ there exists a positive real-valued capacity function c, so the following applies: $\forall(u, v) \in E \Rightarrow c(u, v) \geq 0$. Also if $(u, v) \notin E \Rightarrow c(u, v) = 0$. This means that if the edge is not in the graph it has the capacity 0. An example of a flow network can be seen in figure 3.1.

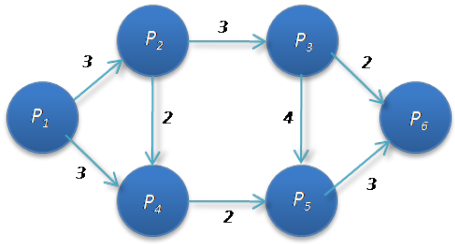

Figure 3.1: Example of a flow network

A Flow Network can have multiple sources and sinks but this thesis will only

cover single-source, single-sink networks. If the extended networks were covered, a supersink and a supersource connected to the sources and sinks via edges with unlimited capacity.

3.1.1.2 Flow

Given a Flow Network $G = (V, E)$ with source s and sink t and a capacity function c. A Flow in G is a real-valued function $f : V \times V \rightarrow R$ where R is the value of the flow over the pair of edges. The function f must satisfy the following properties:

- Capacity Constraint: $f(u, v) \leq c(u, v)$ for all u, v in $V \times V$
- Skew Symmetry: $f(u, v) = -f(v, u)$ for all u, v in $V \times V$
- Flow Conservation: $\sum_{v \in V} f(u, v) = 0$ for all u in $V - \{s, t\}$

In words, the *Capacity Constraint* says that the flow from one vertex to another must not exceed the given capacity. *The Skew Symmetry* defines that the flow from one vertex to another is the negative of the flow in the opposite direction. Finally the *Flow Conservation* says that the total flow going into a vertex must be equal to the total flow going out of the vertex, hence the sum of the ingoing and the outgoing flow is zero. This property is often referred to as the "Flow in equals flow out". This is not valid for the source and sink.

The value of the flow of a network is the net flow entering the sink vertex t and therefore also the net flow leaving the source vertex s. The mathematical notation of this can be seen in equation 3.1.

$$|f| = \sum_{u \in V} f(u, t) = \sum_{v \in V} f(s, v) \tag{3.1}$$

The goal of the maximum flow problem is to compute the maximum possible value for $|f|$ and the corresponding flow values for each pair of vertices in the flow network.

3.1.1.3 Residual network

Given a Flow Network $G(V, E)$ with a flow f. The *Residual Network* $G_f(V, E)$ consists of edges which can admit more flow. If an edge $(u, v) \in E$ has a flow

which is lower than its capacity, then the amount of additional flow we can push before exceeding the capacity $c(u, v)$ is the residual capacity given by equation 3.2.

$$c_f(u, v) = c(u, v) - f(u, v) \tag{3.2}$$

There are cases where the residual capacity can have a greater value than the actual capacity. For example if $c(u, v) = 8$ and $f(u, v) = -2$ then the residual capacity is $c_f(u, v) = 10$. This means that there is a flow of 2 units from v to u which we can cancel by pushing 2 units from u to v. Because the capacity from u to v is 8, we can push additional 8 units without exceeding the capacity, thus we have pushed a total of 10 units from u to v.

Residual networks also introduces the concept of augmenting paths. An augmenting path represents a simple path from the source s to the sink t which is in the residual network. Given the definition of a residual network, it follows that each edge on the augmenting path can admit some additional positive flow without violating the capacity constraint.

3.2 Maximum Flow Algorithms

This section introduces the different maximum flow algorithms relevant for this thesis. Usually the algorithms used to compute the maximum flow of a network are divided into two types [5, chapter 26]:

Augmenting Path Algorithms: These types of algorithms uses the definition of augmenting paths to push flow incrementally from the source to the sink. Augmenting path algorithms always complies with the three basic flow constraints introduces in Section 3.1.1.2.

Preflow Push-Relabel Algorithms: These algorithms start by "flooding" the entire network, and then incrementally relieving flow from unbalanced vertices by sending flow towards the sink t or backwards towards the source s depending on the capacity of their edges. By unbalanced vertices is referred to vertices where the Flow Conservation is not satisfied. As vertices can be unbalanced during the execution of the algorithm the flow constraint is not satisfied trough-out the execution. However, the flow constraint is required to be satisfied when the execution terminates.

3.2.1 Edmonds-Karp Algorithm

The Edmonds-Karp Algorithm [8] is an implementation of the more abstract *Ford-Fulkerson* method [15] which is based on the *Augmenting Path* theory. The main idea of the Ford-Fulkerson algorithm is very simple: As long as there is a path from the source s to the sink t with the possibility to admit more flow (an augmenting path), flow will and should be sent along these paths. This will be done iteratively until no path can be found from the source to the sink. The max-flow min-cut theorem [5, Theorem 26.7] shows that upon termination, the computed flow will be the maximum flow.

How an augmented path should be found is not precisely described in the Ford-Fulkerson algorithm and it can be implemented in different ways. The Edmonds-Karp algorithm is one possible implementation and it basically uses a *Breadth-First-Search* (BFS) to find augmenting paths - The computed path is therefore actually the shortest augmented path from the source s to the sink t. The pseudo code of the Edmonds-Karp algorithm can be seen in listing 3.1.

```
1
2  For each edge(u,v) in G
3    Do f(u,v) = 0
4       f(v,u) = 0
5  While an augmenting path p can be found by BFS
6    Do cf(p) = min(cf(u,v) : for all (u,v) in p)
7       For each edge (u,v) in p
8          Do f(u,v) = f(u,v) + cf(p)
9             f(v,u) = -f(u,v)
```

Listing 3.1: Pseudo code for Edmonds-Karp. cf(u v) is the residual capacity

It has been shown that the running time of the Edmonds-Karp algorithm is $O(V*E^2)$ [8]. This should, however, only be seen as the theoretical running time as in practice the algorithm performs very well on most graph types, especially on sparse graphs.

3.2.2 Push-Relabel

The Push-Relabel algorithm is of the type *Preflow Push-Relabel Algorithms*, and is to date one of the asymptotically fastest maximum-flow algorithms [10].

The algorithm works on one vertex at a time, and it is very local as it only looks at the direct neighbors to the vertex in the residual network. In each iteration

of the algorithm, the flow is a *preflow*, which satisfies:

$$f(u,v) \leq c(u,v)$$

$$f(u,v) = -f(v,u)$$

$$f(V,u) \geq 0, \forall u \in V - \{s\}$$

The first two conditions are well known from the definition of a flow, but the last condition is new: This is called the *Excess Flow* into u, and is the sum of the flow going into and out of v, which can be written

$$e(u) = f(V,u)$$

The way the algorithm works is in short, to push flow from one vertex to others. To compensate for the excess flow we imagine that each vertex has some "extra storage" for temporarily storing the excess flow. Secondly, each vertex has a *height* (also called *label*) - at first the height of the source is set to $|V|$ and the height of all other vertices are set to zero. The height dictates where to push the flow. Only a push from a vertex u to v is legal, if the height of u is greater than for v. That does not mean that a flow from a lower vertex to a higher cannot be positive, but it is only possible to push flow downhill. The algorithm then starts by sending as much flow as possible from the source to all of its neighbors which collects the flow in their temporary storage. From there, it is eventually pushed downhill towards the sink.

If the algorithm reaches a vertex with excess flow, where the only edges capable of receiving more flow is either on the same or a heigher level, a *relabeling* is performed on the vertex. This means that its height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated edge to. This makes sure that there exists at least one outgoing edge where it is legal to push flow.

The sink will eventually have received the maximum amount of flow it can possible receive, but the preflow is probably not a valid flow because some vertices do not satisfy the Flow Conservation constraint. The algorithm therefore continues to push the excess flow around the network until all vertices have an excess flow of zero. A consequence of this is that at some point, it is necessary to push flow back to the source, and it is therefore possible to relabel a vertex to above the fixed height $|V|$ of the source. It has then been proven, that the flow at the end is both a legal flow and a maximum flow [5, chapter 26].

3.2.2.1 Push-Relabel operations

The above introduction to the Push-Relabel algorithm uses the terms "push" and "relabel" as the possible operations the algorithm can choose to take in each step. A more formal explanation of these two operations is given in this section.

Push Operation A Push operation from u to v where $u, v \in V$ can be explained as sending some or all of u's excess flow from u to v. A push operation is only legal if the following conditions are satisfied:

$$c(u,v) - f(u,v) = c_f(u,v) > 0$$

$$e(u) > 0$$

$$h(u) = h(v) + 1$$

The first conditions says, that it shall be possible to admit flow from u to v, which only is the case if the edge between the two vertices is not *saturated*, meaning that the actual flow trough the edge is less than the capacity of the edge. The second condition says that a push operation only is possible if vertex u has more ingoing than outgoing flow and thus being an overflowing vertex. The last condition defines that a push operations only is legal if flow is being pushed to a vertex with a height which is exactly one unit smaller than the heigh of u. The amount of flow which can be pushed between two vertices is determined by the bottleneck of the edge: $min(e(u), c(u,v) - f(u,v))$. The pseudocode of the push opration can be seen in listing 3.2.

```
1 Push(u,v)
2 If e(u)>0 and cf(u,v)>0 and h(u) = h(v)+1 //Checks conditions
3   Do df(u,v) = min(e(u),cf(u,v)) //Finds flow bottleneck
4      f(u,v) = f(u,v) + df(u,v) //Updates flow
5      f(v,u) = -f(u,v)
6      e(u) = e(u) - df(u,v)  //Updates excess
7      e(v) = e(v) + df(u,v)
```

Listing 3.2: Pseudo code for Push operation

Relabel Operation When doing a relabel on a vertex u, its height is increased. A relabel operation is only allowed if u has an excess flow and for every vertex v with a residual capacity from u to v, the height of v is greater than the height of u. In short, a relabel operations should only be performed if a push operation is not possible. A vertex's height is also called its label (hence

Relabel-operation). The pseudocode of the relabel operation can be seen in listing 3.3.

```
Relabel(u)
If e(u)>0 and for all v in V such that (u,v) is in Ef we have h(u)<=h(v)
   Do h(u) = 1 + min(h(v) : foreach (u,v) in Ef) //Add 1 to the smallest
        height of the neighbors of u.
```

Listing 3.3: Pseudo code for Relabel operation. Ef means edges in the residual network.

3.2.2.2 The Generic Push-Relabel

The generic and original push-relabel algorithm defined by A.V. Goldberg and R.E. Tarjan [10], starts out by initializing the preflow. The initialization push as much flow from the source to its neighbors and thereby filling all of its neighboring edges to their capacity. All other edges are initially assigned a flow of zero. The pseudocode of the initialization can be seen in listing 3.4.

```
Initialize-Preflow(G,s)
For every vertex, set height and excess flow to zero
For every edge, set flow to zero
H(s) = |V(G)|
For every vertex u <- Adj(s)
  Do f(s,u) = c(s,u)
     f(u,s) = -c(s,u)
     e(u) = c(s,u)
     e(s) = e(s) - c(s,u)
```

Listing 3.4: Pseudo code for initializing the preflow

When the initial preflow in the flow network has been created, the actual algorithm can be executed. The algorithm can be seen in listing 3.5.

```
Generic-Push-Relabel(G)
 Initialize-Preflow(G,s)
 While there exists an applicable push or relabel operation
    Do Select a push or relabel operation and perform it
```

Listing 3.5: Pseudo code for Generic Push-Relabel

The algorithm, however, is rather abstract and there are a lot of different ways to implement it. It has been proven, that the order of which the push-relabel operations are performed, has no significance on the correctness of the result, but at the same time it has been shown that the order has great impact on the running time of the algorithm. If the operations were executed in a completely

random order, the running time is $O(V^2 * E)$. The first optimization is gained by combining the push and relabel operations in *discharge* operations in which the node being processed, performs push and relabel operations until it no longer has a positive excess flow. This, combined with ordering the active[1] vertices in a *FIFO-Queue* (First in, first out), results in a running time of $O(V^3)$. If the queue uses dynamic trees instead of a FIFO-queue the running time is further improved to $O(V * E * log(V^2/E))$ and if the queue is ordered by height the running time will be $O(V^2 * \sqrt{m})$.

A special case of the push-relabel algorithm called the **Relabel-To-Front** algorithm alters the relabel operation by moving relabeled vertices to the front of the queue of active vertices. The running time of the relabel-to-front algorithm is the same as the generic push-relabel algorithm using FIFO-ordering: $O(V^3)$ [5, chapter 26].

3.2.2.3 Push Relabel with Heuristics

The generic implementation of the push relabel algorithm performs very poorly in practice, which also was the conclusion of work done by A.V. Goldberg and R.E. Tarjan in [10]. The running times are often be close to the worst case - this is because the relabel operation is a local operation which only looks the current vertex and its direct neighbors. The distance to the sink is not taken into account which often results in pushing flow back and forth in the network many times during the execution. To improve this, an implementation which uses heuristics was introduces by A.V. Goldberg and B.V. Cherkassky in [4].

The way the heuristics works is split up into two steps: The first step is to only look at the residual network and do a *Breadth First Search* from the sink. At every iteration the height of the vertex is being set to the actual distance from the sink, so when the BFS is done all the shortest paths to the sink has been found and is represented as the height of each vertex in the residual network.

As the residual network is used in the breadth first search, some vertices might not have been included in the search because no possible path exists from the vertex to the sink. The next step is therefore to look at the complementary to the residual network. As above, a BFS will be performed, but this time from the source. The clever thing about this is, that if an edge is not in the Residual Network, it cannot admit any more flow towards the sink and should therefore push the flow backwards towards the source. By doing this second step, the algorithm can easily push excess flow back towards the source at an

[1] An active vertex is defined as a vertex with an excess flow above zero.

early stage of the execution and thereby minimizing the need for pushing flow around several times.

When doing these BFS-algorithms, all vertices in the network are relabeled with respect to the distance to the sink and source respectively - this is called doing a *Global Relabeling*. These global relabeling procedures are computational heavy compared to push and relabel operations and are therefore only done periodically after a given number of push/relabel operations. The theoretical running time of the Push Relabel algorithms with Global Relabeling is the same as the generic algorithms mentioned earlier, but in practice it is much faster because of the distance-ordering of the vertices. Studies [4] have shown, that by using the Global Relabeling heuristics the push relabel algorithm outperforms all other known maximum flow algorithms on most graph types.

Another type of heuristics mentioned by A.V. Goldberg and B.V. Cherkassky in [4] is the *Gap Labeling Heuristic*. This heuristic updates the labels of the vertices which are unreachable from the sink and sets their label to the label of the source, which is $|V|$. Such a situation arises if there are no vertices with labels g, but vertices v with labels $g < label(v) < |V|$. Then the sink is not reachable from any of these vertices and their labels can be increased to $|V|$. Such an update makes it possible to remove these vertices from consideration for pushing flow to the sink at once. The improvement of the Gap heuristics is not as great as with the Global Relabeling heuristic, but it can to some extend be combined with the global relabeling heuristic and thereby resulting in a very effective Maximum Flow algorithm[4].

Chapter 4

Parallelization of the push relabel algorithm

In this chapter we present the parallelization of two variant of the classical maximum flow algorithm *Push-Relabel*. The level of detail is increasing during the different sections, starting out with a high-level analysis using the step-by-step parallelization method presented in [11], followed by a more detailed explanation of how the actual implementation has been done using Java.

4.1 Analysis

The push relabel algorithm consists of a lot of independent operations (push & relabel) which can be processed in any order without violating any of the central Lemmas concerning maximum flow. Due to the many independent operations, the algorithm could be a good candidate to achieve maximum performance increase using parallelization.

As mentioned in Section 3.2.2 there exists a number of push relabel algorithms using different queues and heuristics. This section will focus on parallelizing a simple push relabel algorithm with FIFO[1] queue ordering, and a more complex

[1]FIFO is an acronym for First In, First Out.

push relabel algorithm with global relabeling heuristics. For each of the two algorithms, the parallelization steps will be explained and the approach used is the one already described in Section 2.2: Decomposition, mapping and access control. A more detailed explanation of how the two algorithms has been implemented using Java, can be found in Section 4.2 and an overview of the source code can be found in Section 4.3.

4.1.1 Simple Parallel Push Relabel

4.1.1.1 Decomposition

The first step of the parallelization is the decomposition into tasks. A natural and perhaps a bit naive approach would be to define a task as a single push or relabel operation. This would, however, be a very fine-grained decomposition and would probably, especially in Java [9, Chapter 15.1], create a lot of synchronization overhead. The solution used in this thesis is to define a task as a complete discharge of a single vertex. This is a more coarse-grained solutions which could perform well on the Java architecture. By performing a complete discharge, push operations are performed in combination with relabel operations until all excess flow of a vertex is removed. A vertex can therefore be removed from the queue of active vertices when a complete discharge has been performed.

4.1.1.2 Mapping

The task size of the push relabel algorithm can be defined in advance but the number of tasks is unknown and varies over time. A complete discharge of a vertex can result in several new active vertices and thereby new tasks. This can not be predicted and a load balancing solution should therefore be applied. Without a load balancing algorithm one or more threads/processors could run out of work and would become idle.

The two load balancing strategies mentioned in Section 2.2.2 has been considered and resulted in the following two solutions:

- **Manager/Worker**: Consists of a centralized manager thread which maintains a global queue containing all active vertices and a number of worker threads (e.g. one per physical processor). Each worker thread repeatedly fetches a batch of active vertices from the manager, process them (using

complete discharges), and returns all newly discovered active vertices to the manager. The most efficient batch size can be hard to determine as all sizes has different impacts on the performance. A large batch size minimizes the communication overhead but the order in which the vertices are processed is different than the sequential FIFO queue implementation, which studies shows can lead to a performance decrease. If the batch size is small the communication overhead is increasing but the vertex processing order is almost identical to the sequential implementations of the queue variant of the push relabel algorithm. An illustration of the manager/-worker load balancing implementation can be seen in figure 4.1.

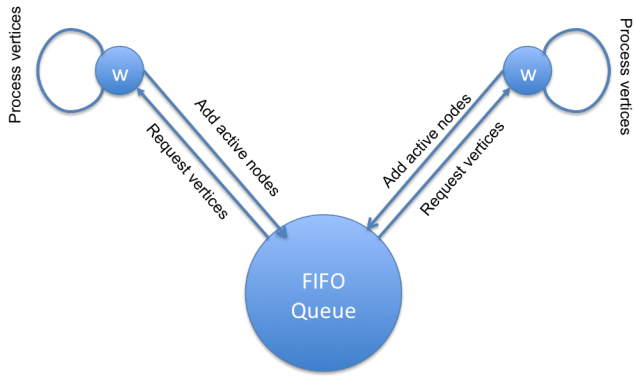

Figure 4.1: Overview of the manager/worker load-balancing implementation

- **Decentralized**: When the initial preflow is created, the active vertices are distributed to each of the worker threads equally. Each worker thread then works autonomously by processing its own active vertices and adding the newly discovered active vertices to its own list of vertices to be processed. In an ideal execution, each worker thread does not run out of work before any other thread, and when the threads are done processing their own list of active nodes the computation of the maximum flow is complete. This, however, almost never hold in a real world execution. A worker thread probably runs out of work to do at some point during the execution and should therefore request work from the other worker threads. This is solved by keeping track of the idle threads and if a thread discovers a new active vertex while there is an idle thread it adds the vertex to a global queue instead of its own local queue. The idle thread is then woken up and fetches the active vertex from the global queue. The advantages of using this load balancing solution is that the communication overhead is keep to a bare minimum but this is at the cost of the vertex processing order which is often far from the FIFO queue ordering of the sequential

push relabel algorithm. An illustration of the decentralized load balancing implementation can be seen in figure 4.2.

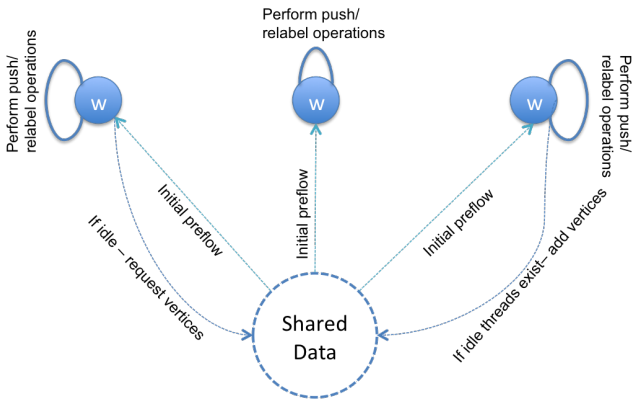

Figure 4.2: Overview of the decentralized load-balancing implementation

4.1.1.3 Access Control

Both of the above mentioned load balancing strategies utilize a shared queue at some point. The manager/worger strategy utilizes the queue all the time and the decentralized strategy only utilizes the queue when a thread becomes idle. In both cases, more than one thread can try to manipulate the queue at the same time and without proper access control the data could become inconsistent. Two possible solutions to this problem has been identified:

1. The queue is implemented as a standard FIFO queue and the access to the list is controlled via Locks or semaphores. Before manipulating the list (read and write) a thread has to acquire the lock to the queue and when it is done it releases the lock. This solution is a well known way to handle access to a shared resource but could suffer from a large synchronization overhead.

2. The queue is implemented using a wait-free algorithm as described in [17]. This requires that the programming language supports atomic increment operations and is only efficient if the operation system supports the compare-and-swap operation (see Section 2.3.4). This solution is often more intuitive to the programmer as he or she doesn't have to take the actual synchronization into account, but it could be less efficient than a

solution based on classical synchronization primitives due to an inefficient implementation of the wait-free algorithm.

Several other possible solution to the shared queue access control exists and one of them would be to use monitors instead of locks/semaphores. This would be a more controlled environment for the programmer leaving less room for human error, but this has not been implemented because the monitor implementation in most languages suffers from performance decrease compared to simple locks/semaphores.

Another important access control issue to address in regards to the push relabel algorithm, is the actual data structure of the vertices and edges. The algorithm repeatedly performs push and relabel algorithms to vertices in the graph, and it would be possible for more than one thread to perform an operation on the same vertex at the same time. A solution to this problem has been presented in [2] and is fairly straight forward:

- **Push** - A thread must acquire the lock on both the vertex which flow is pushed from and the vertex which flow is pushed to. As two locks can't be acquired at the same time it is important to always lock vertices in the same order to avoid deadlocks (eg. if one thread locks vertex A and tries to lock vertex B which already is acquired by another thread that tries to lock Vertex A). This is handled by always locking the vertex with the lowest ID number first.
- **Relabel** - A thread must acquire the lock on the vertex it tries to relabel.

4.1.2 Parallel Push Relabel with global heuristics

The parallel push relabel algorithm with global relabeling heuristics is in many ways similar to the simple push relabel algorithm with a FIFO queue. The only difference is the heuristic function which is described in Section 3.2.2.3.

4.1.2.1 Decomposition

The decomposition into smaller tasks is similar to the simple algorithm but a new kind of tasks is now present due to the global relabeling. The global relabeling consists of two breath first searches which is executed a number of times during the execution of the algorithm. The heuristic function could be decomposed into

even smaller tasks which could be distributed to more processors but this has not been done in this thesis, mainly due to the fact that only a limited number of processors were available.

4.1.2.2 Mapping

The global relabeling could be mapped to physical processors in different ways. One possible solution would be to let the different threads do their normal routine of push and relabel operations and once a while let one of the threads to a global relabeling (eg. every n push/relabel operations). However, studies have shown that the heuristics plays a very important part in the performance and should not only be granted the resources of one thread once a while. Another solution would be to let all threads do the global relabeling together at a given interval. This would probably be the most optimal solution (as suggested by [4]), but is also more complicated to implement as the breath first search also needs to be parallelized. The solution chosen for this thesis is to let one worker thread do the global relabeling continuously throughout the execution. This gives the global relabeling more priority than the first solution but less than the second.

4.1.2.3 Access Control

In the simple push and relabel algorithm only the push and relabel operations manipulated the vertices, but this changes with the introduction of the global relabeling function. The function relabels vertices to a value corresponding to their "distance" to either the sink and source and should therefore acquire a lock on a vertex before performing any relabeling. If the relabeling was performed by several threads at the same time as normal relabeling and push operations, more restrictions and locking mechanism should be implemented to avoid inconsistencies. This is described in [4] and will not be examined further in this thesis.

4.2 Design

This section will cover the general design of the algorithms and the underlying graph framework. A basic graphical user interface (GUI) has also been developed to ease the testing of the algorithms and it will be discussed in Appendix

B. The section is intended to be programming language independent but several object oriented concepts are used throughout the section.

To ease the clarity of the algorithms and the framework, all related objects has been divided into packages. This is best practice in most programming languages and makes it a lot less complicated to acquaint oneself with the framework.

Our design is divided into the following sub-domains which all should be represented as a package[2]:

- `algorithm`: The top package for all algorithm related subdomains.
- `algorithm.parallel`: Contains all versions of the parallelized algorithms.
- `algorithm.reference`: Contains the reference-algorithms which are used to determine the correctness of the results gained from the parallelized algorithms.
- `algorithm.sequentiel`: Contains the basic sequential push relabel algorithms.
- `edge`: Contains all classes related to edges in a graph.
- `vertex`: Contains all classes related to vertices in a graph.
- `graph`: Contains all classes related to the representation of a graph.
- `graph.generator`: Tools/classes for generating new graphs of different types.
- `graphLoader`: Tools for loading a graph from its file-representation into memory.
- `gui`: The graphical representation of the framework.
- `statistics`: The tools used when measuring runtimes and other statistics when executing the algorithms.
- `test`: Contains all tools related to the testing of the correctness of the algorithms.

All the above packages defines the graph framework and the algorithms. The design of the individual packages as well as the interaction between the packages will be explained in the following sections.

[2]The prefix dk.dtu.imm.parallelmaxflow has been removed from the package names to ease the understanding.

4.2.1 Design Overview

The following sections gives an overview of how the actual framework and algorithms are designed. The design is language independent and a programmer should be able to implement a working program by following the design in combination with the analysis from Section 4.1.

Throughout the design we have chosen to create *Interface*'s for every object. This eases the integration with new or existing frameworks and it is important that the interfaces are used to access objects instead of accessing objects directly.

4.2.1.1 Graph Framework

This section will describe how a Graph and its components are being represented. There exists several pre-made Graph Frameworks such as JUNG [3][18], which represents graphs, edges etc, but we chose to make a simple framework from scratch. This decision has been made to maintain full control of all data structures, and it allow us to define the exact functionality and definitions. One limitation of using our own framework is that it is harder to utilize pre-made graph illustration tools, but as graph illustration never has been the target of this thesis, it is not an important shortcoming.

An UML-like illustration of the complete graph framework is shown in Figure 4.3.

As describes earlier, all classes implements an interface and all relations between classes passes trough these interfaces. In the following paragraphs, the details of every class in the graph framework will be described.

Vertex A Vertex is very simple in our framework. It only consists of an identifier to make it uniquely determinable. A vertex could also simply have been represented by index numbers in an array, but as this thesis is written with the Object Oriented programming paradigm in mind, we have chosen to let a vertex be a class of its own.

Edge As the name implies, this is a representation of an edge in a graph. It contains information about the two vertices it connect (source and target), and a reference to the complement edge - that is, the edge which has opposite source

[3] JUNG - the Java Universal Network Graph Framework

and target respectively. An edge also contains information about its current flow and capacity which both can be set or retrieved.

EdgeContainer The `EdgeContainer` class contains a list of `EdgeInterface`'s and is in other word a wrapper. The reason for using a wrapper to contain `edgeInterface`'s is to have a standard way of collecting edges. If there is a need to change the data structure in which the edges is collected in, only one class should be changed.

Graph A Graph contains information of all vertices in the graph. The graph has association to several vertices and two vertices are explicitly defined as being the source and the sink. The graph also has references to several `EdgeContainer`-objects. There exist one `EdgeContainer` for each vertex, and `EdgeContainer`s are used to represents all outgoing edges for the corresponding vertex (the neighbours of the vertex).

4.2.1.2 The Algorithms

This section describes how the algorithm framework is designed. An UML-like illustration of the complete algorithm framework is shown in Figure 4.4.

From figure 4.4, it is shown that every algorithm implements the `MaxFlow AlgorithmInterface`. This makes it easy to develop new algorithms and use it in our existing framework. An abstract object `MaxFlowAlgorithm` represents the basic functionality in a Maximum Flow algorithm without the actual algorithm. The packages `algorithm.reference` and `algorithm.sequentiel` contains algorithms which are extending by the `MaxFlowAlgorithm` and will not be discussed further as the title of the classes should be self explaining.

The `algorithm.parallel`-package contains all the parallel implementations of the algorithms. In this package, some classes are introduced:

PushRelabelParallelWorker This class acts as the worker mentioned in the load balancy strategies discussed in 2.2.2. It performs the relabel and push operations and because it extends a `thread` it can have multiple instances running simultaniously.

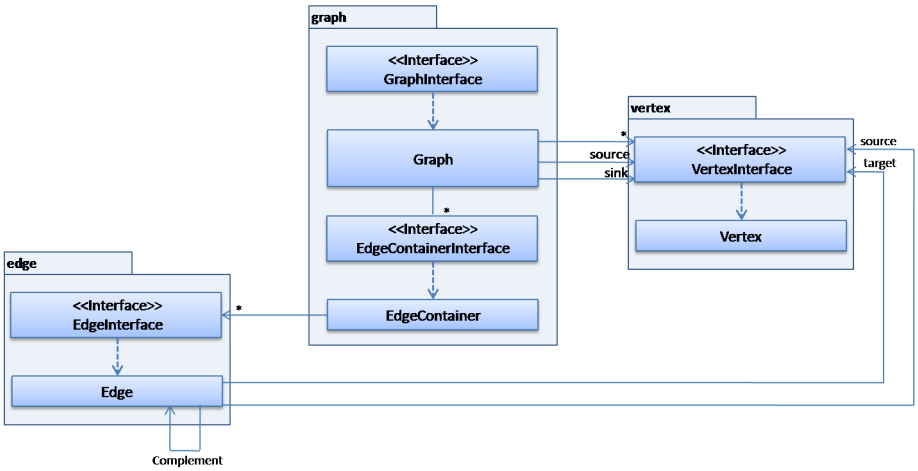

Figure 4.3: An illustration of the Graph Framework

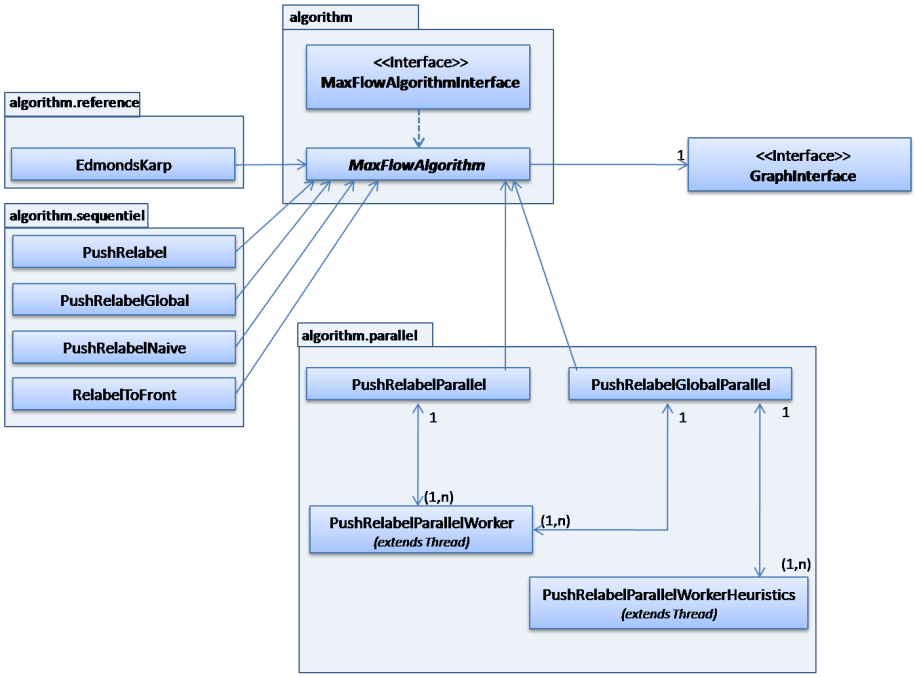

Figure 4.4: An illustration of the structure of our Algorithms

PushRelabelParallelWorkerHeuristics The purpose of this class is to do the global heuristics described in Section 3.2.2.3 with a Breadth First Search. It extends `thread` so it can run simultaniously with other processes.

PushRelabelParallel Works as an "organizer" or manager which creates a number of worker threads and "distribute" the subproblems to them. The worker threads are represented by `PushRelabelParallelWorker`'s and the number of associated workers are defined by the number of processes chosen.

PushRelabelParallelGlobal The "organizer" for the Global Heuristics implementation of the Push Relabel algorithm. As shown on figure 4.4 the class has several *PushRelabeParallelWorker*'s just like the *PushRelabelParallel*-class, but because of the heuristics, it references a *PushRelabelParallelWorkerHeuristics* as well.

4.3 Implementation

This section will describe important details on the implementation of the algorithms. We will not go in details of the complete implementation but only describe the significant parts. For details on the complete source code see Appendix C.

4.3.1 General implementation overview

Overall there has been a focus on using fast data structures which ensures good performance, but at the same time using data structures which has great flexibility and makes use of all the benefits of Object Oriented Programming. Throughout the implementation there has been a tradeoff between performance and flexibility, e.g. if the choice was to choose between a standard `Array` or an `ArrayList` the `ArrayList` would be selected even though it has a slight drop in performance. This is because the `ArrayList` does not have a fixed size, and therefore makes it easier to add objects.

As mentioned earlier all algorithms inherit from the same abstract class. This class is illustrated in figure 4.5.

The method `ComputeMaxFlow()` of the `MaxFlowAlgorithm` class is abstract,

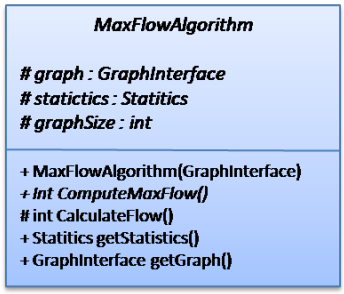

Figure 4.5: The abstract class which all algorithms inherits from.

because this implementation differs for the different kind of Max Flow algorithm. The `CalculateMaxFlow` method, on the other hand, is implemented directly in the abstract class, because no matter how the maximum flow is computed, the result is always extracted from the graph the same way, that is, to look at the amount of flow going out of the source. The `GetStatistics()` method is for benchmark-testing and documentation for our Statistics-framework can be seen in Appendix D.

The Edmonds-Karp reference-implementation also inherits from the `MaxFlow Algorithm`'s class but as it is simply a reference implementation, this section will not go into futher details. One should refer to the source code for more details on the reference implementations.

4.3.2 Common Push Relabel implementation details

All Push-Relabel algorithms have some common behavior and implementation choices, which will be described in this section.

Initialization The initialization of the graph in the different Push Relabel algorithms is almost identical (small changes can occur because of different data structures). The initialization method is straight-forward and does exactly what is described in Section 3.2.2.2. The method for initialization is called `initialize()`.

Excess flow and labels Common for all algorithms are that information about vertices and edges which only is relevant for the Push Relabel algorithms,

are kept in a central place rather than being kept "inside" the vertex or edge itself. The main reasons for this is to only keep general data at the vertices and edges and thereby making the graph framework more generic for use in the future. If some information is relevant for all Maximum Flow algorithms, such as flow and capacity, it will naturally be kept inside an edge.

In all of the push relabel algorithms, two `HashMap`'s are created, both with `VertexInterface` as keys and `Integer` as value. These `HashMap`'s represents excess flow and labels respectively. The reason for selecting the `HashMap` data structure is to make the lookup of values for the different vertices and edges easier.

Push/Relabel-Methods The push and relabel methods are almost identical among the different algorithms. They are implemented straight forward as describes in Section 3.2.2.1. Small deviations among the implementations occur because of different data structures.

4.3.3 Sequential implementation details

This section will describe the implementation of the sequential Push Relabel algorithms.

4.3.3.1 Naive Push-Relabel

This implementation is the most naive and straight forward implementation and does not use the *discharge*-method (as mentioned in Section 3.2.2.2). It contains `Java`'s `Queue` containing all active vertices. In the initialization this queue will be populated with the neighbors of the source and the algorithm will then process the vertices by the *First In First Out*-principle.

The algorithm implements the method `int ComputeMaxFlow()` which works as the *main-method* which is implementing with a `while`-loop which runs until no active vertices exists in the queue. A helper method has been implemented called `void performOperation(VertexInterface u)` which determines if a push or relabel operation should be made.

The push- and relabel-methods are implemented straight forward as described in Section 3.2.2.1.

4.3.3.2 Push Relabel

This implementation is almost the same as the naive Push Relabel algorithm (Section 4.3.3.1), but it discharge's a vertex completely before doing any relabeling. The list of active vertices is again implemented by `Java`'s `Queue` and the method `ComputeMaxFlow()` is still implemented with a `while`-loop which runs until no vertex is on the queue.

Important methods:

void discharge(VertexInterface u) This is the discharge-method which is called in each iteration of the loop describes above. It takes the vertex from the argument and process it until it has no excess flow. A `while`-loop iterates trough all the neighbors of the vertex u, and pushes flow to them, as long as the the excess flow of u is greater than zero. If it is not possible to push flow to any of u's neighbors, u is relabel and the push operations proceeds again.

4.3.3.3 Push Relabel To Front

This implementation is almost identical to the Push Relabel-implementation mentioned above and the strategy of finding the maximum flow is identical. The main difference is, that the `discharge`-method will not iterate until the vertex has no excess flow, but only until no push operation is possible.

After iterating trough all neighbors it will do a relabel on the vertex if it has an excess flow. If the new label is greater than the old, it will be put in front of the list and be processed first.

4.3.3.4 Push Relabel Global Heuristics

This implements the heuristics described in section 3.2.2.3. The implementation is much like the implementation described in Section 4.3.3.2 with one addition to the `ComputeMaxFlow()`-method. The Push Relabel algorithm with heuristics checks if the number of discharge operations performed is greater than the graph size and if this is true it performs a global relabeling. The addition to the `ComputeMaxFlow()`-method can be seen in listing 4.1.

```
1 if(noOfDischarge >= this.graphSize)
2 {
3     this.globalRelabeling();
```

```
4       noOfDischarge = 0;
5   }
```

Listing 4.1: Global Heuristics

The `globalRelabeling()` method applies the heuristics to the graph and is implemented as a breadth first search as described in section 3.2.2.3.

4.3.4 Parallel implementation details

The implementation of the parallel algorithms will be discussed in this section. The different implementations of the parallel push relabel algorithm have a lot of functionality in common. This section will describe these commonalities as well as the differences between the algorithms.

Locking vertices If a vertex is being processed, it should not be accessed by other threads. The solution to this, is to have a central `Map<VertexInterface, Lock>` to keep track of which vertices are being processed. The Java `Lock` is an `Interface`, and we have used `ReentrantLock` when implementing. This can be seen as a Binary Semaphore described in Section 2.3.2. By using a `Map` we make a tradeoff in the performance area, but avoids the risk of inconsistency which could occur if a normal array was used.

Determine when the computation is done When doing concurrent programming it is not always easy to determine when the computations are actually finished. When the main-class creates all the threads it simply waits until signaled that the computation is done. A solution could be to have a flag in the main class represented as a `boolean` data type and when a thread finds out that the computation is done, it will change the value of this flag. This solution however, will result in a *busy-wait* which will result in a decrease in performance. A better solution, and the one we chose, is to implement it as a `Semaphore` which is initialized to zero. When all threads have been created, the main-class performs an `Acquire`-operation on the semaphore and therefore does nothing until a `Release` operation is used on the semaphore. When a thread computes that the algorithm is done, it therefore simply releases this semaphore and the main-class is woken up and terminates the program.

The way a thread finds out if the computation is done, is implemented in the following way:

```
1 if(idleThreads == this.organizer.getNoThreads())
2 {
3     this.organizer.computationIsDone();
4 }
```

Listing 4.2: Computation is done

In words, listing 4.2 checks if the number of threads which has no tasks is equal to the total number of threads. If this is the case it means that there is no more work left and thus the algorithm is done. The worker thread then signals the semaphore which the organizer waits at and the program terminates.

Creating and stopping threads When the algorithm is being started it has to create a number of threads. This is done straight forward because the `Worker`-classes all inherit from the `Java Thread` class. When stopping the threads, we have a method in the organizer-class called `stopThreads()` which interrupts all threads. The threads thereby throws `InterruptedException`s which are then caught inside the threads and terminates them.

4.3.4.1 Mapping

The actual mapping of tasks to the different worker threads can be done in many ways, and should be considered very thorough and is essential for the performance.

In section 2.2.2 three possible mapping strategies were proposed and these have been implemented in the following way:

S1 This strategy is the most simple to implement and uses the Manager/-Worker mapping-scheme. It uses `Java`'s `ConcurrentLinkedQueue` which is an efficient *Wait-free*-algorithm described in Section 4.1.1.3 and minimizes the use of access control-management. The task-mapping is very simple: The workers gets an active vertex from the manager, process it and puts it back onto the Queue in the manager. Because the tasks are so small, some overhead will occur from the communication between the manager and worker. The implementations using this strategy are named `PushRelabelParallel2` and `PushRelabelParallelWorker2`.

S2 This strategy also uses the Manager/Worker scheme, but instead of only fetching one task from the shared Queue it receives a batch of tasks. This is done to minimize the communication overhead between the threads.

In this strategy a normal FIFO-Queue is being used because the access to the queue will not be as congested because of the batches of vertices. Each worker has two queues - one with active vertices and and one with processed vertices. Whenever the output-queue has reached the batch-size, the thread sends the vertices back to the manager. The implementations using this strategy are named `PushRelabelParallel3` and `PushRelabelParallelWorker3`.

S3 This is the only implementation which uses the Decentralized Scheme. When initializing and creating the threads all active vertices in the preflow are assigned equally to all the workers. From there, all workers will mainly put newly found active vertices on their own local queue an process them. When a worker runs out of active vertices it goes into idle-mode. When a non-idle threads discovers a new active vertex while there are other threads idle, it will add the vertex to a global queue instead of its own local queue. This is done in the `addOutput(VertexInterface)` method which can be seen in Listing 4.3. This scheme ensures that there is only minimal communication between the different threads.

```
1     protected void addOutput(VertexInterface u)
2     {
3         idleLock.lock();
4         if(idleThreads }= 0)
5         {
6             this.organizer.addActiveVertex(u);
7             idleQueue.signalAll();
8         }
9         else
10        {
11            this.localActiveQueue.add(u);
12        }
13        idleLock.unlock();
14    }
```

Listing 4.3: Global Heuristics

In theory, strategy number three is the fastest as the communication between threads, which often is the bottleneck in parallelization, is minimized. The practical performance of the three strategies are tested in Section 5.4.1.

4.3.4.2 Implementation of Global Heuristics

The implementation of the Parallel Push Relabel with Global Heuristics is almost identical to the implementation without heuristics. The only addition is that when creating all the threads an additional thread is being created - the `PushRelabelParallelWorkerHeuristics` whose only job is to repeatedly compute the Global Heuristics. Instead of running the heuristics after a fixed

number of discharge-operations as in the sequential version, it is running constantly. The main reason for this is because of a more simple implementation and in future versions the heuristic function should also be parallelized.

4.4 Test

When developing and implementing an algorithm it can be hard to verify and prove that the algorithm performs as it should. In the case of this thesis, the focus is on the parallelization process and performance and not necesarrily on proving correctness. The push relabel algorithm has been proven to be correct in earlier studies [10] and these results are relied on in this thesis. Futhermore, when developing a concurrent algorithm, it can be even harder to verify if the program complies with the stated safety and liveness properties. One way to do this is by using a tool like the SPIN verifier [13], to actually prove that the properties hold at all times, but this is out of the scope of this thesis.

That said, the algorithms has been tested extensively during the work of the thesis. The testing has mainly been performed using reference algorithms and graphs with known maximum flows. All test-graphs has been tested using both the reference algorithms, and the developed algorithms, and in all cases the maximum flow has been the same. The test-graphs are large in size and quite different from each other, and to some extend, this ensures that the developed algorithms are performing correctly.

If the time had allowed it, unit testing of all important methods should have been performed, but this was outside of the scope of this thesis.

Chapter 5

Experimental Results

In this chapter we present the results of the experiments perform during the work of this thesis. This includes a description of the test setup and a discussion of the results obtained during the experiment.

5.1 Hardware and programming language

The experiments were conducted on two different setups:

- **SYSTEM1:** Lenovo Thinkpad T61 with one 2.16 ghz Intel Core2 Duo processor with 2 cores. The processor is equipped with 4 MB level 1 cache. The system is running Windows XP SP3 and it is equipped with 2 GB of memory. The system is a using the Shared memory model as described in Section 2.1.
- **SYSTEM2:** Sun Fire E25K (newton.gbar.dtu.dk) with 72 UltraSPARC IV+ dual-core CPUs (1800 MHz/ 2 MB shared L2-cache, 32 MB shared L3-cache). The server is running Solaris 10 operating system and is equipped with 416 GB of memory. The system is a using the Shared memory model as described in Section 2.1.

The purpose of *SYSTEM1* is to test the parallel performance of the developed algorithms on a general purpose PC or laptop. *SYSTEM1* only has 2 cpu cores and it would have been preferable to have 4 or 8 cores instead. This, however, was not available at the time of the experiment.

The purpose of *SYSTEM2* is to test the parallel performance of the developed algorithms on specialized high performance computer with a high number of cpu cores.

The developed algorithms has been implemented using the Java programming language and has been compiled using both Java version 1.5.0_13 and 1.6.0_05. The reason to use two different compilers is to investigate a possible performance difference.

5.2 Input graphs

This section describes the the graph format used in the thesis as well as the different graph types used for testing.

5.2.1 Graph Format

To have some common ways of handling and storing graphs, all graphs stored in our framework inherits the definitions from the DIMACS Challenges [7]. The DIMACS format is a way to represent graphs in plain text-files and has the following specifications for lines in the file:

Comment Lines "`c this is a comment`" - Comment lines can appear anywhere and are ignored by programs.

Problem Line "`p sp n m`" - The problem line is unique and must appear as the first non-comment line. This line has the format on the right, where n and m are the number of nodes and the number of edges, respectively.

Edge descriptor lines "`a U V W`" - Edge descriptors are of the form on the left, where U and V are the tail and the head vertex id's, respectively, and W is the edge weight.

Node Definition "`n U W` - Defines a node to either a source or a sink. `U` determines if a source or a sink is being defined and must be either the character s or t for source or sink respectively. `W` tells which node (the ID-number) has to be defined.

5.2.1.1 Graph Types

The developed algorithms has been tested on several kinds of graphs to examine how the performance is influenced by adjusting different factors. The graph types are inspired by the problem families used in the DIMACS[7] contest and are as follows:

AK graph The AK graph type was introduced by Boris V. Cherkassky and Andrew V. Goldberg [4]. A graph of this type is generated using a single parameter k and consists of $4k + 6$ vertices and $6k + 7$ edges. The AK graph type was developed to produce problems that are especially hard for the push-relabel and Edmond-karp algorithms. The acronymm *ak* will henceforth be used to designate AK graphs.

Grid graph The Grid graph is composed of a grid of $n_1 \cdot n_2$ vertices. Each vertex is connected to its neighbors in the grid with an edge. The capacity of an edge is selected randomly in a specified range. The source is located in the top left corner of the grid and the sink is located in the lower right corner. It is possible to specify a probability value used to determine if a vertex should have an edge to one of its neighbors. An example of a grid graph can be seen in figure 5.1. Two subclasses of this graph type has been used in the experiment: *GridW*, in which the graph is 4 times as wide as it is long, and *GridL*, in which the graph is 4 times as long as it is wide.

Row graph The Row graph was introduces by Richard Anderson and Joao C. Setubal [2][1] and is composed of n_1 number of rows each containing n_2 vertices. Each vertex in a row is connected via edges to n_3 vertices in the next row. The capacity of each edge is randomly selected from a specified range. The source is connected via edges to each vertex in the first row and all vertices in the last row are connected via edges to the sink. All edges from/to the source/sink has a capacity specified by the maximum capacity possible between ordinary vertices. An example of the grid graph can be seen in figure 5.2. Two subclasses of this

[1] Originally known as Random level graph

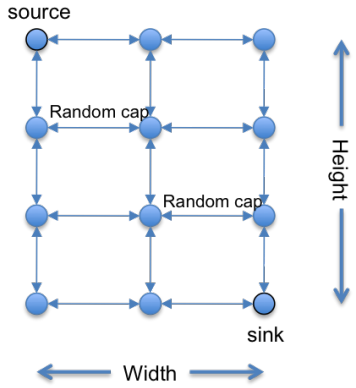

Figure 5.1: Example grid graph

graph type has been used in the experiment: *RowS*, in which there are a few number of edges between each each row (2 edges per vertex in a row), and *RowD*, in which there are many edges between each row (8 edges per vertex in a row).

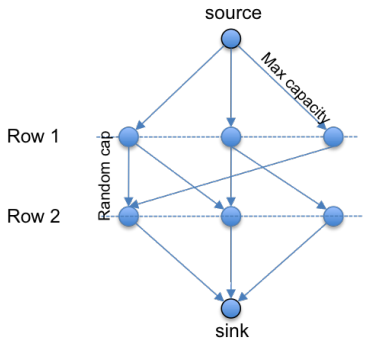

Figure 5.2: Example row graph

5.3 Test principles

The following methods and principles were used during the experiment:

- For each graph type several graphs of different sizes was generated. See

Appendix A for a complete description of all the graphs used in the experiment.

- For each graph instance, 3 runs with each algorithm was conducted and the running time is calculated as a mean of these 3 runs.
- For each parallel algorithm all graph instances were tested using 1/2/4 threads on *SYSTEM1* and 4/8 threads on *SYSTEM2*[2].
- The running time of each run does not include input and output time.
- All tests were performed with the java virtual machine flag `-XX:AggressiveHeap`. This flags ensures that the virtual machine allocates a sufficient amount of heap space for the compution.
- The running times of the algorithms are calculated using the Java command `System.currentTimeMillis()`.
- The speedup gained from parallelization is defined as the running time of the sequential implementation divided by the parallel running time.

5.4 Test results and discussion

During the experiment several tests have been performed to verify the different aspects of parallelization. The parallelization of the simple push relabel algorithm is first tested using different load-balancing strategies to determine the most efficient implementation and then the most efficient implementation is compared to the reference sequential algorithm. This is followed by a test of the more advanced push relabel algorithm with global heuristics in comparison with a reference sequential algorithm also with global heuristics. Following the individual testing of the algorithms is a final comparison between all the implemented algorithms. Both the simple and the more advanced algorithm are tested on both test systems to clarify if the underlying hardware and operating system has an impact on the performance.

5.4.1 Results of the simple push relabel algorithm

As mentioned in the analysis of the parallelization potential of the simple push relabel algorithm (see Section 4.1.1) different load-balancing algorithms/strategies should be considered. In the experiment three versions has been tested:

[2]The number of threads does not necessarily maps to the exact same number of physical processors.

One version of the decentralized scheme and two versions of the manager/worker scheme (one using a global standard queue and one using Java's datastructure `ConcurrentLinkedQueue` which eliminate the need for locking mechanisms). The running time of the different implementations can be seen in figure 5.3 (the tests have been performed with 2 threads on SYSTEM1).

Graph type	Vertices	Edges	Push Relabel (decentralized)	Push Relabel (manager worker 1)	Push Relabel (manager worker 2)
AK5	32774	49159	43.2	44.4	45.3
GridW5	19600	54969	465.9	569.1	393.2
GridL5	19600	54969	523.4	558.4	596.4
RowS5	10002	20000	45.5	62.0	51.4
RowD6	19602	155960	765.6	792.8	832.1

Figure 5.3: Test of different load-balancing strategies on SYSTEM1

It is clear from the results that the decentralized scheme is the most efficient load-balancing algorithm on most graph types, and in all of the following tests this is the scheme used. It is, however, interesting to see that the manager/-worker solution which utilizes the wait-free datastructure from JAVA is performing so well. This solution has been much easier to implement, and it could easily become very useful tool for developers.

The reason why the decentralized scheme outperforms the two other load balancing schemes, is probably because it minimizes the need for synchronization, as the worker threads rarely needs to access the global queue. Another plausible explanation of why the master/worker scheme is not as effective as the decentralized scheme, could be because threads would spend too much time waiting to acquire a lock on the shared queue. This, however, does not seems to be the case. After an extensive test performed using the Eclipse Test and Performance Tools Platform (TPTP)[3], it can be concluded that only a minimal time is spend in queues waiting to acquire locks. A graphical representation of the results gained from running the TPTP tool can be seen in figure 5.4, in which the two worker threads has been emphasised. It is clear from the figure that the two threads only spends a very minimal time waiting to acquire locks.

Another important thing to notice with regards to the results, is despite that the decentralized scheme cannot ensure in which order the active vertices are

[3] A performance suite developed by the Eclipse team which among other things provides the developer with tools to measure parallel performance. See http://www.eclipse.org/tptp/ for more details on the tool and its purpose

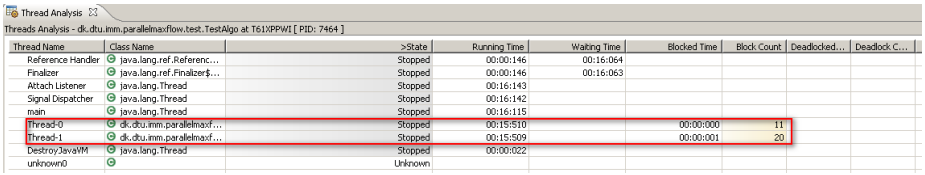

Figure 5.4: TPTP test results

processed this doesn't seem to have any great impact on the performance. Earlier studies [10] have shown that it is important to process vertices in either FIFO ordering or by processing the nodes with the largest labels first, but to some extend the results in this thesis shows that the order doesn't have to follow the FIFO ordering strictly to be effective. It should be mentioned that this has not been investigated any further in this thesis and is therefore not a conlcusive result.

After selecting the most efficient load balancing strategy the parallelized version of the simple push relabel algorithm was tested against a sequential version of the same algorithm and a reference implementation of the Edmonds-Karp algorithm. The test results can be seen in figure 5.5 and 5.6.

Graph type	Vertices	Edges	Edmond Karp	Push Relabel Sequential	Push Parallel (2 threads)	Push Parallel (4 threads)
AK	32774	49159	459.2	61.6	43.2 (1.42x)	42.1 (1.46x)
GridW	19600	54969	1.4	877.3	465.9 (1.88x)	463.7 (1.89x)
GridL	19600	54969	2.6	953.1	523.4 (1.82x)	520.8 (1.83x)
RowS	10002	20000	27.2	87.9	45.5 (1.93x)	44.2 (1.99x)
RowD	19602	155960	853.2	1447.7	765.6 (1.89x)	761.1 (1.90x)

Figure 5.5: Push Relabel test result on SYSTEM1

The most important results regarding the purpose of this thesis is that the parallelized version of the algorithm is significantly faster than the sequential algorithm on SYSTEM1. Running on a two cpu core machine with two threads, the algorithm is running close to twice as fast as the sequential algorithm on most of the graphs. This confirms that the parallel potential of the push relabel algorithm is high, and that the implementation in this thesis is efficient. By increasing the number of threads from two to four a small performance increase is detected on some graphs, despite that SYSTEM1 only is equipped with 2 cpu cores. This is mainly due to the fact the the two cores are utilized more effectively by using more threads than cores. The reason for this is that each

Graph type	Vertices	Edges	Edmond Karp	Push Relabel Sequential	Push Parallel (4 threads)	Push Parallel (8 threads)
AK	32774	49159	823.5	77.3	119.3	304.8
GridW	19600	54969	3.4	1163.1	Timeout	Timeout
GridL	19600	54969	3.8	33.0	Timeout	Timeout
RowS	10002	20000	47.6	116.5	387.0	616.6
RowD	10002	79400	398.5	410.3	1646.6	1373.1

Figure 5.6: Push Relabel test result on SYSTEM2

tread often waits on I/O operations and during waiting periods the thread does not use any computational power.

While SYSTEM1 performed as expected a more unexpected result of the test is that the parallelized algorithm was actually slower on many graph types than the sequential algorithm on SYSTEM2. This performance decrease requires a more extensive explanation and is addressed in Section 5.4.4.

A second interesting result from the test is that both the sequential and the parallel implementation of the push relabel algorithm is significantly less efficient than the reference Edmonds-Karp algorithm, on almost every graph type. Only the AK graph is performing better on the push relabel algorithm. Even tough the worst case running time of push relabel is better than the worst case running time of Edmond Karp, the practical performance is much worse. This, however, was expected and is on par with the findings of A.V. Goldberg and R.E. Tarjan in [10].

5.4.2 Results of the push relabel algorithm with global heuristics

The test for the most efficient load balance strategy in Section 5.4.1 also applies for the push relabel algorithm with global heuristics, and the test for an optimal load balancing strategy has therefore not been repeated. The algorithm has been tested against a sequential version of the same algorithm and a reference implementation of the Edmons-Karp algorithm. The test results can be seen in figure 5.7 and 5.8.

As with the simple push relabel algorithm, this parallelized algorithm with global heuristics outperforms the sequential implementation on SYSTEM1. The

Graph type	Vertices	Edges	Edmond Karp	Push Relabel Heuritic	Push Parallel heuristic (2 threads)	Push Parallel heuristic (4 threads)
AK	32774	49159	459.2	194.7	105.8 (1.84x)	103.9 (1.87x)
GridW	160000	638000	69.5	5.3	3.2 (1.66x)	3.1 (1.70x)
GridL	160000	638000	94.0	8.3	4.3 (1.93x)	4.3 (1.93x)
RowS	40002	80000	566.8	7.9	4.5 (1.76x)	4.3 (1.84x)
RowD	40002	318800	3661.9	12.1	6.5 (1.86x)	6.4 (1.89x)

Figure 5.7: Push Relabel with heuristics test result on SYSTEM1

Graph type	Vertices	Edges	Edmond Karp	Push Relabel Heuritic	Push Parallel heuristic (4 threads)	Push Parallel heuristic (8 threads)
AK 5	32774	49159	823.5	271.9	274.2	274.0
GridW	160000	638000	390.1	6.8	12.7	9.3
GridL	160000	638000	441.5	10.6	11.4	12.4
RowS	40002	80000	969.3	10.7	11.8	11.0
RowD	19602	155960	1730.1	5.6	5.5	4.3

Figure 5.8: Push Relabel with heuristics test result on SYSTEM2

algorithm running on two cpu cores is not twice as fast as the sequential implementation but it is still significantly faster. The reason why the performance increase is not as good as the optimal result, is probably because the heuristics function has not been parallelized. We decided to let one thread do the global relabeling continuously but we could probably have gotten better results by using the method discussed by R. Anderson and J.C. Setubal in [2]. This would, however, have required a much more elaborate synchronization scheme and would have complicated the implementation a lot.

As it was the case with the simple push relabel algorithm the parallelized version did not perform very well on SYSTEM2. This is addressed more thoroughly in Section 5.4.4.

5.4.3 Recapitulation of the results of the implemented algorithms

All graphs has been tested with all developed algorithms and a table showing all running times can be found on the attached CD (See appendix C). The table shows the running time for each algorithm on a single graph of each graph type.

It was the conclusion of B.V. Cherkassky and A.V. Goldberg in [4] that the push relabel algorithm with global relabeling outperforms all known maximum flow algorithms and this is on par with the results in this thesis. As seen in figure 5.7, even the sequential implementation of the algorithm is performing very well and with the performance increase gained from the parallelization it is even faster.

It is also clear from the results, that the parallelization potential of the push relabel algorithm is high, even with a standard object oriented programming language. The "standard" programming language of almost all previous research has been C or C++, and it was therefore one of the greatest concerns in this thesis, that the widely accepted parallelization procedure (as specified in [11, chapter 3] would not apply to a modern object oriented language. This, however, has too some extend been proven wrong with the work performed during this thesis, as the results are close to the results achieved in previous studies (e.g. [2]). Due to the problems experienced testing the algorithms on SYSTEM2 (see Section 5.4.4), more extensive testing should be done on multi-core systems with more than two cores to ensure that the results are correct and conclusive. The performance problems on SYSTEM2 also leads to the conclusion that parallelization with the widely accepted parallelization procedure still is very error prone, and it is hard to make a general parallelization strategy. One still has to take the underlying architecture into account when parallelizing, even with

a modern object oriented language like Java.

5.4.4 Advantages and disadvantages of using Java

This section discusses why Java was selected as the programming language as well the good and not so good experiences gathered throughout the work of this thesis.

5.4.4.1 Advantages

One of the major goals of this thesis, has been to explore if a modern object oriented programming language is suited for parallelizing classical algorithms. Java was selected due to its recent updates with regards to its concurrency libraries, and because it is a widely used object oriented language. Microsoft's C# was considered as well, but no multi-core system with more than two cores capable of running C# was available to us during the thesis work.

The actual implementation of the algorithms using Java was to some extend rather painless, as Java provides all of the well known synchronization primitives out-of-the-box. The object oriented aspect of Java also eased the implementation, as it was easy to share functionality between algorithms using inheritance.

One of the major concerns in the beginning of the thesis work, was that the selected decomposition into sub tasks, might be too fine-grained for Java. The performance of the synchronization primitives in Java was unknown, and it was not known if the many synchronization operations in the algorithms would affect the performance. This has partly been disproved by the results from SYSTEM1, which shows that synchronization primitives provided by Java, are very efficient and does not affect the performance. The reason why the word partly is used is that the synchronization operations seems to affect the performance in different ways on different operating systems as described in Section 5.4.4.2.

With the recent updates of Java (1.5 and 1.6), several new data-structures have been implemented to compete with the classical synchronization mechanisms. One of them, the `ConcurrentLinkedQueue` has been tested in this thesis and it provided some promising results. Even though it was not as effective as fine-grained synchronization using locks, it was very easy to implement, and thereby minimizes the number of typical errors experienced when trying to parallelize. Another promising data structure in Java is the Atomic variables found in the

package `java.util.concurrent.atomic`. These variables can be incremented and decremented atomically, and could potentially be very useful when writing concurrent programs, as they minimize the need for synchronizing access to critical regions.

5.4.4.2 Disadvantages

Java has in the recent versions improved the parallel performance a lot, by utilizing the native synchronization mechanisms in the operation system on which they are executing. This, however, proved to be a major problem when testing the algorithms developed in this thesis. While the algorithms performed well on *Windows* systems (SYSTEM1), they were actually slower than the sequential algorithms on both *Mac OS X* and *Solaris 10*.

This performance decrease lead to much confusion during the development, as the first thought was that the algorithms was inefficiently implemented. After trying the different synchronization mechanisms available, it was discovered that the test setup on the different systems was not identical. The Java version was actually different on the systems as SYSTEM1 was using Java version 1.6 and SYSTEM2 as well as an Apple laptop were using version 1.5. One of the major differences between the two versions is in fact the performance of the synchronization primitives, and this could explain why the algorithms performed so differently on the two systems.

The next obvious step was to use Java version 1.6 on all systems, and this resulted in a significant performance boost on the Apple laptop but not on SYSTEM2. The reason for this is at this time still unknown, but one possible explanation could be that the Java virtual machine for Solaris implements the different synchronization mechanisms in an inefficient way. This, however, seems unlikely as Solaris is developed by the same company as Java (SUN), and one would think that they would know how to implement efficiently on their own operating system. Due to time constraints it has not been possible to further investigate this problem, but if one would try to find a solution, it should be noted that by using the TPTP framework in eclipse, it has been shown that the threads does not spend very much time waiting for other threads to release locks on a shared resource, and it is therefore more likely, that the performance problem is related to the actual acquire and release operations in the Solaris operating system.

Another plausible reason why the algorithms did not perform well on SYSTEMS2, could be related to the fact that the Java garbage collector requires special configuration on systems with many cores (10+). We have tried to per-

form some of the configurations mentioned in [14] but mainly because of the limited time available a working solution to the problem has not been found.

Chapter 6

Conclusion

The objective of this thesis was to analyze, design and implement a parallelized version of a classical sequential algorithm suited for use on a standard multi-core computer. The algorithm selected as the parallelization subject has the Push Relabel algorithm. This algorithm is one of the most effective Maximum Flow algorithms available and has in previous studies provided some promising parallelization results.

The Push Relabel algorithm has in this thesis been thoroughly analyzed using the widely accepted parallelizing procedure described in [11], and a parallelized design has been proposed. The proposed design has been implemented in different versions with and without heuristics, all using the object oriented programming language Java. The thesis describes this development process in details, and comments on the experiences gathered using Java as the tool for parallelization.

To test the correctness of the developed algorithms, reference implementations of the sequential Push Relabel algorithm and the Edmonds Karp algorithm has been implemented. All Maximum Flows calculated by the developed algorithms has been compared to the results of the reference implementations, and at the time of publication no errors related to the calculated results have been found.

The performance of the developed algorithms has been measured using several

different graph types on two different test setups. The results gathered from the standard dual-core test setup were promising, and the parallelized algorithms were running almost twice as fast as the sequential algorithms using two threads. This is close to the optimal, and shows that the parallelization potential of the Push Relabel algorithm is good. The results gathered from the second test setup, which consisted of a server with many cores (50+), was less promising, and the parallelized algorithms often performed worse than the sequential algorithms. A possible explanation for this performance decrease has been given in Section 5.4.4.

Common for all test results is that the general Push Relabel algorithm is not performing well compared to the Edmond Karp algorithm, despite a better theoretical running time. With heuristics, however, the algorithm is much faster and outperforms the Edmond Karp algorithm on most graph types. This is on par with earlier studies.

The thesis leaves room for future work, and especially the performance problems on multi-core systems with more than a few CPU cores, could be an interesting field of study. Another interesting extension to this thesis, would be to use a tool like the SPIN verifier to prove the different concurrent properties of the developed algorithm.

As a conclusive remark, we believe that the requirements of this thesis has have fully achieved. The experiences gathered with regards to parallelization procedures and parallelization using object oriented languages, provides some interesting results and a good foundation for future work.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 79–81, 2000.

[2] Richard Anderson and Joao C. Setubal. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *J. Parallel Distrib. Comput.*, 29(1):17–26, 1995.

[3] David A. Bader and Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In Michael J. Oudshoorn and Sanguthevar Rajasekaran, editors, *ISCA PDCS*, pages 41–48. ISCA, 2005.

[4] Boris V. Cherkassky and Andrew V. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical report, Stanford, CA, USA, 1994.

[5] Thomas H Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass., 2nd ed edition, 2001.

[6] Edsger W. Dijkstra. The structure of "the"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[7] Dimacs implementation challenge, http://www.dis.uniroma1.it/~challenge9/, 06 2008.

[8] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[9] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 1st ed edition, 2006.

[10] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[11] Ananth Grama. *Introduction to parallel computing*. Addison-Wesley, Harlow, England, 2nd ed edition, 2003.

[12] Per Brinch Hansen. Monitors and concurrent pascal: a personal history. *SIGPLAN Not.*, 28(3):1–35, 1993.

[13] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st ed edition, 2003.

[14] Tuning garbage collection, http://java.sun.com/docs/hotspot/gc5.0/, 06 2008.

[15] D. R. Fulkerson L. R. Ford. Maximal flow through a network. *Canadian Journal of Mathematics*, (8):399–404, 1956.

[16] Hans Henrik Løvengreen. *Basic Concurrency Theory*. Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Denmark, 1.1 edition, 2005.

[17] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM.

[18] Joshua O'Madadhain, Danyel Fisher, Scott White, and Yan-Biao Boey. Java universal network/graph. World Wide Web electronic publication, http://jung.sourceforge.net/, 2006.

[19] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. Seti@home—massively distributed computing for seti. *Comput. Sci. Eng.*, 3(1):78–83, 2001.

Appendix A

Graphs

All graphs used for testing in this thesis is listed below. The actual graphs can be found on the included CD in the folder *graphs*.

AK Graph #	Number of nodes	Number of edges	Parameter
1	2054	3079	512
2	4102	6151	1024
3	8198	12295	2048
4	16390	24583	4096
5	32774	49159	8192
6	65542	98311	16384

Figure A.1: AK graphs

GRIDL Graph#	Number of nodes	Number of Edges	Width	Heigth
1	1156	3182	68	17
2	2500	6880	100	25
3	4900	13604	140	35
4	10000	28062	200	50
5	19600	54969	280	70
6	40000	112936	400	100
7	78400	312200	560	140
8	160000	638000	800	200

Figure A.2: GridL graphs

GRIDW Graph#	Number of nodes	Number of Edges	Width	Heigth
1	1156	3182	17	68
2	2500	6880	25	100
3	4900	13604	35	140
4	10000	28062	50	200
5	19600	54969	70	280
6	40000	112936	100	400
7	78400	312200	140	560
8	160000	638000	200	800

Figure A.3: GridW graphs

ROWD Graph#	Number of nodes	Number of edges	Width	Heigth	Edges per Node
1	627	4850	25	25	8
2	1227	9590	35	35	8
3	2502	19700	50	50	8
4	4902	38780	70	70	8
5	10002	79400	100	100	8
6	19602	155960	140	140	8
7	40002	318800	200	200	8

Figure A.4: RowD graphs

ROWS Graph#	Number of nodes	Number of edges	Width	Heigth	Edges per Node
1	627	1250	25	25	2
2	1227	2450	35	35	2
3	2502	5000	50	50	2
4	4902	9800	70	70	2
5	10002	20000	100	100	2
6	19602	39200	140	140	2
7	40002	80000	200	200	2

Figure A.5: RowS graphs

Appendix B

GUI

To make it easier for testing and running the different algorithms, a Graphical User Interface (GUI) has been developed. This section will figure as a user manual for the GUI and explain the different areas of it.

B.1 Starting the GUI

The program can be found on the attached CD in the folder GUI. The program has been compiled into a .jar file using the 1.6.0 version of Java and this version is required for execution. The GUI can be launched by double-clicking on the jar file with the filename maxflow.jar. If this isn't possible the program can be launched by running the following command `java -jar maxflow.jar` provided that you have navigated to the correct folder.

If you intend to test the program on very large graphs the following command should be used to launch the program: `java -XX:+AggressiveHeap -jar maxflow.jar`. This command allocates as much memory to the program as possible which is often possible with large graphs.

It should be noted that the GUI has been optimized for use on Windows systems and is meant as a simple testing tool.

B.2 Navigating the menu

The GUI consists of three parts: *Benchmark*, *Run algorithms* and *Graph Generation*. The user can switch between these parts by using the menu tabs on top.

Figure B.1: The menu tabs of the GUI

B.3 Generating Graphs

In the GUI it is possible to generate graphs. The menu tab *Graph Generation* contains this. An example of the Graph Generation area can be seen in Figure B.2.

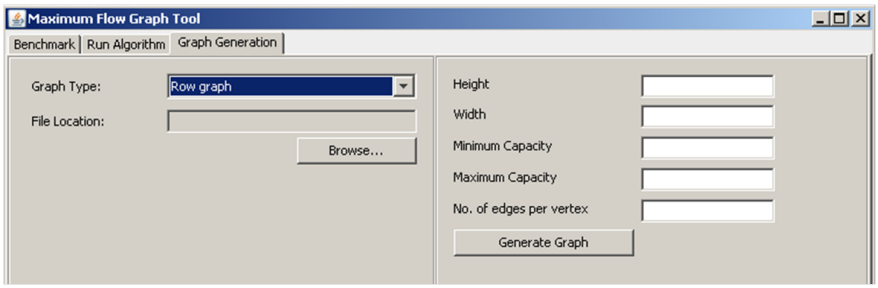

Figure B.2: Example of Graph Generation

To select what type of graph to generate, the drop-down menu *Graph Type* should be used. The panel to the right will look different with respect to which graph is chosen because they all have different properties. All fields must be filled out to be able to generate the graph.

Then, a location for the file containing the graph must be selected. This is done by pressing the *Browse* button and navigate to the desired location and write the filename. Any file-extension can be used.

When all fields are filled out correctly, the graph is ready to be generated which is done by pushing the *Generate Graph* button. A status message will appear and tell if the creation failed or succeeded.

B.4 Running algorithms

For a simple run of an algorithm this section should be used. By clicking the *Run Algorithm* menu tab, this section of the GUI will appear. An example of how the GUI looks, see figure B.3.

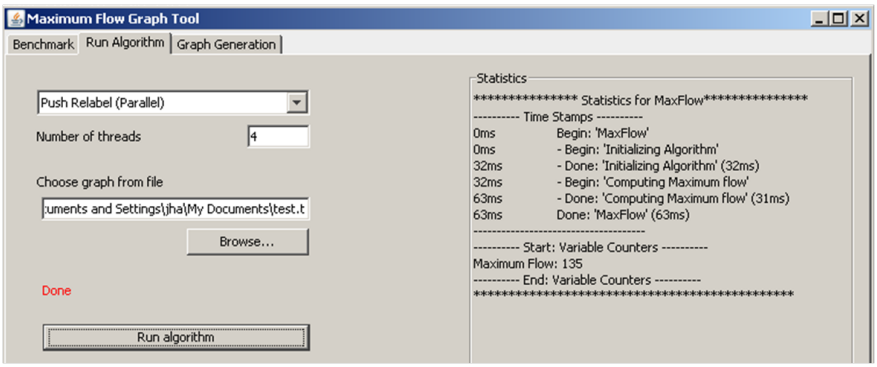

Figure B.3: Example of "Run algorithm" area after algorithm is done computing.

The desired algorithm should be chosen in the drop-down box to the left - if a sequential algorithm is chosen, the *Number of threads* text field will be disabled but with a parallel algorithm the user can specify how many threads should be created within the algorithm.

The desired graph is chosen by clicking the *Browse*-button and navigation to the file of wish. To run the algorithm on the desired graph, the *Run algorithm*-button should be clicked. Then some text will appear in the Statistics-window with information about the execution.

B.5 Benchmark testing

This area of the GUI allows the user to test multiple algorithms on multiple graphs many number of times with just one click and allow the user to compare

the running times and results of the different algorithms.

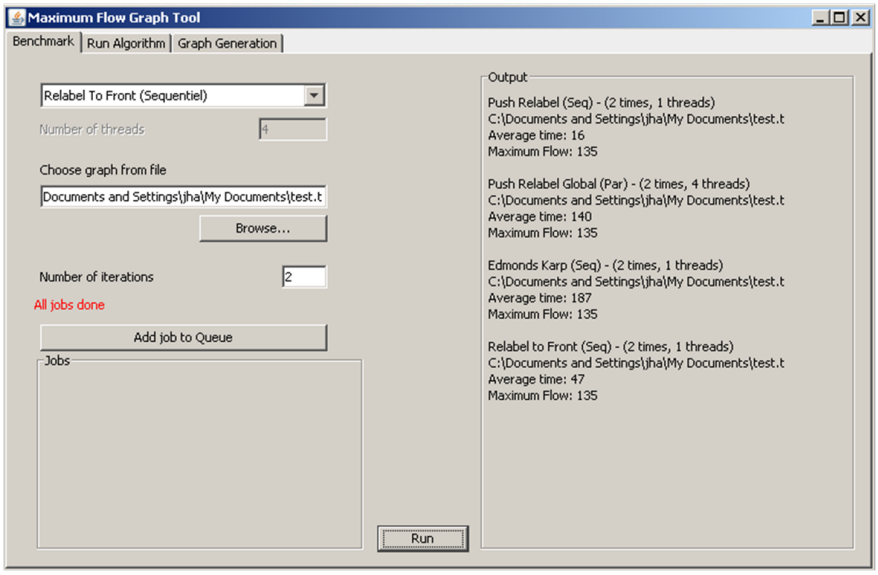

Figure B.4: Example of the "Benchmark" area.

An example of the Benchmark area can be seen in figure B.4. A benchmark consists of multiple jobs and a job is an algorithm type, a graph and a desired number of executions. A job is created the exact same way as described in Section B.4 and when all the fields are filled out, the *Add job to Queue*-button is clicked and the job will figure on the job-list at the button.

When all desired jobs are added to the queue, the *Run*-button in the bottom middle can be pushed and the benchmark testing starts. The program will be frozen until all iterations of all algorithms are done - if large graph are generated this can take a while.

Appendix C

Source Code

The source code of the developed algorithms as well as an electronic copy of the thesis can be found on the attached CD.

The content of the CD can also be acquired by contacting one of the authors by email (s052425@student.dtu.dk, s052905@student.dtu.dk) or by downloading it from http://www.student.dtu.dk/~s052425/bachelor.zip. The content of the CD is as follows:

Eclipse/ Folder containing the full source code as an eclipse project. Could be imported directly into eclipse by clicking on "File → Import → Existing Projects into Workspace".

Graphs/ Folder containing all the test graphs used in the thesis.

GUI/maxflow.jar The application as an executable jar-file.

Results/results.xls Microsoft Excel file containing all test results gathered throughout the thesis work.

Source/ Folder containing the full source code of the application. Ordered into folders which represents the package structure mentioned in Section 4.2.

Thesis/bachelor.pdf An electronic copy of the final thesis.

Appendix D

Statistics Framework

To make it easier to to measure the performance of the different algorithms, a statistical tool has been designed and implemented. The design and implementation will not be described in details but the main class and the important methods will be mentioned.

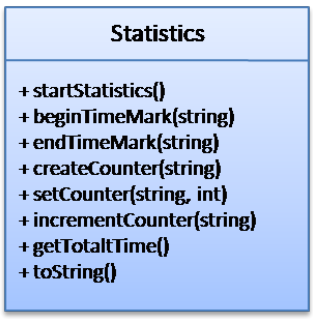

Figure D.1: Important methods of the Statistics-class

On Figure D.1 the main class and its important methods are shown. The statistics tool makes it possible to measure the performance of different areas of the algorithm and make sub-measurements to existing measurements alltogether. The framework is very easy to integrate in any algorithm. Besides performance-

measuremtent it can containt multiple counters so the output can tell how time the algorithm did a certain operation etc. Each method will be described here:

startStatistics() This method will tell the statistics that the recordings should start.

beginTimeMark(string) Sets a mark that a new measurement should be started. The string in the argument will be the title of the measurement.

endTimeMark(string) Stops the measurement with the title equal to the argument.

createCounter(string) Creates a counter where the title of the counter will be the string in the argument. The default start value of the counter is zero.

setCounter(string,int) Sets a counter to a value. Overwrites the current value.

incrementCounter(string) Increment the counter with the title given in the argument by one.

int getTotalTime() This returns the total time the statistics-method has been measurering.

toString() Returns a nice output of the totalt statistics including the counters.

An example of an output after using the tool looks like this:

```
***** Statistics for Example of Flow-Statistics!*******
---------- Time Stamps ----------
0ms Begin: 'Example of Flow-Statistics!'
15ms - Begin: 'Initialisering'
15ms - Done: 'Initialisering' (0ms)
```

```
46ms - Begin: 'Breadth First'
250ms - - Begin: 'Create Threads'
468ms - - Done: 'Breadth First' (422ms)
703ms - Done: 'Create Threads' (453ms)
781ms Done: 'Example of Flow-Statistics!' (781ms)
-------------------------------------
---------- Start: Variable Counters ----------
Number of push-operations: 200
Number of relabels: 300
---------- End: Variable Counters ----------
*********************************************
```

The use of indent text makes it easy to see if two measurements are running at the same time and makes it easy to see if the measurements finishes in the same order as they are started.