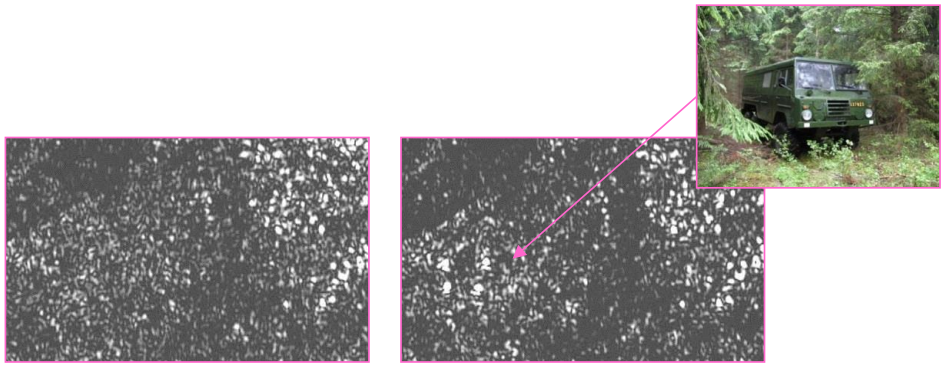
CHALMERS



Realization of efficient change detection in SAR images

Master of Science Thesis in
Computer Science: Algorithms, Languages and Logic

JOANNA ERIKSSON

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015

Realization of efficient change detection in SAR images
JOANNA ERIKSSON

Examiner: Bengt Nordström
Supervisor: Peter Damaschke

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone: +46 (0)31-772 1000

Cover: Image representing change detection. The left image is a reference image and the right image is a new image where trucks have been hidden in the forest. Image courtesy of FOI.

Department of Computer Science and Engineering
Gothenburg, Sweden. June 2015.

Acknowledgments

I would like to thank Saab Electronic Defense System for giving me the oppurtunity of doing this thesis for them. In particular I would like to thank my adviser at Saab, Anders Åhlander, for valuable help and feedback. Also, I thank Patrik Dammert for sharing his expertise on change detection. I would like to express my gratitude to Peter Damaschke for his comments that has improved this report. Finally, I am grateful to Anders Nordin and Hannes Sandahl for their critical review of this report.

Joanna Eriksson, Gothenburg, June 11, 2015

Abstract

There exist algorithms for detecting if a change has taken place in a land area by analyzing images over the land from two occasions. The process of detecting changes using these algorithms is called **change detection**. The current MATLAB implementation of the algorithms are too slow for running in a **real-time environment**. This thesis has the purpose of determining whether a real-time performance would be possible by translating from **MATLAB** to **C++** and parallelizing. The **parallelization** is performed with the API **OpenMP** and FFT (Fast Fourier Transform) computations are done with the C library **FFTW**. It was not previously known if it is possible to run this change detection implementation in real-time. Similar research has studied this problem for other types of images and other types of algorithms for change detection. The considered algorithms of change detection can be divided into five parts and within the time frame of the project there has been time to thorughly study the first step, called pie-filter or spatial band-pass filter and to do a brief and theoretical study of the second step, matching. The speedup of the pie-filter is shown to be about 9 times compared to the original MATLAB code. The maximum possible total execution time is the time it takes to take a SAR image of the ground. The pie-filter is executing in 10% of the maximum possible total execution time of the real-time change detection. This is a result that indicates a possible real-time execution. The report also contains an evaluation of different methods for performing FFT using FFTW. These methods are analyzed and it is found that for different levels of parallelization and input size, different methods are optimal to use. The results regarding FFT are important because they can be applied to other applications.

Contents

Abbreviations

SAR Synthetic Aperture Radar

CARABAS Coherent All RAdio BAnd Sensing

VHF Very High Frequency

GPS Global Positioning System

SPMD Single-Program Multiple-Data

MIMD Multiple-Instruction Multiple-Data

FFT Fast Fourier Transform

DFT Discrete Fourier Transform

API Application Program Interface

IFFT Inverse Fast Fourier Transform

1 Introduction

In many situations it is of great interest to be able to monitor vast land areas. One system for taking pictures of a big land area is CARABAS, which is short for Coherent All RAdio BAnd Sensing. For civilians, CARABAS can be used to determine the number of trees in a land area and their dimensions [25]. The military, on the other hand, can use CARABAS to monitor so that no new vehicles appear in a forest, e.g. hidden under foliage.

In order to take pictures of a big area with a decent resolution, it is necessary to have a very big antenna. Aperture is a measure of the "size". A large aperture gives a high resolution in angle which gives high resolution in the image. The aperture can synthetically be made bigger by putting a smaller antenna onto a moving platform [6], this is called SAR (Synthetic Aperture Radar) and is used by CARABAS. The data from the antenna is then processed like it came from one physically large antenna.

All radar systems can operate regardless of weather and light conditions. For SAR to be able to penetrate leaves and camouflage, it needs to operate in the low end of the VHF (Very High Frequency) band. Operating in this band makes it suitable to find meter-sized objects, such as vehicles, that might be concealed. When monitoring a wide area, it can be hard for a person to visually detect changes in images taken at different occasions. Therefore, an important part of wide area ground surveillance is change detection. What the current system lacks is the ability to run change detection in real-time.

The problem of running change detection in real-time has been addressed in several papers such as [1], [17], [18], [23] and [29]. In these papers the authors study real-time change detection and processing of SAR images and investigate which requirement there are on the hardware in order to run change detection in real-time.

1.1 Problem description

The algorithms for change detection that are used in CARABAS today are written in MATLAB and execute sequentially. It would be advantageous being able to detect changes while the image is being created. The existing implementation is too slow to run in such a real-time application. The aim of this project is to rewrite the algorithms in C++ and parallelize critical parts in order to see if it then would be possible to run change detection in real-time. To our knowledge there is no previous research which has tried the same approach to obtain a real-time performance.

1.2 Scope

The project will focus on the parallelization and change of language as a measure of speedup. Thus, no major changes will be made to the existing algorithms. No native implementation of fast Fourier transform is written, but the existing library FFTW version 3.3 is used.

1.3 Thesis outline

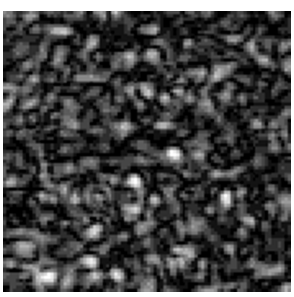
The thesis is written with a specific system in mind, CARABAS, and a specific set of algorithms. As a result of this, the thesis will start with some theory about CARABAS and the used change detection algorithms. The thesis also contains a chapter about parallelization which has the goal of introducing important terms for readers who have little knowledge of parallelization.

- Chapter 2 provides an overview of the CARABAS system and change detection.
- Chapter 3 describes important concepts and terms regarding parallelization and how performance of a parallelization can be measured.
- Chapter 4 presents what similar work has been done previously and which methods they have used to solve the problem.
- Chapter 5 introduces OpenMP. The chapter describes it properties, how to use it and what risks there are.
- Chapter 6 describes how the Fourier transform is used, gives its definition and explains how the library FFTW is used in order to compute the Fourier transform.
- Chapter 7 gives a description of how that work was conducted.

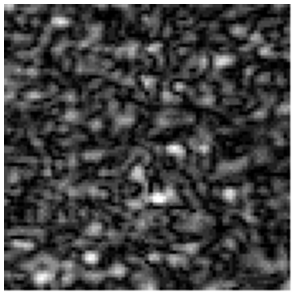
- Chapter 8 describes in detail the change detection algorithms which are analyzed in this report.
- Chapter 9 presents results from the theoretical analysis of the code, i.e. time complexity. It also presents execution times achieved when testing and analysing the performance.
- Chapter 10 contains an analysis of the results that have been presented as well as a discussion about the ethical aspects of this work.
- Chapter 11 summarizes the results and the discussion. In the chapter future work is described.

2 Background

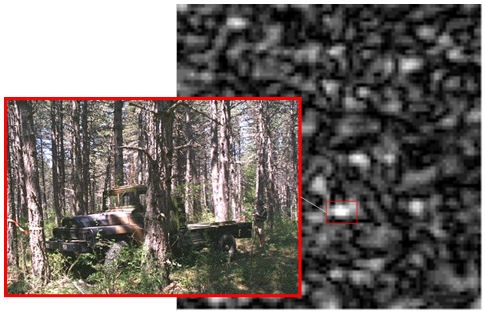
Surveillance of the ground could be used to estimate the amount of biomass and in a military context to find potentially hostile objects. An example is that it was used to estimate how many trees were still fallen after the storm Gudrun had hit Sweden. In Chapter 1 it was mentioned that it might be hard to visually detect changes in a scene. An example of this can be seen in Figure 2.1, which shows a reference image and a new image where a truck is placed under the foliage. Thus it is practical to automate the change detection process so that it does not depend on a person's ability to visually see changes.



(a) Reference Image



(b) New Image



(c) Truck in new image

Figure 2.1: SAR image of forest. Image courtesy of Saab.

In this report the focus lies on the military context, which is finding meter-sized and larger objects. In Chapter 1 it was said that the SAR used in CARABAS operates in the VHF band, to be more precise it operates in the lower part of the VHF band in the span 20-90 MHz. A false alarm in this context means that the algorithm claims it has found a change, but in reality there is none. Using SAR in this bandwidth have shown to give a low number of false alarms per unit area and high probability of detecting objects with the size of a vehicle [27].

The strength of the VHF is its ability to penetrate camouflage and foliage, i.e. that it can see through leaves of trees. On the other hand, if the goal is to find smaller objects it is necessary to use a higher frequency on the radar which means it cannot see through foliage. The system is also scalable. It can be mounted on a helicopter as shown in Figure 2.2, but can also be put on a large airplane. A helicopter flying at six kilometers height can cover five square kilometers per minute whereas a business jet can cover five square kilometers per second [2].

Figure 2.2: Helicopter with SAR antennas. Image courtesy of Saab.

2.1 Overview CARABAS processing architecture

Before describing how change detection works, a short description of the CARABAS system will be given. CARABAS3 is the third generation of CARABAS and the most recent one. The processing of the SAR images in CARABAS is shown in Figure 2.3. It describes the process of getting data from the antennas until there is an image and change detection can be performed.

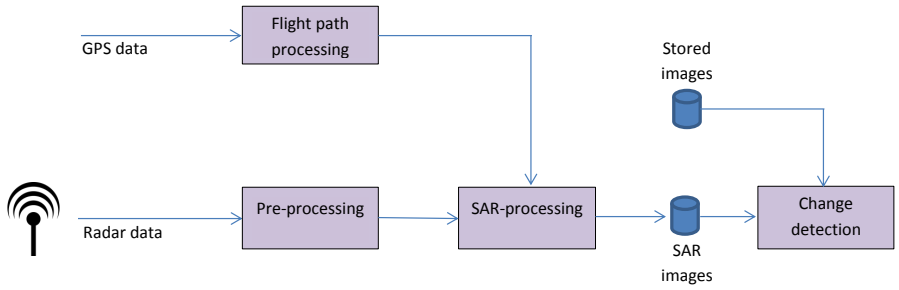


Figure 2.3: Processing blocks CARABAS3.

The antenna picks up radar signals which are initially processed. The system also collects GPS (Global Positioning System) data, which consists of data from airborne and ground GPS. The data then enters the *flight path processing* block, which in short means the block where the flight path is determined based on the GPS coordinates.

Ideally, when sending out signals from an antenna the pulse should be a narrow peak with a high amplitude in order to get a high resolution in range (distance). Generating a peak with these properties is, however, impossible to achieve in practice as it requires a lot of power. CARABAS uses pulse compression to solve this problem. The basic idea of pulse compression is when sending out the pulse it is spread out over time using a certain pattern. When receiving the signal, the peak can be determined using the pattern that was used to spread it out [28]. The pulse compression among other things is taking place in the *pre-processing* block.

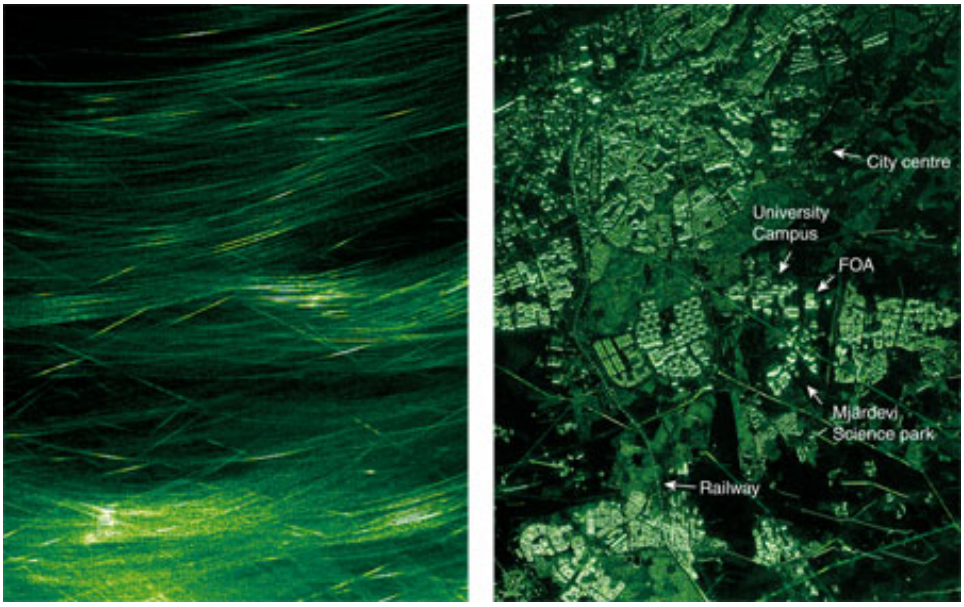


Figure 2.4: Image after pulse compression (left) and after SAR-processing (right). Image courtesy of FOI.

In the *SAR-processing* block, the SAR images are created. The input to the SAR processing can be seen in the left picture in Figure 2.4. The result after the SAR processing can be seen to the right. SAR is a type of side-looking airborne radar. When flying, each point on the ground is illuminated a big number of times from different directions as illustrated in Figure 2.5. This would explain the arches to the left in Figure 2.4. So each point in reality gives a series of virtual points, which means that the next step is to combine the information from each point in the arch to get a specific point in the SAR image [20]. Each pixel in the image is represented by a complex number, i.e., an angle (phase relative emitted signal) and an amplitude (strength pixel). The last operation done in the *SAR-processing* block is called *geocoding*. Geocoding consists of resampling the image to geographical coordinates, i.e., each pixel is mapped to a GPS-coordinate [12].

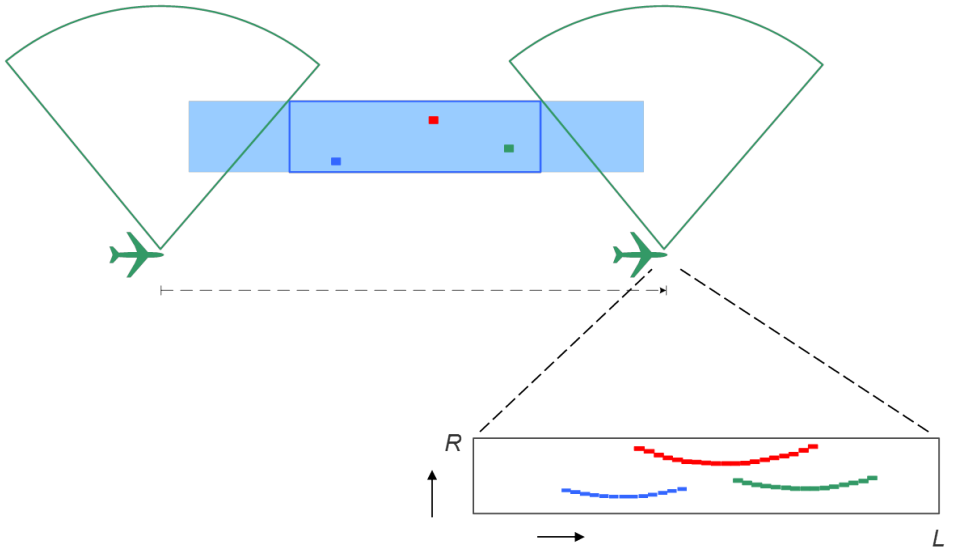


Figure 2.5: Illustration of how each point at the ground looks like in the SAR data. Image courtesy of Saab.

2.2 Change detection

After the data has been processed through this system, there is a SAR image and at this point change detection can be performed. Change detection basically consists of taking a newly processed image and compare it to an older image of the same area stored in a database. The change detection then identifies coordinates where some change has taken place. The change detection that is currently used is sequential and written in MATLAB. For the future it is beneficial to be able to run change detection in a real-time environment. This means that the change detection is running while flying and continuously outputting potential targets.

The considered change detection is developed by FOI (Swedish Defence Research Agency) and consists of a number of processing parts. These steps are illustrated in Figure 2.6. An alternative change detection is under development. The new and existing change detection have the first two steps in common.

A question one might ask is why such a complicated process is needed to find changes. *Why is not possible to just subtract the images from each other?* This is due to the fact that it is impossible to fly exactly the same path twice and noise from the background in the signals might differ from time to time.

Figure 2.6: Steps in algorithm

In Section 2.1 the concept of geocoding was introduced. Change detection takes two geocoded images as input. The first part of the process is the *pie-filtering* which takes a geocoded SAR image and a spectral domain as input and outputs an image with the wanted spectral domain. A spectral domain is a part of the frequency plane. A more detailed explanation of how and why it is sometimes useful to operate in the frequency domain instead of the spatial domain (the normal image space) will be presented in Section 6 and the pie-filter is described in more detail in Section 8.1.

Next step in the process is *matching*, which consists of determining how much the pixels need to be translated in the new image in order to make the images match as good as possible. This will be further elaborated in Section 8.2.

The process then continues with *change analysis*. This is where the actual comparison of the images takes place. The fundamental idea is to suppress the stationary background and enhance the changes. The change analysis is performed using a number of statistical methods, including Bayes linear classification [24].

The output from the change analysis is changes between the images. In order to reduce false alarms the algorithm uses a threshold. However, to be able to use the threshold the algorithm first calculates the local background statistics in the change image. The normalized image is produced using a *Constant False Alarm Rate (CFAR)* filter [24][27].

Last in the change detection process is *detection analysis*. As indicated before, this consists of using a threshold to reduce false alarms. The detection analysis also consists of rejecting detections indicating objects that are too small to be detected. It might also be the case that an object is big enough to cover several pixels which gives several detections, thus a part of this step is also to cluster detections that are coming from the same object. After these steps are done the remaining detections are shrunk to one pixel and the algorithm outputs a list of coordinates where changes have been detected [12][24][27].

3 Parallelization

Parallelization is a large area which concerns many applications. In this chapter, terms and concepts relevant for understanding the project will be introduced. A wide range of notions will be explained, from hardware properties to different methods for parallelization. The chapter also presents methods for measuring the performance of a parallel algorithm compared to a sequential algorithm.

3.1 Parallel hardware

In order to know how to approach a parallelization problem it is useful to be able to categorize the properties of the system. A categorization of a system can be made with Flynn's taxonomy. The categorization is done based on how data flows in and through the system. Here follows a description of the four categories in Flynn's taxonomy [19, p. 28-33] [22, p. 10].

Single-Instruction, Single-Data (SISD) There is one input flow of data and the instructions are executed one instruction at time.

Multiple-Instruction, Single-Data (MISD) Several processing units obtain the same data and executes their instructions using this data. This means that each core runs different instructions on the same input data. This kind of system is rarely used in practice.

Single-Instruction, Multiple-Data (SIMD) The instructions are run on several processing units, but the input to the units differs. This is useful in applications which have a high degree of parallelization, e.g. computer graphics algorithms. Another way of expressing a system with multiple data is data-parallelism.

Multiple-Instruction, Multiple-Data (MIMD) This is the most common concept for parallel computers. In MIMD, different instructions are run on each processing unit and the input also differs between the processing units. A typical example of this type of system is a multi-core processor. The processing of the different processors/cores is asynchronous.

Another way to distinguish between systems with parallel code is to look at the memory. Either the processors/cores can share the memory or the memory is distributed. In this project a multicore processor is used, which means that the memory will be shared [22]. Thus, the focus from here on will be on shared memory systems with several cores.

3.1.1 Cache memory

When working with a shared memory system, the behaviour of the cache memory is very important to understand. The purpose of the cache memory is to reduce the gap in time between executing instructions and getting data from the main memory. The cache is a small and fast memory that stores a subset of the main memory, this subset contains the most frequently used data. In modern computers there are several layers of cache for each processor [22, p. 64-65]. There is a principle called *locality*, which means that an access of a memory location is often followed by an access to a memory location nearby. Due to this fact the cache operates on memory blocks instead of individual instructions or pieces of data [19, p.19].

An important term in the context of cache memories is *cache coherence*. Cache coherence is relevant when working with a multicore processor, where each core has its own caches. Suppose there are two cores, core `A` and core `B`, which have their own caches. Next assume there is some piece of data `x`, which is contained in both caches. If the value of `x` would be changed in cache `A` then there needs to be a way of informing the cache of `B` that its copy of `x` is invalid. The problem of making sure that all caches are using the same value of `x` is called *cache coherence*. There are several protocols to solve the problem of cache coherence [22, p. 75-82].

3.2 Parallel software

When having a hardware which enables parallelization, the next step is to make sure that the software can use it in an efficient and secure way. This section presents important aspects of parallelizing software, both properties on a system level as well as methods for parallelizing an algorithm. Finally, some risks regarding parallelizing will be presented.

3.2.1 Processes and threads

To begin with there are two terms related to the operating system that are important to know: *process* and *thread*. A process is an instance of an executing computer program. An important property of a process is that it has its own stack and its own heap. A thread on the other hand is "lighter" than a process. A process can consist of several threads. The threads share address space which means that threads belonging to the same process will use the same heap but not the same call stack. The fact that threads use the same address space makes them vulnerable to data races (data races are described in more details in Section 3.2.4) [19, p. 17-18]. The term *thread* is often used when referring to a shared memory system and *process* often is used in connection with distributed memory system [22]. As mentioned in Section 3.1, a shared memory is used and therefore the focus will from here on lie on threads.

3.2.2 Parallelizing software

The procedure of parallelizing a program can be described by the following three steps. In this project the API OpenMP will handle step two and three, thus no detailed description of these steps will be given.

1. The algorithm is decomposed into tasks. A task is one instruction or several consecutive instructions. The computation time of the tasks is called *granularity*. A more detailed description of this step will follow in Section 3.2.3.
2. Assign each task to a thread. At this point it is necessary to solve a load-balancing problem, because in order to get as good total execution time as possible, the work needs to be as evenly distributed as possible over the threads. However, there are also other things to take into consideration such as good cache usage. Assigning tasks to threads is called *scheduling* [22, p. 96-97].
3. Each thread is mapped to a core in the hardware where it will run. Just like in the second step, the goal is to spread out the work as evenly as possible over the cores [22, p. 96-97].

3.2.3 Decomposition

This section presents methods for decomposing algorithms. First, different ways of decomposing a problem will be presented, then, the pros and cons of different granularities are laid out.

The decomposition can be divided into two categories: *data decomposition* and *control decomposition*. In the first case, the partitioning is done on the data and in the second case on the computations [5, p. 47-48][10]. Parallelization can be done at different levels, either at a higher level called a functional parallelism or at a lower level called instruction level parallelism [19]. In this section only the types of parallelism relevant for this project will be presented.

Instruction level parallelism (ILP) is a fine-grained level of parallelization. The goal of ILP is to make it possible for several instructions to be executed simultaneously. There are here two different approaches: *pipelining* and *multi issue*. Pipelining is similar to the concept of an assembly line. The execution of several instructions are overlapping. This requires that there are no data dependencies between the overlapping instructions. If there are overlapping data dependencies, pipelining can still be used, but there will be waiting times. The stages of the pipeline should take roughly the same time in order to minimize waiting times [22, p. 8-9]. Multi issue, on the other hand, means that several instructions are simultaneously instantiated. For example, there is a loop without data dependencies between the iteration, then the iterations can be divided between the cores so that several iterations can be executed at the same time which will reduce the running time of the loop [19, p. 25-26].

Data parallelism was mentioned briefly already in Section 3.1. Data parallelism means that it is the data, and not the code itself, that is divided between the cores [5, p. 47]. This is often used by MIMD models, particularly in a special case of MIMD called SPMD (Single Program, Multiple Data) where each core processes different data, but the instructions are the same at all cores. In addition, the execution of the instructions are asynchronous. An example of data parallelism is a big set of data that will be processed by the same program. The data can be divided into subsets and assigned to different cores. The same program will then run on all units, but the data will differ [22, 100-101].

A question one might ask is whether fine-grained or coarse-grained is best to use in a multithreading environment. There is no clear answer to this question, as it depends on a number of factors. In a fine-grained parallelization, the number of times the processor is idling can be minimized since if a thread is not ready to execute, another can be picked instead. However, if the system is old, threads are often implemented as processes which means that the time for switching between the threads might be significant. Fine-grained parallelization also have the drawback that a thread ready to execute a long sequence of code may have to wait. A fine-grained decomposition also entails more communication which can slow down the execution. Having a coarse-grained parallelization on the other hand means that the processor will switch when a thread is waiting for some time-consuming operation. The drawback here is that the processor might idle when a thread is blocking for only a short time [19, p. 29].

3.2.4 Risks with threads

At times a parallel algorithm can get a worse execution time than its sequential equivalent or even output the wrong result. There can be a number of reasons why this happens and some of the reasons will be presented in this section. It is important to be aware of these risks in order to take necessary precautions.

Data race A data race, also called race condition, might appear when there are two or more threads which have access to the same variable and at least one thread is writing to the variable. For example, thread `A` computes $a+=b$ and thread `B` performs $b=7$. This will yield different results depending on if `A` completes its operation before `B` performs $b=7$ or if B changes the value of b before A reads it. Essentially, the goal is to synchronize the data access so that a race cannot occur. This is done by only allowing mutual exclusive access to critical sections of the code [19, p. 51]. A part of the code which requires mutual exclusion will enforce serialization on the critical section [19, p. 51]. This fact can have a big impact on the execution time.

Deadlock A deadlock can occur when having sections with mutual exclusion. Deadlock basically means that two or more threads enter a state where they are waiting for each other to finish executing. For example, thread `A` enters critical section `1` and then wants to enter critical section `2`. At the same time thread `B` enters critical section `2` and then wants to enter section `1`. This then leads to a deadlock state [19, p. 250-251].

False sharing When describing cache memories in Section 3.1.1 it was mentioned that the caches operate on blocks of data. These blocks can also be called *cache lines*. The reason for using blocks was to improve performance, but the blocks themselves can also be a source of bad performance in a multi-core system. In case any value of a cache line is changed and that cache line is also stored in some other cache, the entire cache line will be invalid. For instance, there is a core `A` and a core `B` which have their own caches. Suppose there is an array `y` and the entire `y` is stored in one cache line. `A` operates on the first half of `y` and `B` on the second half, hence, there are no shared variables. However, since `y` is stored in one cache line, `A` needs to get a new copy of `y` each time `B` has made a change in its half of the array and vice versa. The cores will probably need to access the main memory many times, which will result in bad performance, but still a correct result [19, p. 46, 254-255].

Data dependency There are three categories of data dependency: *flow dependency*, *anti dependency* and *output dependency*. Data dependencies occur between instructions I_1 and I_2. In the first category, flow dependency, I_1 computes a result value that is then an operand for I_2. Thus there is a flow from I_1 to I_2. The second, anti-dependency, means that I_1 uses a value in a register or variable which I_2 later on uses to store a result of a computation. The third, output dependency, is when I_1 and I_2 use the same register or variable to store the result of a computation [22, p. 98-99].

3.3 Performance

When a sequential program is parallelized, it is of interest to know how successful the parallelization was. There are several different tools for measuring the performance of a parallel program and these will be presented in this section. If a program is not parallelizing well, this might be an indication that the sequential program is not suitable for parallelization. The sequential execution time is denoted T_s and the parallel execution time is denoted T_p. The processor has p cores, one thread is running on each core.

3.3.1 Speedup

Speedup is a factor of how much faster the parallel program is than the sequential program. A linear speedup means that $T_p = T_s/p$ and T_p is the optimal parallel running time. Achieving a linear speedup is in practise unlikely, since having multiple threads introduces overhead and it is hard to

make all threads execute exactly the same amount of work. The speedup, denoted S, is defined in Equation (3.1). If the speedup is linear then $S = p$. In a formal definition, T_s is defined as the execution time of the fastest sequential algorithm. It might, however, be hard to know the running time of the fastest algorithm and therefore the speedup is often computed for the sequential version of the parallel algorithm [19, p. 58] [22, p. 162-163].

$$S = \frac{T_s}{T_p} \tag{3.1}$$

In theory $S \leq p$, but in practise it can sometimes be observed that $S > p$ and this phenomenon is called *superlinear speedup*. When a superlinear speedup occurs, it is most likely due to cache effects. Cache effects means that when running a program on p cores instead of one core, there are also p times more cache memories to use. If the data set is distributed over all these caches, it might result in all data fitting into the cache memories, hence, there will be no cache misses. Getting a superlinear speedup is unusual [22][p. 163].

3.3.2 Efficiency

Another way of measuring the performance is to look at the efficiency. The definition of efficiency E is given by Equation (3.2). The efficiency expresses the fraction of time for which the processor is performing useful work that also has to be performed in the sequential program. The speedup $S = p$ corresponds to the efficiency $E = 1$ [19, p. 58] [22, p. 164].

$$E = \frac{S}{p} = \frac{T_s}{p \cdot T_p} \tag{3.2}$$

3.3.3 Amdahl's law

Even if there are parallel resources, the speedup of the execution time cannot be infinite. The number of cores is one restriction which gives an upper bound on the speedup, but it is not the only restriction. Another restriction can be data dependencies or synchronization which limits the speedup. Amdahl's law expresses the attainable speedup and is given in Equation (3.3). A fraction f of a program can only be executed sequentially, $0 \leq f \leq 1$. This equation says that even though the parallel part can have an arbitrary speedup, the speedup of the entire program can never be greater than the inverse of the fraction f. For instance, if 20% of the code needs to be sequential then the speedup cannot be higher than $1/f$=5 [19, p. 61] [22, p. 164].

$$S = \frac{T_s}{f \cdot T_s + \frac{1-f}{p} T_s} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f} \tag{3.3}$$

3.3.4 Scalability

The expression that a program is scalable can often be used in a rather informal way, but when considering parallel programs there is in fact a formal definition. Scalability seeks to answer the question *What happens with the execution time if the number of cores increases?*. A program is said to be scalable if for an increase in number of threads there exists a corresponding rate in the increase of the input size such that the efficiency E is unaltered. In short, if the input size increases it is possible to, by using a larger number of cores, preserve the same efficiency. In order to express this more formally Gustafson's law can be applied, which is a variant of Amdahl's law [19, p. 62][22, p. 165-166].

Gustafson's law

In this report a special case of Gustafson's law is considered: the execution time of the sequential part is constant, hence, it does not depend on the input size. This means that the part of the code which needs to be executed sequentially is not constant, it decreases with the input size. Gustafson's law is given in Equation (3.4) where n is the input size [22, p. 165-166].

$$\lim_{n \to \infty} S(n) = p \tag{3.4}$$

4 Parallelization of change detection in SAR

Parallelization of change detection in SAR images is a rather unexplored research area. Some similar work has been done for example in [17], where they derive a parallel algorithm for change detection in videos. The algorithm is run on a cluster of computers in a master-slave manner. The video frames are divided into subframes, distributed to the nodes in the cluster (slaves) where change detection is run on each subframe. A similar approach is presented in [1]. In this paper, they are working with MODIS images, which is a type of satellite image and a multithreaded platform. The similarity with the previous approach is that it also uses a master-slave fashion. Here, the images are split into their spectral bands and each slave processes one band.

There are also some papers which focus on possibilities closer to the hardware. In [18] the authors are studying which requirements there are on the hardware, such as size of memory, communication bandwidth and processor speed in order to be able to run the change detection algorithms developed by RSRE (Royal Signals and Radar Establishment) in real-time. In [23], the author describes approaches to improve the running time when processing SAR images. He discusses also different ways of decomposing the problem and the pros and cons with different platforms for SAR processing. Another approach was taken in [29]. The method used in this paper uses a CUDA (Computed Unified Device Architecture) on a GPU (Graphics Processing Units) and compares its performance with a C/C++ implementation running on a CPU (Central Processing Unit). The image is split into square subimage and change detection is run on each subimage separately, thus data decomposition. Each thread processes one subimage.

5 Parallelization with OpenMP

In Chapter 3, important concepts of parallelization were introduced. A limitation set from the start of the project is to use OpenMP to realize the implementation. In this chapter important properties of OpenMP will be presented. These properties will set the possibilities and limitations of the parallelization. OpenMP is a widely used API (Application Program Interface) for parallel applications with shared memory. OpenMP can be used to extend Fortran, C and C++ [22, p. 339] In this project OpenMP will be used with C++.

5.1 Properties of OpenMP

OpenMP uses a programming pattern called *Fork-Join*, which is a quite common to use when having a shared memory system [22, p. 109]. Fork-Join means that initially there is a single thread called *master thread*. When the master thread reaches a parallel region, it forks into child threads, which do the execution of the parallel section. When the threads are done with their work they synchronize their work and terminate. The master thread is never destroyed, it executes alone the sequential parts. The execution mode of the parallel regions are SPMD or MIMD [5, p. 171] [22, p. 109,339] which means that the threads are running asynchronously. The child threads are called *slaves* or *workers* [19, p. 214]. The terms *master* and *slave* comes from the *master-slave* programming paradigm. In this paradigm, there is a master node which controls the worker nodes. Communication only occurs between the master and each worker and never among the workers [5, p. 46-47].

5.1.1 Memory

OpenMP has shared memory model. All threads have access to the same place to store and retrieve variables. Each thread also have some memory space of its own. This memory can be used to cache variables, but threads also have a memory called *threadprivate*, where it can store variables that it does not want to be accessible by other threads [5, p. 172][22, p. 339-340].

5.1.2 Loops

One of the features that makes OpenMP powerful is its parallelization of serial for-loops. After a parallel section, the workers are waiting for all other workers to finish in order to synchronize. The workers behave in the this way after a for-loop, but there are ways of avoiding this behaviour. If using the special `parallel for` directive, there are some things that are different, e.g. the variable that you are iterating over is by default private, which means that there is no risk that the iteration variable is a source of race condition [19, p. 214,224].

5.1.3 Scope of variables

The definition of the scope of a variable in sequential programming is the parts of the code which can access the variable. In order to avoid bugs, it is important to reflect upon which is the scope of a variable in OpenMP. Since there already is a definition for the scope of the sequential parts of the code this section will be about the parallel parts of the code. In OpenMP, a part of the code that runs in parallel has the directive `parallel` before. Variables can be either private or shared [19, p. 220-221].

A private variable can only be accessed by a single thread whereas a shared variable can be accessed by all threads. A variable being private means that each thread needs to create its own instance of the variable. Upon entry and exit of a `parallel` block, the private variables are undefined [3, chap. 3.4][19, p. 221].

A shared variable can be accessed by all threads in the team while executing the `parallel` block. There is only a single instance of the variable and it is placed in the shared memory. It is necessary to be attentive when having a pointer or reference that is shared [3, chap. 3.4.2]. Variables declared before the `parallel` block are by default shared. This can be of importance, e.g., if there are references from inside the parallel block to a variable declared before it [19, p. 221]. Variables declared inside the `parallel` block without an implicit scope is by default shared [3, chap. 3.4.4].

6 Fourier Transform and FFTW

When looking for the result of two signals in the time domain, e.g, when filtering a signal, these signals needs to be convolved. However, following from the multiplication theorem, if the signals are transformed to the frequency domain they can be multiplied instead, which in general is much easier to do [21, p. 24].

This section will explain how one transforms a matrix from the spatial or time domain to the frequency domain both in theory and practise. The section starts by giving the theory of transformation to frequency domain, discrete Fourier transform. Next, the implementation of transformation is given, fast Fourier transform. Finally, we will introduce the library FFTW, its properties in both sequential and parallel environments.

6.1 Discrete Fourier transform

Transforming discrete data from the spatial (position) domain or time domain into the frequency domain is done by computing the discrete Fourier transform (DFT). Suppose there is an array containing N pieces of data uniformly sampled, i.e., it is sampled with equally spaced intervals. The DFT of this array is computed by Equation (6.1), where $X^F(k)$ is the k'th DFT coefficient, $x(n)$ is the content of the array where $n = 0, 1, \ldots, N-1$ and $j = \sqrt{-1}$ [21, p. 5].

$$X^F(k) = \sum_{n=0}^{N-1} x(n) e^{\frac{-j2\pi}{N} kn}, k = 0, 1, \ldots, N-1 \tag{6.1}$$

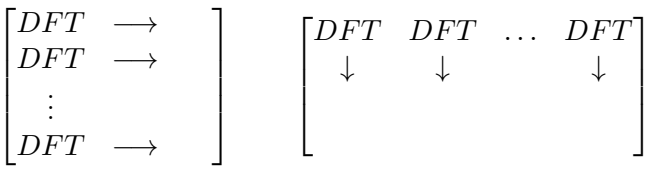


Figure 6.1: Processing of DFT on 2D matrix

Equation (6.1) describes how to compute the frequency content of a one dimensional array, but in this project transforms in two dimensions (2D) needs to be computed. For a 2D $N \times N$ matrix computing the DFT corresponds to computing the Equation (6.1) for each row giving us a new matrix. However, the computation of the 2D DFT is not finished here, the equation also needs to be calculated for each column of the new matrix in order to get 2D DFT. The procedure of 2D FFT is illustrated in Figure 6.1. It is of no importance if the DFT is applied to rows or column first, important is that it is done in both directions [21, p. 128-130]. From the description of 2D DFT it can be shown that the time complexity is $O(N^4)$ ($O(N^2)$ for N points).

6.2 Fast Fourier transform

In Section 6.1 it was shown that the time complexity of DFT is O(N^2) for a N samples. This means that calculating the DFT quickly becomes very slow for large matrices. In order to compute DFT without using a naive implementation one can implement FFT (Fast Fourier Transform) instead. To transform from the frequency domain to the spatial or time domain, the inverse Fourier transform (IFFT) is used. The definition of IFFT is very similar to the definition of FFT, the only difference is a change of sign.

There are several versions of FFT, in this report the one named Cooley-Tukey will be studied. The reason is that Cooley-Tukey is the most commonly used algorithm and also the one that is used in the library FFTW, which will be described in the next section [9]. A number of variants of Cooley-Tukey exists, where small changes has been made to the algorithm. For example, assume the image is padded in order for its dimensions to be powers of two. This makes it possible to use the Cooley-Tukey algorithm with the case radix-2. The Cooley-Tukey algorithm is a divide-and-conquer algorithm which recursively divides the set into even and odd indices. The $O(N \log_2(N))$ running time can be proved using the Danielson-Lancoz lemma and described by a Butterfly diagram [21, p. 41-44].

6.3 FFTW

FFTW is a library containing subroutines for C programs (it can be used with some other languages too using wrappers). Using FFT, it computes the DFT of matrices with arbitrary sized input and arbitrary numbers of dimensions. FFTW is developed to operate on matrices stored in row-major order. FFTW uses the Cooley-Tukey algorithm which was described in Section 6.2. Depending on the properties of the matrix, it uses different variants of Cooley-Tukey. Multi-dimensional transforms are in general realized by applying the 1D-transform on each dimension of the matrix [11].

FFTW is generic in the sense that it is not developed for a specific hardware. This means that before computing the FFT, FFTW needs to do some planning which is hardware specific. The settings, which are done with respect to the hardware, forms a *plan*. The plan can be quite time-consuming to determine and due to this the programmer is given the opportunity to choose between waiting for the optimal plan to be found or to settle for a solution that is suboptimal, but rather quickly found [11].

FFTW can either take a pointer to an array as input and output the result to the same location, which is called *in-place* transform. Another way is that the pointer to the input array and output array are at different places in memory, which is called an *out-of-place* transform. Which kind of transform is used might in some cases have an impact on the execution time.

6.3.1 Parallel execution of FFTW

FFTW uses different ways of parallelization depending on if the transformation has one or several dimensions. For a transform of one dimension it uses the fact the Cooley-Tukey divides a problem of size N into subproblems such that $N = N_1 N_2$. The N_2 subproblems are of size N_1 and the N_1 subproblems is of size N_2. Then the subproblems are divided among the threads. If the transform to be calculated is of two dimensions or more, then FFTW runs 1D transforms along the rows, columns, etcetera of the matrix. These transforms can then be divided among the threads [9].

FFTW supports parallelization with OpenMP. This means there are built-in functions one can use to instruct FFTW to do the parallelization using OpenMP. All functions in FFTW are not thread safe which means that there might be race conditions that result in an incorrect result if not being attentive [11].

Cooley-Tukey performs at its best when operating on matrices whose dimensions are small prime factors, where factors of two are particularly fast [11]. Having a matrix which does not have dimensions of small prime factors, one can zero pad the matrix in order to achieve this.

6.4 FFT in MATLAB

MATLAB uses FFTW to do FFT computations and also adds its own layer of optimizations. If computing the FFT followed by the IFFT in MATLAB one gets back the original matrix. This is due to the fact the IFFT in MATLAB computes a normalized transform whereas calling FFTW directly will give an unnormalized transform. To normalize a transform you divide each element by the number of elements in the matrix. Another important property of MATLAB is that is stores its matrices in column-major order.

7 Method

The work proceeded in an iterative manner. As described in Section 2.2, the considered change detection is in fact a processing chain consisting of five parts. All the steps described in Section 7.1 were done for each part in the processing chain before moving on to the next part. The motivation behind this setup was in case of running into unforeseen obstacles making it impossible to finish the work within the time frame for the project, there would be a good ground for someone else to continue the work. This chapter also describes how the verification of the program was performed.

7.1 Working procedure

First, a study of the algorithms was conducted in order to understand how they work. This consisted of studying existing code and reports regarding the algorithms. A good knowledge of the algorithms was critical since the next step was to identify time and space consuming parts of the algorithms, i.e. parts that are potential bottlenecks. Bottlenecks in the processing chain might also be data reorganization that is done between the steps. Data reorganization could be that a matrix needs corner turning since it in one step is processed row-by-row and in the next step in column-by-column.

The second step was rewriting the MATLAB code in C++. Even though the algorithms already exist implemented in MATLAB, there are a number of things to take into consideration when implementing this. Since MATLAB is a very different language from C++, it is necessary to carefully choose which data structures to use. The reason behind rewriting the code in C++ before parallelizing was to see whether that particular part of change detection needed parallelization at all or get a sense of about how much of a speedup was needed.

The third step was to study different approaches for parallelization. Change detection consists of several algorithms for which most likely different methods of parallelization are optimal. The problem was to find one method which would give the overall best performance. As mentioned earlier, determining which method to choose was based on the operations identified as critical in the first step and see the effect the different approach have on these operations.

Having decided on a method for parallelization, the next step was to implement it. Since the code was running in a parallel environment, this meant that it was also important to ensure that no data races occurred. The plan was to use OpenMP, but if it turned out there was another API, which was better suited then it might be used instead.

After the implementation was completed, the next step was to evaluate the result. The evaluation consisted of studying the attained execution times and evaluate whether these were enough for a real-time performance. If the algorithms were not fast enough, then it was analyzed how much faster the algorithms would need to be, both in time and speedup. The algorithms were also analyzed in order to determine whether there were any specific bottlenecks preventing the real-time performance.

7.2 Verification

The verification of the results were challenging since there is a large amount of data. The goal was, therefore, to minimize the manual verification of the data. A script was written which compared outputs from the C++ and MATLAB program. The result of the FFT differed slightly between MATLAB and C++ and therefore required manual verification. The output from the sequential and parallel C++ program needed to be identical and could therefore be verified using a script.

8 Change detection algorithms in detail

Change detection consists of several steps. The steps of change detection are studied all through one at a time. Even though the different steps have different tasks to perform some operations are done at several places. One such operation is FFT, which was described in Section 6.2. The input to change detection is two geocoded SAR-images. The term geocoding was introduced in Section 2.1 and means mapping each pixels to GPS coordinates. The first step of change detection, *pie-filtering*, will in this chapter be thoroughly described. The description of the second step, *matching*, is quite brief since there was not enough time within the time frame of the project to study the matching deeper.

8.1 Pie-filtering

Another name for pie-filtering is *spatial band-pass filtering*. In short, pie-filtering means that unwanted frequencies in the input are eliminated. Since the goal is to compare two images, it is necessary that these images overlap. As a consequence of this, the parts of the images that do not coincide need to be cut away. Even though identical flight paths are desired, there will always be some differences as illustrated by Figure 8.1.

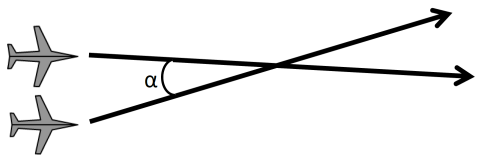


Figure 8.1: Flight paths at different runs

When CARABAS is operating, many different frequencies are sent out. These frequencies bounce differently on objects depending on the angle with which the signal hits them. The objects can be the ground, trees, houses, etcetera. The angle is depending on the properties of the flight path, such as flight height. In order for the images to be comparable, only the frequency components that exist in both images are to be kept. In other words, the frequencies that bounce in the same way remain and the rest is filtered.

Filtering unwanted frequencies is done by finding what the common angle is and creating a filter which filters away all components which are not within this angle. In Figure 8.2 are the frequency spectra from two images. The difference in distance from the origin is caused by differences in flight altitude and the difference between the centre of the two spectra is due to different bearing direction (the direction in which the plane is heading). The difference in bearing is denoted α and can be seen in Figure 8.1 and Figure 8.2. The part that is dark grey is the common spectral components which will remain after the pie-filtering [13].

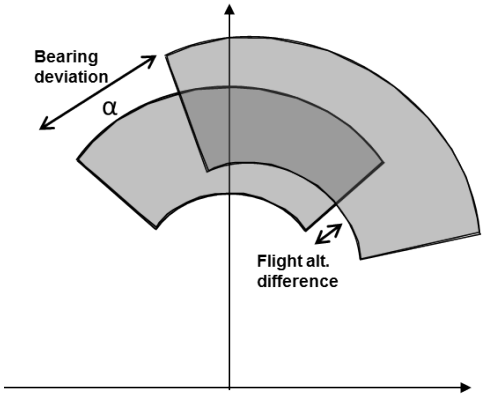


Figure 8.2: Spectral components from two images

8.1.1 Description of pie-filter implementation

The pie-filter takes an angle which defines the spectra that we want to remain after filtering. As indicated before the pie-filter is in fact a band-pass filter. The filter is defined as the intersection between a circle sector and annulus in the frequency domain [24]. The implementation of the pie-filter can be divided into four parts which are described schematically in Figure 8.3. The figure also shows that each step can be divided into a number of subparts. The subparts consists of operations that are either important for the algorithm or time-consuming. The derivation of the time-consuming parts will be explained in Chapter 9.1.

Preprocessing prepares the matrix representing the image, i.e. ensuring it is in the right format for further processing. The purpose is also to calculate important parameters for the subsequent parts. *Find nonzero* refers to finding the first nonzero row and the first nonzero column as well as the last row and column that are nonzero. The second subpart, *Find filter properties*, defines properties of the desired filter such as which frequencies to keep and filtering. The last subpart, *Calculate flight properties*, calculates properties of the flight track.

Filter construction creates a filter based on the properties calculated in the *preprocessing* and image properties. The filter is a matrix where each cell corresponds to a frequency. Formats the pie-filter in order for it to have the same format as the image matrix.

Filter image first transforms the image matrix to the frequency domain by FFT. Secondly, the filter matrix and image matrix are multiplied element wise. Finally, the resulting matrix is brought back to the spatial domain by IFFT.

Post-processing ensures the resulting matrix has the correct dimensions. Since both the original image matrix and filtered image are padded, the size of the padding needs to remain the same.

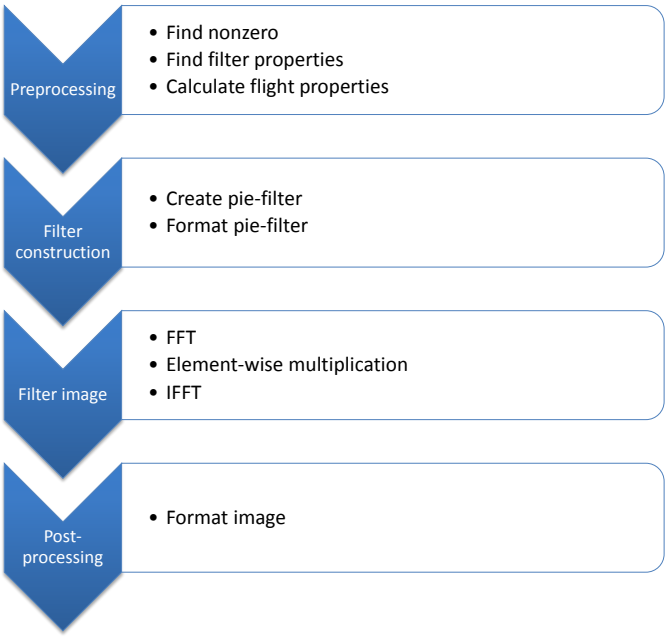


Figure 8.3: Implementation of pie-filter process

The calculations of the first step need to be completely finished before continuing with *Filter construction*. In Figure 8.3 the sequential behaviour was described whereas the flow of the data and data dependencies are described in Figure 8.4. It is possible to divide the preprocessing block into subblocks. The *FFT* and *Filter construction* will use different results from the *preprocessing*, but since the *preprocessing* already is fast, which will be shown in Section 9.1, it will be left as one block.

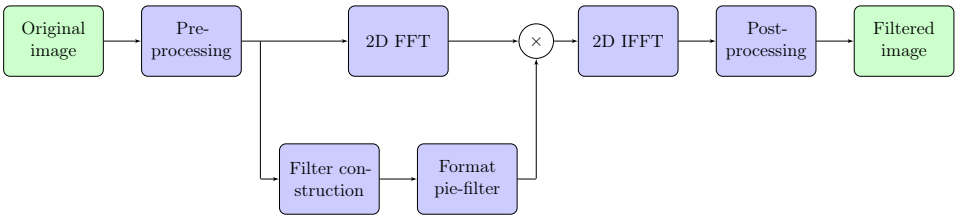


Figure 8.4: Data flow in change detection

Both FFT and IFFT are in themselves complex algorithms and therefore worth studying in more detail, which is done by focusing on the middle part of Figure 8.4. The middle part contains FFT, element-wise multiplication and IFFT. Using the information about FFT from Section 6.4, this part can be described as shown in Figure 8.5. This figure is correct for languages storing their matrices in column-major order such as MATLAB. For languages storing in row-major order, such as C++, the order is inverse: first FFT on each row and then FFT on each column as described in Section 6.3.

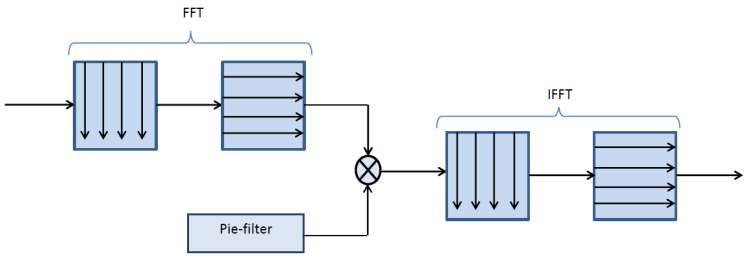


Figure 8.5: Part of pie-filtering process (matrix stored in column-major order)

8.2 Matching

Having pie-filtered the images it is now time for the second step in the change detection process. In order to run a change analysis on two pictures it is necessary that they match. The reason they do not match, even though they have been geocoded and pie-filtered, is mainly due to inaccuracies in flight track and DEM (Digital Elevation Model) data [24]. After pie-filtering, the images contain the same frequencies, matching makes sure that these also look as much alike as possible.

8.2.1 Description of matching implementation

The basic concept of matching is that you want to find the translation of the image that gives the best possible match. The matrix that describes an image consists of a number of subimages as illustrated in Figure 8.6. First, the algorithm finds the maximum correlation for each corresponding subimage pair. This is done by performing the following steps for each subimage pair.

1. Optional: Detect the edges in each image
2. Calculate the Fourier transform F_1 and F_2 of each subimage
3. Calculate the inverse Fourier transform of $F_1^* \cdot F_2$
4. Calculate the maximum correlation, which is the translation (shift) which will make the images coincidental.

After these steps have been performed on all subimage pairs, the global translation can be calculated from the median of the subimage results. Since the algorithm is looping over subimage pairs, the more subimages there are the more pairs there are. Thus, from this point of view it is beneficial to keep the number of subimages as low as possible, even only computing a global translation between the images without dividing into subimages. The problem is unfortunately not this simple because the more subimages there are the more accurate is the result because by taking the median of the translations bad values can be ignored. Thus, this is an optimization problem between speed and accuracy. The algorithm for matching is described in pseudocode in Algorithm 1.

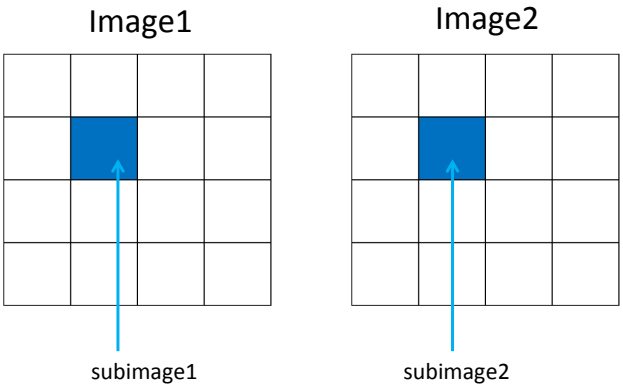


Figure 8.6: Images and their subimages which are to be matched

The computation of FFT and IFFT was described in Section 6, but something that needs a detailed description is the edge detection (step 1). The edge detection is done by convolving the subimage with Sobel convolution kernels. No detailed description of how this works will be given, the important is that a convolution is performed. Observe the star after F_1 in Algorithm 1, which means that the conjugate of F_1 is used.

Algorithm 1: Pseudocode for matching

```
input  : image1 is the reference image and image2 is the new image
output : image2 updated
bool flag;
for i = 0 to #subimages do
    subimage1 = image1.subimage(i);
    subimage2 = image2.subimage(i);
    if flag then
        conv(subimage1, Sobel kernels);
        conv(subimage2, Sobel kernels);
    end
    F1 = fft(subimage1);
    F2 = fft(subimage2);
    corrIm = ifft(F1* · F2);
    corrMax = max(corrIm);
    thisShift = shift(corrIm);
end
medShift = median(allThisShift);
meanCorr = mean(allCorrMax);
```

9 Result

In this section is a presentation of what has been achieved during the project. The results are divided according to the steps in the change detection process, shown in Figure 2.6. The results of each step will consists of the parts described in the list below. Note that the execution times are only there to give an indication, they differ slightly from time to time. One reason for these differences is that it might vary how many cores there are available for the execution.

1. MATLAB code
2. Theoretical analysis of algorithm
3. Data-structures for speedup in C++
4. Sequential C++ code
5. Identification of operations to be parallelized
6. Parallel C++ code
7. Summary

Taking a picture of the ground that is roughly 6000×6000 pixels takes about 30-40 seconds. This fact is used to get a hint of how fast the change detection needs to be. The testing is done on a server with twelve cores. The execution times presented in this section will be from running on this particular hardware.

9.1 Pie-filter

In change detection, there are several operations that are performed multiple times and it is, therefore, important to do a thorough analysis from the start since many results can be reused. This section will present the results found for the pie-filter. Figure 9.1 provides a reminder of the current location in the process. This section will provide a profound analysis of different approaches for computing FFT using FFTW.

Figure 9.1: Pie-filter, the current position in the change detection processing chain.

9.1.1 MATLAB code

A summary of the pie-filtering steps can be seen in Figure 9.2, which is a simplified version of Figure 8.3. Based on the partitioning of the code presented in Figure 9.2 the running time of the different parts are studied. The result of this is presented in Table 9.1. From the result of the table it is deduced that the *Filter construction* and *Filter image* will need special attention.

Figure 9.2: Pie-filter

Part	Time [s]
Preprocessing	6.2
Filter construction	11.2
Filter image	14.4
Post-processing	1.5
Total	33.3

Table 9.1: Running time of MATLAB code coarsely divided

The next step is to study the running time of the subparts of *Filter construction* and *Filter image*, which are described in Figure 9.3. The execution time of these parts of the code is described in Table 9.2. This table shows that the parts *Create filter*, *FFT*, and *IFFT* are the bottlenecks of the algorithm.

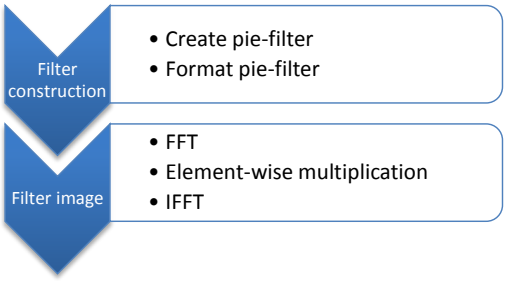


Figure 9.3: Time consuming parts of pie-filter algorithm

Part	Time [s]
Create pie-filter	10.8
Format pie-filter	0.4
FFT	5.7
Element-wise multiplication	0.6
IFFT	8.1

Table 9.2: Running time of MATLAB code of subparts in Figure 9.3

Since the contributions to the execution time from the preprocessing is not negligible, the execution times of its subparts, which are given in Figure 9.4, are briefly studied and presented in Table 9.3. The purpose of this is to see whether there is one single operation, which is considerably more time-consuming.

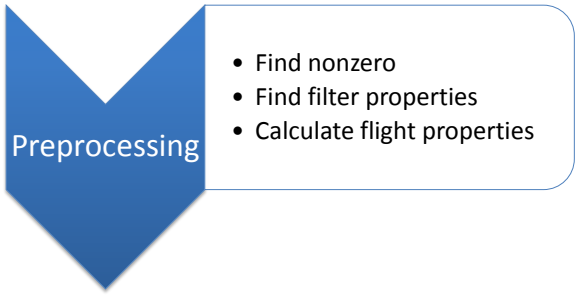


Figure 9.4: Subparts of preprocessing part in pie-filter

Part	Time [s]
Find non-zero	2.5
Calculate filter properties	1.6
Calculate flight properties	0.0

Table 9.3: Running time of MATLAB code of subparts in Figure 9.4

9.1.2 Theoretical analysis

In Section 9.1.1, it was found that the most time-consuming parts are the creation of the filter and the computation of the FFT and the IFFT. Based on this fact, the asymptotic running times will be presented in order to determine how these will react to a scaling of the input. The rest of the code has also been briefly studied in order to avoid that parts that are fast with this input size will scale up very quickly with input size, but no such problem has been found.

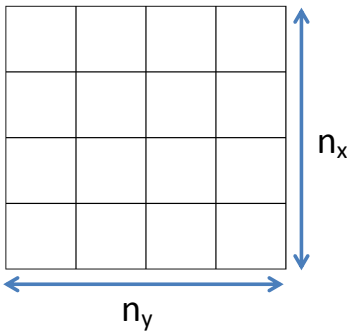


Figure 9.5: Size of matrix in each dimension

In Section 8.1 it was described that the creation of the pie-filter is done by looping over the entire matrix and set each cell of the matrix to zero or one depending on if the frequency corresponding to each cell is to pass the filter or not. Denote the number of pixels in the x-direction of the image n_x and corresponding number of pixels in y-direction n_y as in Figure 9.5. The pie-filter operates on square matrices which means $n = n_x = n_y$ From the theory about time complexity given in Section 6.2, the time complexities presented in Table 9.4 are derived.

Part	**Time complexity**
Create filter	$O(n^2)$
FFT	$O(n^2 \log_2(n))$
IFFT	$O(n^2 \log_2(n))$

Table 9.4: Time complexity of identified bottlenecks in Section 9.1.1 for a $n \times n$ matrix

9.1.3 Data structures for speedup in C++

C++ provides a number of container classes. A container class is an object in which a collection of other objects are stored, i.e., its elements. One such class is `vector`, which is an array with dynamic size which can check that the index to access it is not out of bounds [4].

The matrices of the pie-filter was initially implemented using the class `vector`. In Figure 9.6, this means that `matrix` was a vector. This was due to avoiding doing a manual check of indices. At the final implementation this was changed and this change resulted in a considerable speedup of the code. Instead memory is allocated and a pointer to this memory location is used as it would have been done in C.

Accessing specific elements in the matrix requires calculating the index at which it is stored in the array. This corresponds to looping over rows and columns and in each iteration do a call to the `get` function illustrated in Figure 9.6. An improvement was to instead get a pointer to `matrix` and then simply loop over the array.

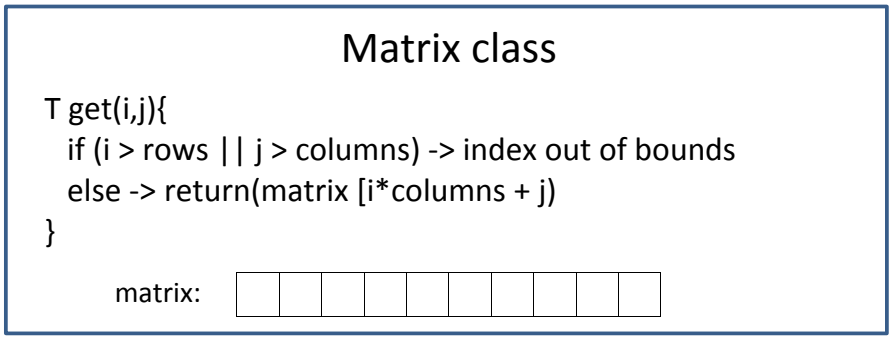


Figure 9.6: Access elements in matrix class

9.1.4 Sequential C++ code

Based on the MATLAB code was the C++ code developed. The execution time for the sequential code is described in Table 9.5 and Table 9.6. Observe that almost all operations are faster in C++ than MATLAB except for *Format pie-filter* and *Element-wise multiplication*. These operations contain matrix operations which MATLAB has very efficient ways of doing, whereas they have a low degree of optimization in the C++ code. As described in Section 6.4, MATLAB calculates a normalized FFT and C++ an unnormalized, which is the reason the normalization is included in the table. Otherwise, the result from MATLAB and C++ would not be comparable.

Part	Time [s]
preprocessing	1.6
Filter construction	5.8
Filtering image	11.7
Post-processing	0.9
Total	20.0

Table 9.5: Running time of C++ code coarsely divided as in Figure 9.2.

Part	Time [s]
Create pie-filter	5.2
Format pie-filter	0.6
FFT	4.4
Element-wise multiplication	1.4
IFFT incl. normalization	5.9

Table 9.6: Running time of C++ code of subparts in Figure 9.3

Finally, the execution times of the subparts of the preprocessing block are presented in Table 9.7. It is clear from this that the largest contributor to the execution time is the *Find nonzero* part.

Part	Time [s]
Find nonzero	1.3
Calculate filter properties	0.5
Calculate flight properties	0.5

Table 9.7: Running time of C++ code of subparts in Figure 9.4

FFT

In Table 9.6, it is clear that the FFT calculation is a heavy computation as about 50% of the execution time is spent on these computations. Due to this, it is studied which alternative ways there are of doing the 2D FFT computation using FFTW.

Approaches for running FFT on 2D matrix with FFTW:

I. FFTW performs a 2D FFT. See Algorithm 2.

II. Iterate over all rows and let FFTW perform 1D FFT on each row. Then do a corner turning so the array representing the matrix is stored in column-major order. Next, iterate over all columns and let FFTW perform 1D FFT on each column. Finally, do a corner-turning again so that the array is back in row-major order. See Algorithm 3.

III. Let FFTW perform FFT on each row separately. Next, let FFTW perform FFT on each column. See Algorithm 4.

Approach I means that the programmer only needs to present FFTW a matrix stored in row-major order and tell it to perform a 2D FFT on it. Approach I is the approach which has been used this far. In Approach II the program iterates over the rows in a for-loop. For each row call a for-loop iterating over all rows and all columns. At each iteration FFTW performs a 1D FFT on the current row or column. The third approach is in one sense a combination of the first two. First, let FFTW perform FFT on the rows. Second, let FFTW perform FFT on the columns.

Algorithm 2: Pseudocode for Approach I

```
fftw_2d(#rows, #columns, matrix);
```

Algorithm 3: Pseudocode for Approach II

```
foreach row in matrix do
 | fftw_1d(length of row, row);
end
row-major order → column-major order;
foreach column in matrix do
 | fftw_1d(length of column, column);
end
column-major order → row-major order;
```

Algorithm 4: Pseudocode for Approach III

```
fftw_many_1d_on_rows(length row, #rows, matrix);
fftw_many_1d_on_columns(length column, #columns, matrix);
```

In Table 9.8 are the running times of the different approaches presented. From this table it is clear that using Approach I in FFTW is the best choice for the sequential case. In Table 9.8, the execution time for the different parts are given in the parenthesis. For Approach II, this means that FFT takes 1.3 sec to compute and corner turning takes 2.1 sec. In the same way, FFT on rows takes 1.3 sec and 12.6 sec is the running time for FFT on columns. From these numbers, it can be concluded that for this input a corner turning is less expensive compared to a cache miss. An observation when running Approach III is that the execution time varies more than for the other approaches. Sometimes this approach is up to four seconds faster than the value given in the table. The value in the table is the slowest that has been observed then all cores have appeared available.

Approach	**Time [s]**
I	4.4
II	6.7 $(1.3 + 2.1 + 1.3 + 2.1)$
III	$13.9(1.3 + 12.6)$

Table 9.8: Sequential running time of approaches for FFT with FFTW.

9.1.5 Identification of operations to be parallelized

There are two things that are considered when deciding which parts to parallelize: time-consuming parts of the code and parts containing for-loops. The first is parallelized because speedup is crucial and the latter because OpenMP is efficient at parallelizing for-loops and it is easy and rather safe to parallelize for-loops in OpenMP. The parts described in Table 9.9 are chosen for parallelization and it is also described why they are chosen. The part *Create pie-filter* is marked as being computationally heavy of the execution but it is also a for-loop. This means that being computationally heavy is the reason for the parallelization, and the fact that it is a for-loop is simply a property of the implementation. In Section 9.1.4, three methods for running FFT were introduced and one of these methods explicitly used for-loops, this is ignored for the moment and Approach I is considered. The parallelization is multi-issue and data parallel.

Part	**For-loop**	**Computationally heavy**
Find nonzero	x	
Create pie-Filter		x
FFT		x
Element-wise multiplication	x	
IFFT		x
Format image	x	

Table 9.9: Parts chosen to be parallelized and the reason they are chosen

It is now possible to calculate the attainable speedup according to Amdahl's law described in Section 3.3.3. The parts in the processing called *Calculate filter properties* and *Calculate flight properties* need to be sequential. However, it is possible to run them parallel to each other. Since they have the same execution time, the total time of the sequential part of the code $T_s = 0.5$. It is assumed that the rest of the operations can be perfectly parallelized. This assumption is valid since it will only result in a looser upper bound. The reason this is a loose upper bound is that parallelization introduces overhead and the rest of the operations are not perfectly parallelizable. From Table 9.5 it is known that the total running time $T_{tot} = 20.0$. The fraction f can be computed as $f = T_s/T_{tot} = 0,025$ which results in $S = 40$ according to Amdahl's law, Equation (3.3). This means that no matter how many cores are available the speedup can never exceed 40.

9.1.6 Parallel C++ code

Based on the operations identified in Table 9.9, the parallelization was implemented. The for-loops were parallelized as described in Chapter 5 and in particular Section 5.1.2. Special attention had to be paid to *Find nonzero* in order to avoid race conditions. This means that the code has some critical section which only can be accessed by one thread at the time, which decreases the parallelism of the operation. Table 9.10 contains the resulting execution times for the more coarsely divided code presented. The results of parallelizing the computationally hard parts are presented in Table 9.11.

Further down in this section a deeper analysis of parallelization of the different methods for FFT will be given, but initially only Approach I, FFTW performs a 2D transform, is considered. In this approach, FFTW takes care of the parallelization as described in Section 6.3.1, i.e. by distributing the rows and columns over the threads and then compute the 1D transform over all rows and all columns.

Part	Time [s]
Preprocessing	0.5
Filter construction	1.1
Filtering image	2.8
Post-processing	0.1
Total	4.5

Table 9.10: Running time of C++ code coarsely divided as in Figure 9.2.

Part	Time [s]
Find nonzero	0.5
Create pie-filter	0.5
FFT	1.2
Element-wise multiplication	0.1
IFFT + normalization	1.5
Format image	0.1

Table 9.11: Running time of parts listed in Table 9.9.

FFT

This far, the results have been under the assumption that Approach I (FFTW performs a 2D FFT) is used. A problem when running 2D FFT with FFTW is that if allowing more than four threads, the execution time is worse than sequential. Even though the 2D FFT was expressly the best for the sequential case, it is an issue that if all threads cannot be used as this can prevent efficient scaling. Therefore, the alternative ways of computing FFT become interesting. The methods and the ways they are parallelized are described below. The results of parallelizing using these methods are described in Table 9.12. In the parentheses are the execution times of each part in the approach given.

I. FFTW performs a two dimensional FFT and also takes care of the parallelization using OpenMP. See Algorithm 5.

II. First iterate over all rows and perform FFT on each row. Then do a corner turning so the array representing the matrix is stored in column-major order. Next, iterate over all columns and perform FFT on each column. Finally, do a corner-turning again so that the array is back in row-major order. The iterating of each for-loop can be divided among threads using OpenMP. The corner turning is also a for-loop and is thus parallelized in the same way as the FFT. See Algorithm 6.

III. Let FFTW perform FFT on each row separately. Next, let FFTW perform FFT on each column. FFTW is taking care of the parallelization of the two parts itself using OpenMP. See Algorithm 7.

Algorithm 5: Parallelized pseudocode for Approach I

```
fftw_num_threads(4);
fftw_2d(#rows, #columns, matrix);
```

Algorithm 6: Parallelized pseudocode for Approach II

```
#pragma omp parallel for num_threads(get_max_num_threads());
foreach row in matrix do
 |  fftw_1d(length of row, row);
end
row-major order → column-major order;
#pragma omp parallel for num_threads(get_max_num_threads());
foreach column in matrix do
 |  fftw_1d(length of column, column);
end
column-major order → row-major order;
```

Algorithm 7: Parallelized pseudocode for Approach III

```
fftw_num_threads(get_max_num_threads());
fftw_many_1d_on_rows(length row, #rows, matrix);
fftw_many_1d_on_columns(length column, #columns, matrix);
```

Approach	Time [s]
I	1.2
II	0.8 $(0.1 + 0.3 + 0.1 + 0.3)$
III	1.5 $(0.1 + 1.4)$

Table 9.12: Parallel running time of approaches for FFT with FFTW.

9.1.7 Summary

This section summarizes the results found regarding the pie-filter and present comparisons between the MATLAB, C++ sequential and C++ parallel code. Earlier results will be used, which means that for the sequential C++ code Approach I (2D FFT) will be used, but for the parallel C++ code Approach II (1D FFT with for-loops) will be used since it has been established that it is the fastest. It is reasonable to use different approaches for computing FFT since in the formal definition of speedup, which was given in Section 3.3.1, it was said that the comparison should be between the parallel algorithm and the fastest sequential algorithm.

Table 9.13 presents the running times of the coarsely divided segments of the code. If first studying the total running time, the speedup of going from MATLAB to C++ is about 9.2. In Section 9.1.5 it was found that the speedup cannot be greater than 40 when parallelizing the C++ code. From Table 9.1 it can be derived that S =5.6 which means it is within the expected limit. The operations which cannot be parallelized do not depend on the input size, which means that Gustafson's law can be applied. With the currently used hardware this would be that as the input size goes to infinity (becomes very large) the speedup will approach twelve, which is the number of cores. By using the result that the speedup can never be greater than 40, the pie-filter will, with the input size that has been used while testing (6001×6001 pixels), never be faster than 0.5 seconds. However, as was pointed out when the upper bound of the speedup was calculated: this is a loose upper bound. This means that the upper bound is in reality tighter and an execution time of 0.5 seconds is most likely not achievable.

Part	MATLAB sequential time	C++ sequential time	C++ parallel time	Optimal parallel execution time
Preprocessing	6.2	1.6	0.5	0.1
Filter construction	11.2	5.8	1.1	0.5
Filtering image	14.4	11.7	1.9	1.0
Post-processing	1.5	0.9	0.1	0.1
Total	33.3	20.0	3.6	1.7

Table 9.13: Running time of code coarsely divided. The times are given in seconds.

Table 9.14 presents the speedup of each step. Column one shows the improvement due to change of programming language. The most interesting about this table is, however, column two and three. Column two is interesting because it shows the improvement due to parallelization and can be compared with a linear speedup which in this case is twelve. It is also interesting because it shows how the execution time changes in relation to the number of cores. Column three on the other hand is interesting because it shows the total improvement and can be used to draw conclusions on whether a real-time performance is possible.

Part	**↓ MATLAB to ↓C++**	**↓C++ to ↓↓C++**	**↓MATLAB to ↓↓C++**
Preprocessing	3.9	3.2	12.4
Filter construction	1.9	5.3	10.2
Filtering image	1.2	6.2	7.6
Post-processing	1.7	9.0	15.0
Total	1.7	5.6	9.2

Table 9.14: Speedup for code coarsely divided. ↓ = sequential, ↓↓ = parallel

In Section 9.1.5 it was decided which parts to parallelize. Since only specific parts of the code are parallelized, it cannot be expected that the execution times of the coarsely divided code will be optimal. It is , however, an interesting comparison in that sense that it shows the potential.

The parts that are parallelized are compared with an optimal parallelization in order to see how well the code parallelizes. The execution time of the parallelized parts are summarized in Table 9.15. In this table, it is also shown what the running time would be with an optimal parallelization. The running time is improved by translating from MATLAB to C++ except for the element-wise multiplication.

Part	MATLAB sequential time	C++ sequential time	C++ parallel time	Optimal parallel execution time
Find nonzero	2.5	1.3	0.5	0.1
Create pie-filter	10.8	5.2	0.5	0.4
FFT	5.7	4.2	0.8	0.3
Element-wise multiplication	0.6	1.4	0.1	0.1
IFFT + normalization	8.1	6.1	1.0	0.5
Format image	1.5	0.9	0.1	0.1

Table 9.15: Running time of parallelized parts of code (given in Table 9.9). The times are given in seconds.

Looking at Table 9.15, the running times are pretty close to each other, but since the goal is to have a code that is scalable with regard to the input it is important to know how good the improvement is relative to an optimal speedup. This can be seen in Table 9.16. For an optimal parallelization the speedup would be twelve. In the table it shows that the parts of the code with for-loops (see Section 9.1.5) efficiency is at least 75% of the optimal value whereas FFT only has an improvement of about 50%. A deviant value is that of *Find nonzero*, which is a for-loop, but it only has an improvement of about 20%. A clearly deviant value is the one of the *element-wise multiplication* which shows a super-linear speedup and an efficiency exceeding one. The reasons behind these results will be discussed in Section 10.

Part	Speedup ↓C++ to ↓↓C++	Efficiency
Find nonzero	2.6	0.22
Create pie-filter	10.4	0.87
FFT	5.5	0.45
Element-wise multiplication	14.0	1.17
IFFT + normalization	5.9	0.49
Format image	9.0	0.75

Table 9.16: The speedup and how well it parallelizes compared to an optimal parallelization. ↓ = sequential, ↓↓ = parallel

FFT

It has been pointed out that the computation of FFT can only be run with four threads for a 2D FFT. In Table 9.17, it is shown that for the alternative methods the speedup in time is significant and therefore it has also been studied how large the improvement it is in relative terms. Table 9.18 presents the speedup for each FFT approach. The goal is to get close to twelve. Approach I is efficiently parallelizing with respect to the fact that it is using at most four threads, which is shown in the parenthesis. However, since it cannot use as many threads as the other approaches, it will quickly become slow in comparison with the other approaches if the system is scaled. The content of Table 9.18 shows that all three approaches are interesting and useful. Approach I is useful for sequential FFT and for programs only using a few threads. Approach II on the other hand parallelizes rather well and is the fastest for this input size. Finally, Approach III is having the largest speedup and will probably be the best if the system is scaled.

Approach	Sequential time	Parallel time	Optimal parallel execution time
I	4.4	1.2	0.4
II	6.7 (1.3+2.1+1.3+2.1)	0.8 (0.1+0.3+0.1+0.3)	0.6
III	13.9(1.3+12.6)	1.5 (0.1+1.4)	1.2

Table 9.17: Running time of FFT for different approaches. The time given in parenthesis in the time for the different parts of the FFT. The times are given in seconds.

Approach	Speedup ↓C++ to ↓↓C++	Efficiency
I	3.7	0.31 (0.92 for 4 threads)
II	8.4	0.70
III	9.3	0.77

Table 9.18: The improvement of running time by parallelization for different approaches of FFT and how well it parallelizes compared to an optimal parallelization. ↓ = sequential, ↓↓ = parallel

9.2 Matching

Matching is the second step in change detection as illustrated by Figure 9.7 This section presents some results for the matching. No C++ implementation has been done yet. Thus, the results presented here are theoretical and are meant to be a foundation for future work. Matching can use some results from the pie-filter, e.g. the FFT computation. The input to matching is two SAR-images that usually are pie-filtered. It is possible to run matching without having done pie-filtering first if the flight paths are very similar.

Figure 9.7: Matching, the current position in the change detection processing chain.

The implementation in Algorithm 1 contains a for-loop which loops over all subimage pairs. This for-loop contains, unlike the pie-filtering, some heavy computations. Inside the for-loop is the computation of FFT, IFFT and convolution.

Outside the for-loop are a number of simple operations such as assignments, calculation of mean and median. The operations taking place outside are simple, hence, the time complexity of the matching algorithm will only depend on the for-loop. There are no data dependencies between the iterations of the for-loop. The iterations are not updating the same variable outside the loops either. Hence, the for-loop is possible to parallelize.

Denote the number of columns and rows of the subimage m_x and m_y. Further, the number of subimages are denoted n_x and n_y. These variables are also described by Figure 9.8.

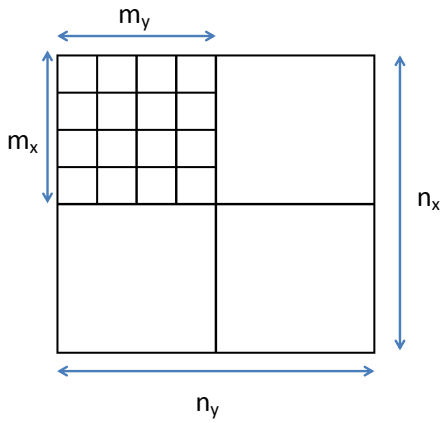


Figure 9.8: Image containing $n_x \times n_y$ subimages each subimage of size $m_x \times m_y$

The exact implementation of convolution in MATLAB is not known, but suppose that it uses FFT as many algorithms does for calculating the convolution. The convolution of two matrices is computed according to Equation (9.1) as was described in words in Chapter 6.

$$conv(a, b) = fft(a) \cdot fft(b) \tag{9.1}$$

In this case the convolution is between the subimage and a 3×3 matrix (the Sobel kernel). The size of the subimage matrix is assumed to be bigger than 3×3, thus, at least as large as the kernel. This assumption means that the size of the subimage will be critical for the running time. The running time of FFT was given in Section 6.2.

FFT is run on every subimage and IFFT is run on the product of the conjugate of one subimage and another image, i.e. $IFFT(subimage1^* \cdot subimage2)$. The multiplicands $subimage1^*$ and $subimage2$ have the same size. Finally, since the images (but not necessary the subimages) are square matrices, the simplification $n = n_x = n_y$ can be done. From these properties of the code, the time complexities in Table 9.19 are derived.

Part	**Time complexity**
Convolution	$O(m_x m_y \log(m_x m_y))$
FFT	$O(m_x m_y \log(m_x m_y))$
Matching	$O(n^2 m_x m_y \log(m_x m_y))$

Table 9.19: Time complexity of matching for a $n \times n$ matrix with dimensions according to Figure 9.8 and $n = n_x = n_y$

10 Discussion

Within the time frame of this project, the pie-filter has been completed and a small theoretical study of the matching. In this chapter, the results from these parts are discussed. Among the results to be discussed are the possibilities for parallelization, the observed superlinear speedup and the different approaches for computing FFT. This chapter also points out what ethical aspects there are for this research. The discussion is conducted based on the questions from the problem description:

- Is change detection parallelizable?
- If it is parallelizable, how can it be parallelized?
- Is parallel change detection scalable?
- Is the translation to C++ and parallelization enough for a real-time performance?

The first question is trivial to answer, since change detection is a processing chain, pipelining (see Section 3.2.3) can be used. In the simplest case, each step in the process is assigned to one core. The pie-filter and matching can be run on several cores and the rest of the process needs to be evaluated further in order to know if parallelization is possible. In Figure 8.4 we saw that a parallelization of the pie-filter was possible at high level, i.e., that the construction of the pie-filter and the FFT of the image were independent parts that could be run in parallel. It was also mentioned in Section 8.1.1 that there were independent subparts of the preprocessing which could then be parallelized. Also, Section 6.2 and 6.3 about FFT and FFTW indicated that some parallelization would be possible on lower levels of the program.

10.1 Realization of parallelization

The second question, how the parallelization can be performed, demands a more elaborate answer. It was presented in Section 9.1.6 that 2D FFT could only be run with maximum four threads. If allowing a higher degree of parallelization, the execution time got worse than sequential. This result might appear quite counterintuitive, but there are several possible reasons for this. An overall explanation is that the more threads that are used, the bigger the overhead becomes. For example, the load balancing is more complicated for more threads, since the more fine grained the problem is the more possible solutions there are to the load balancing problem. It was explained in Section 6.3.1 that a parallel 2D FFT is in fact many 1D FFT divided among the cores. It has also been found in the project that if running the FFT with Approach II or III, the FFT could be parallelized using all available cores. This indicates that the problem lies in the implementation of the parallelization of the 2D transform and its interaction with OpenMP. It could be interesting to look into the implementation of FFTW. It was, however, shown in Table 9.18 that if running 2D FFT with four threads it has a very good performance compared to an optimal parallelization using four threads.

If choosing to do the parallelization on a higher level, e.g., by running the filter construction and FFT in parallel, which means that none of them can use all cores. An interesting concept to try would be to run the FFT on four cores and the construction of the pie-filter in parallel. Then, we could take advantage of the fact that the parallel 2D transform is very efficient for a small number of threads and since the parallelization of the filter construction will be easier to parallelize, it might have an execution time closer to the parallel one. Let us continue to evaluate this idea further. It was shown in Table 9.11 that *Filter construction* takes 1.1 seconds and Table 9.12 showed that Approach I has running time 1.2 seconds. Thus, currently these two takes about the same time to execute. In addition, it has been shown in Table 9.5 that *Filter construction* takes 5.8 seconds in the sequential case. Suppose *Filter construction* is optimally parallelized over eight cores. Then the parallel running time of *Filter construction* would be 0.7 seconds. This results in FFT + *Filter construction* taking 1.2 seconds instead of 1.3 as these two added takes running *Filter construction* and Approach II over all cores.

Another interesting result from the parallelization was the operation called *find nonzero*. It was shown in Table 9.16 that its efficiency is about 0.22 which is bad in comparison to the rest of the parallelized code. The reason for this is most likely that special attention was needing in order to avoid race conditions, which were described in Section 9.1.6. Each thread finds their minimum and maximum value and when all threads have finished, they enter a critical section where the global minimum and maximum values are updated.

In Table 9.16 it could be seen that the *Element-wise multiplication* showed a superlinear speedup, which is very unusual, as described in Section 3.3.1. A possible explanation to this result could otherwise be that the execution times are not exact and therefore it could be that we were getting a good parallel time and a bad sequential time. A similar explanation is that the execution time has been rounded to one decimal. This means that the parallel execution time of *Element-wise multiplication* could in fact be 0.14 seconds which would mean that the speedup is 10.0. The reason the execution times are not given with a higher precision is that it would give a false impression of that the times are more exact than they actually are.

It was described in Section 9.1.3 that the choice of data structure can have a big impact on the execution time. We could see that the more we used the data-structures that C++ has common with C, the better the execution time. A possible way of making the code even faster could be to make the data structures used to a bigger extent be the ones that also exist in C. In the current implementation, the focus has been on the matrices since they are very large.

A theoretical study of the matching step was also conducted. Matching showed good possibilities for parallelization. The question here is whether parallelization will be enough. It could be seen in the pie-filter that FFT is time-consuming to compute and in matching many FFTs will need to be computed, but they are also much smaller.

There are still some things that can be done to speedup the pie-filter further, such as using `inline` and parallelize more of the code. But since the code is already rather fast, it is better to continue with the rest of change detection and speedup the rest first in order to see how far away a real-time performance is before doing these small improvements that will give a small improvement in execution time.

It was mentioned in Section 6.3 that this FFTW is generic with regard to hardware it needs some time to adapt to the specific hardware. If the same program is run many times of the same sized matrices it is possible to reuse the settings that has once been calculated, which can reduce the execution time of FFTW. This feature is called `wisdom` in FFTW [11].

10.2 Scalability

In this section we will answer the third question, the scalability of the system. A possible approach for further parallelization was to execute the filter construction and FFT in parallel. A problem with the fact that the FFT needs exactly four cores, is that it is not flexible and scalable. If more cores are available, then it is necessary to do some testing in order to find a new optimal number of cores. Thus, there might be ways of making the code more efficient by letting the FFT run on four cores, but this is not a scalable solution.

It was shown in Section 9.1.7 that the different approaches for FFT can be used at different occasions. This opens up for making the FFT computation more intelligent by using different methods depending on if the code is run in parallel, the input size and number of available threads. Having a solution like this would also make the implementation more scalable and flexible.

10.3 Real-time performance

In this section the fourth question, the possibility of a real-time execution, is discussed. In Section 9.1.7 it was presented that the final running time of the pie-filter is 3.6 seconds. The total time available for change detection processing, for this input size, is 30-40 seconds which means that the pie-filter will need about 10% of this time. Having looked briefly at the rest of the change detection process, the pie-filter is one of the fastest parts even before a parallelization. Considering there are five steps in processing the result from the pie-filter do not eliminate the possibility of a real-time performance, even though it cannot give any guarantees.

It was pointed out in Section 9.1.4 that the sequential execution time of Approach III has not been as stable as for the other approaches. This can become a problem in a real-time application since a predictable execution time is crucial.

10.4 Methodology

To work iteratively with each step of the change detection process has been a good setup. It has made it possible for someone else to continue the work without having to go into depth with what has been done in this project. Using this method it is enough to only keep one part of change detection in mind when working and it is a good setup for getting a deeper understanding of the current algorithm.

The downside of this setup is that one does not know much of the rest of the chain, e.g., which effects the choices in the implementation taken at an early stage will have later on. It is also hard to do an estimation of the possibility of a real-time performance before having completed the entire process. Parallelization one step at the time can also make it more difficult to find a good level of parallelization. After all, the goal is to make change detection run in real-time, but by completing one step at the time it is hard to know when it is good enough. Thus, there is a risk of making the code faster than necessary or stop working on speeding it up before it is fast enough. In the latter case one can go back and in an iterative manner make the code faster until it runs in real-time. However, in the first case there is a risk that the project takes far longer time to finish than necessary or that a lot of work is put into the project before it is found that real-time is infeasible. To work more iteratively over the entire change detection process also means constantly refreshing the knowledge of the algorithms.

There has only been a very limited number of test files available which has made it difficult to test more unusual cases such as when major formatting of image needs to be done. The verification has been done by comparing with the MATLAB output, which assumes that the MATLAB code is perfect. It would have been good if it was possible to do a more systematic testing of the functions independently of the MATLAB output too.

10.5 Ethical aspects

Change detection can contribute with much good, e.g., imagine a dense rainforest, the government can let a CARABAS system fly over and change detection can find if any drug factory has been constructed in the forest beneath the foliage. In a land at war, change detection can detect if enemies are trying to hide in the forest or beneath camouflage. The fact that change detection is running in real-time would enable faster actions to be taken when a detection is found. If a land is at war it might be crucial to stop the enemy as quickly as possible to limit the damages. In a civil application, there is the example when the storm Gudrun had hit Sweden, CARABAS was used to determine how many trees there were left lying on the ground even though they might be lying among standing trees, hiding the view in ordinary aerial photographs. Since the CARABAS system works in all light and weather conditions it can always be used, e.g., to find a person whose gone astray by car in a snowstorm.

All advanced surveillance systems like the CARABAS system can on the other hand result in terrible consequences, if used by the wrong people. Imagine a dictatorship where rebels are trying to introduce democracy, then the dictator can detect that an army is gathering in a forest and prevent them from succeeding. Another aspect is private integrity, where a surveillance system makes it possible to know to some extent what the citizens are doing. Change detection in real-time is planned to do all the calculations in the aircraft and then send only the coordinates of detections to the ground. Theoretically speaking the detections could be sent to a drone which bombs those locations. The point is you cannot know how the person or system receiving the coordinates is going to use them.

11 Conclusion

Change detection in real-time is a concept with large potential. It makes it possible to fast and accurately find changes on the ground. With the current MATLAB implementation, real-time performance is not attainable. The results of this report shows that there is a good chance being able to run change detection in real-time. Since the entire change detection process has not been analyzed in this project, nothing definite can be said of the real-time performance.

This report has shown that the pie-filter, which is a part of change detection, can be speeded up about nine times compared to the sequential MATLAB code by translating the code to C++ and parallelize it. The pie-filter has a running time of approximately 3.5 sec and the time available for doing change detection is about 30-40 sec for an image of size 6000×6000 pixels. Thus, about 10% of the available time will be used for pie-filtering. Since the remaining steps of change detection are quite heavy it will still be challenging to get the entire process to finish within the time limit.

The work has also included a study of different approaches for computing 2D FFT using the library FFTW. Three different approaches has been presented and evaluated. This study has shown that the different ways of computing 2D FFT are all good, but for different occasions. One method is good in the sequential case, another method works very well for this input size and scales pretty good whereas the last method is the one which gives the largest speedup which makes it relevant if the system and input is scaled.

Since there are several steps left in change detection it seems natural to continue the work parallelizing these in the same manner. In the pie-filter part, a lot of results were found that can be used when encountering similar problems later on in the process. As mentioned in Section 2.2 there is a new algorithm under development which could replace the last three steps in the processing chain and it would be interesting to see if it could be parallelized too. It would also be intersecting to see which one will be fastest and which one will provide the most accurate result. The hardware supporting parallelism are constantly improving and a possible future work could be to see how change detection can handle a high level of parallelism, like hundreds or thousands of threads.

References

[1] R. Alecsandrescu, F. Pop, and V. Cristea. A distributed algorithm for critical area detection in satellite imagery. In *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 380–386. IEEE, 2008.

[2] Carabas - foliage penetration radar. `http://saab.com/air/sensor-systems/ground-imaging-sensors/carabas/`. Accessed: 2015-05-19.

[3] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, Calif, 2001. Available from: `library.books24x7.com`. Accessed: 2015-04-07.

[4] Containers. `http://www.cplusplus.com/reference/stl/`. Accessed: 2015-05-20.

[5] Yuefan Deng and ebrary (e-book collection). *Applied parallel computing*. World Scientific, Hong Kong; Singapore, 2013.

[6] Maitre H. (ed). *Processing of Synthetic Aperture Radar Images*. ISTE Ltd and John Wiley & Sons, Inc., 2008.

[7] Zelnio E.G., editor. *Algorithms for Synthetic Aperture Radar Imagery VI*, volume Proc. SPIE 3721, Orlando, FL, April 1999.

[8] Fftw. `http://www.fftw.org/`. Accessed: 2015-04-20.

[9] Parallel versions of fftw. `http://www.fftw.org/parallel/parallel-fftw.html`. Accessed: 2015-04-20.

[10] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley, Reading, Mass, 1995. Retrived from `http://www.mcs.anl.gov/~itf/dbpp/`. Accessed 01-04-15.

[11] Matteo Frigo and Steven G. Johnson. *FFTW for version 3.3.3*, November 2012.

[12] G. Haapalahti, D. Murdin, M. Blom, P-O Frölind, and L. Ulander. Implementation of real-time CARABAS-II image formation and change detection. Technical Report FOI-R–1609–SE, FOI, 2005.

[13] G. Haapalahti, L. M. H. Ulander, and M. Lundberg. Evaluation of spectral matching for change detection with CARABAS -II. Technical report, FOI (Totalförsvarets forskningsinstitut), 2006.

[14] Fast Fourier Transform (FFT). `http://se.mathworks.com/help/matlab/math/fast-fourier-transform-fft.html`. Accessed: 2015-02-03.

[15] fft2. `http://se.mathworks.com/help/matlab/ref/fft2.html`. Accessed: 2015-02-03.

[16] N. Mohanty. *Signal Processing: Signals, Filtering, and Detection.* Springer Netherlands, Dordrecht, 1988.

[17] Mian M. Mubasher, M. S. Farid, Abdul Khaliq, and Muhammad M. Yousaf. A parallel algorithm for change detection. In *15th International Multitopic Conference (INMIC)*, pages 201–208. IEEE, 2012.

[18] C. J. Oliver. Real-time sar change-detection on a parallel computer architecture. In *Second International Specialist Seminar on the Design and Application of Parallel Digital Processors*, pages 19–23, Lisbon, April 1991. IET.

[19] Peter S. Pacheco. *An introduction to parallel programming.* Morgan Kaufmann Publishers imprint of Elsevier, Amsterdam; Boston, 2011.

[20] A. Åhlander, H. Hellsten, K. Lind, J. Lindgren, and B. Svensson. Architectural challenges in memory-intensive, real-time image forming. In *International Conference on Parallel Processing*, pages 35–35. IEEE, 2007.

[21] K. R. Rao, D. N. Kim, and J. J. Hwang. *Fast Fourier transform: algorithms and applications.* Springer, Dordrecht; New York, 2010.

[22] Thomas Rauber, Gudula Rünger, and SpringerLink (e-book collection). *Parallel programming: for multicore and cluster systems.* Springer-Verlag, Berlin, 2010.

[23] Jesus A. Segoviano. Parallel strategies for sar processing. In *Passive Optical Remote Sensing of the Atmosphere and Clouds IV*, volume 5652, pages 174–181, 2004.

[24] L. M. Ulander, B. Flood, P. Follo, P-O Frölind, A. Gustavsson, T. Jonsson, B. Larsson, M. Lundberg, W. Pierson, and G Stenström. Flight campaign Vidsel 2002 CARABAS-II change detection analysis. Technical Report FOI-R–1001–SE, FOI, Linköping, 2003.

[25] L. M. Ulander, B. Flood, P. Follo, P-O Frölind, A. Gustavsson, T. Jonsson, B. Larsson, M. Lundberg, W. Pierson, and G Stenström. Flight campaign Vidsel 2002 CARABAS-II forest report. Technical Report FOI-R–0962–SE, FOI, Linköping, 2003.

[26] L. M. Ulander, P.-O. Frölind, A. Gustavsson, H. Hellsten, and B. Larsson. Detection of concealed ground targets in carabas sar images using change detection. In E.G. [7].

[27] L. M. H. Ulander, M. Lundberg, W. Pierson, and A. Gustavsson. Change detection for low-frequency sar ground surveillance. *IEE Proceedings - Radar, Sonar and Navigation*, 152(6):413, 2005.

[28] Stimson G. W. *Introduction to Airborne Radar*. SciTech Publishing Inc., second edition, 1998.

[29] Ke Zhu and Shiyong Cui. Near real-time sar change detection using cuda. In *International Geoscience and Remote Sensing Symposium*, pages 2004–2007. IEEE, 2012.