UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Oskar Hint

Efficient parallel algorithms for synthetic aperture radar data processing using large-scale distributed frameworks

Bachelor's Thesis (9 ECTS)

Supervisor: Pelle Jakovits, MSc

Tartu 2015

Efficient parallel algorithms for synthetic aperture radar data processing using large-scale distributed frameworks

Abstract:
Processing radar satellite images is a considerable computing task due to large image sizes. Distributed computing can often be leveraged to speed up algorithms that are too time-consuming on a single machine. It is however unclear which radar image processing algorithms can be efficiently migrated to parallel environments and what is the proper way to implement them. Previous works have concentrated on parallel image processing as a general computing task but either the unique properties of radar images or newer distributed computing frameworks are not considered or only some specific algorithms have been examined. This thesis proposes a classification of radar image processing algorithms that can potentially be parallelized. Each class of algorithms is studied based on the properties of current popular distributed computing frameworks and file systems. Algorithms that best represent their respective classes are implemented using some concrete distributed computing framework. The classification simplifies the gauging of potential algorithms in terms of parallel speedup and provides general implementation steps, thus easing the task of leveraging distributed computing for radar image processing.

Keywords: Cloud computing, Distributed computing, Synthetic aperture radar, SAR, Image processing

Tõhusad paralleel-algoritmid radarsatelliidipiltide töötluseks kasutades suuremahulisi hajusraamistikke

Lühikokkuvõte:
Radarsatelliidipiltide töötlemine on märkimisväärse suurusega arvutusülesanne kuna piltide mõõtmed on äärmiselt suured. Hajusarvutust kasutatakse sageli et võimendada algoritme, mis jooksevad ühel arvutil liiga aeglaselt. Kuid on ebaselge, milliseid radaripiltide töötlusalgoritme on võimalik tõhusalt paralleelsetesse keskkondadesse ümber viia ning kuidas neid korrektselt implementeerida. Eelnevad tööd on keskendunud paralleelsele pilditöötlusele kui üldisele arvutusülesandele, kuid unikaalseid radarpiltide omadusi või uuemaid hajusarvutusraamistikke pole käsitletud või on käsitlus keskendunud mõnele üksikule algoritmile. Käesolev töö pakub välja potentsiaalselt paralleliseeritavate radaripiltide töötlusalgoritmide klassifikatsiooni. Iga algoritmide klassi uuritakse enimkasutatavate hajusraamistike ja -failisüsteemide omadustel. Kõige paremini mingeid klasse esindavad algoritmid implementeeritakse konkreetsetel tehnoloogiatel. Klassifikatsioon lihtsustab huvipakkuvate algoritmide võrdlust ja pakub üldisi implementatsioonisamme ning hõlbustab seeläbi hajusarvutuse rakendamist radarsatelliidipiltide töötlusel.

Võtmesõnad: Hajusarvutus, Pilvearvutus, Radarsatelliidipildid, Pilditöötlus

Contents

1 Introduction

1.1 Problem overview

The last decade has seen a huge increase in the amount of data generated in almost all walks of life. This has also been the case with satellite imagery, nowadays many airplanes carry optical or radar imaging equipment and there are several ongoing large-scale Earth monitoring programmes with the most extensive ones being NASA's Earth Observing System (EOS) which launched its first satellite in 1997 and the more recent European Space Agency's (ESA) Copernicus programme with 3 satellites being launched between 2014 and 2016. In Copernicus programme's case, satellite data is freely available in various different forms, meaning both public institutions and private companies have fast access to the latest images. The large quantity of available images and the growing number of imaging techniques employed has opened up new research opportunities and possibilities for real-world applications. With new satellite images generated for the same area in as little as 6 days, it is possible to study changes on Earth's surface on relatively small time scales, which in turn enables researchers to more efficiently tackle problems such as the monitoring of polar ice caps and deforestation, farmland usage, urban development, natural disasters' relief efforts and so forth.

1.2 Motivation

However, as has been the case for many other fields, the volume of data being collected also brings about many challenges in satellite imaging, especially regarding the storage, processing and visualisation of said data. Even if focusing on a certain small area of land, the number of images needing processing can be in hundreds and with image sizes in gigabytes it quickly becomes apparent that due to the sheer amount of resources and storage needed, this cannot be feasibly done using just a single machine running some processing algorithm.

Thus, in many ways the next logical step is to use several machines. Firstly, nowadays almost all universities and other research institutions have their own computing clusters capable of parallel processing. Secondly, with the the rise of infrastructure as a service (IaaS) providers such as Amazon and Google with Elastic Compute Cloud (EC2) and Google Compute Engine respectively, anyone can get quick access to parallel computing resources. Furthermore, there are great open source computing frameworks and distributed file systems freely available with great documentation and community support with the most widely-used one being the Apache Hadoop software library.

1.3 Limitations

Since at the time of writing, the only launched satellite of ESA's Coperincus programme is the Sentinel I which is equiped with a certain type of imaging equipment called Synthetic Aperture Radar (SAR), this thesis will focus on processing SAR data in particualar.

The distributed computing technologies chosen are Apache Spark and Hadoop Distributed Filesystem(HDFS). The common consideration for the technologies chosen was that both Spark and HDFS are widely-used among both researches and private companies. In addition, since at the time of writing Spark is still a relatively new framework and therefore the amount of research conducted on it is still considerably smaller than older more established frameworks such as MapReduce. Additional reason for choosing

HDFS as the file system is that as it is part of the Hadoop stack, it integrates well with other Apache products.

1.4 Goals

The main goals of this thesis are as follows:

- Determine which SAR data processing algorithms are good candidates to be parallely implemented and what is the correct way to do so
- Provide a framework in which different SAR data processing algorithms can be compared in terms of parallel implementation feasibility
- Implement some algorithms on HDFS and Spark to gauge parallelization speedup in practice
- Provide proof of concept for a general SAR data processing tool running on HDFS and Spark
- Propose real-world applications that build on this work
- Propose areas of future research
- Whenever possible generalize solutions to not only cover SAR data processing, but large-scale image processing as a whole

1.5 Contributions

As part of this thesis, a classification of SAR image processing algorithms was proposed. The classification simplifies estimating the parallel speedup potential of different SAR data processing algorithms and therefore answers the question, which algorithms can be efficiently moved to distributed frameworks. Specific example algorithms from the most important classes were implemented using Spark and HDFS to prove the feasibility of leveraging distributed computing for SAR data processing.

1.6 Outline

Chapter 2 gives the essential overview of the technologies used and explains what are SAR images, their characteristics, how they are produced and what sets them apart from other image types.

Chapter 3 contains the main part of this work, classification of potentially parallelizable SAR image processing algorithms is given. Each class of algorithms is studied to determine if they can be efficiently parallelized and implementations of example algorithms are discussed

Chapter 4 presents results of running the implemented algorithms on a Spark cluster

Chapter 5 concludes the thesis and provides areas of further research and ideas for possible real-world applications

2 Background

2.1 Relevant work

Relevant previous work to this thesis is largely lacking since almost no previous distributed image-processing works have considered SAR images. This is mostly since open access to a large variety of different SAR images of various locations has been a recent development and the computing framework used for the purposes of this thesis, Spark, is new as well. Some background can still be provided however.

The most closely related preceding work is SAR Image Denoising Using Non-Local Means on MapReduce by Võssotski. Võssotski shows that Hadoop MapReduce is well-suited for running parallel non-local means noise reduction on SAR images. However, no other algorithms were examined and thus the solutions proposed do not generalise well. Since the MapReduce model used imposes considerable restrictions to algorithms' implementations, coming up with more general solutions would have likely proved difficult. The file system used was HDFS and thus the storage and serialization of SAR images was considered. The images were converted to PNG format and stored as striped splits with certain overlap. However, this way of storing is not suitable for some SAR processing tasks since SAR-specific information regarding the waveform is lost. Also, since the image was split into long and narrow stripes, many sliding window algorithms might only work reasonably well if the window size is small.

Another slightly related work is Distributed Processing Of Large Remote Sensing Images Using MapReduce by Tesfamariam. Tesfamariam finds that MapReduce is an efficient framework that scales well for large-scale edge detection so long as the size of input is reasonably large[1] Tesfamariam also found that MapReduce model was simple in the sense that minimal changes were needed to the sequential algorithms. Since MapReduce algorithms can generally be implemented in Spark in similar fashion, this indicates that Spark is likely suitable for various remote sensing image processing tasks. Remote sensing images were considered from a broad and general spectrum and any SAR image specifics were not examined.

2.2 Technologies

2.2.1 Apache Spark

Motivation Apache Spark is a cluster computing framework currently maintained and developed by the Apache Software Foundation. Spark was originally built to address the most common issue with the single-pass batch processing model of MapReduce, which was the need to write intermediate results of a multi-job task to a distributed file system. This meant that iterative algorithms brought about considerable disk I/O overhead and made MapReduce slow for many common data processing tasks, such as machine learning and interactive data analytics and exploration. [ZCD+12a] As both of these tasks are common in image processing, using Spark for SAR data processing provides a larger number of potential algorithms to be efficiently implemented as well as faster querying. For example, a researcher can in principle load some images of a certain location to memory, then perform noise-reduction and run some change detection algorithm on the same data all without the need for multiple reads and writes to the underlying file system.

[1] In gigabytes.

Spark is also well integrated with Hadoop and many of the various systems used in tandem with MapReduce, such as different file systems or the YARN resource manager, work fine with Spark. [ZCD+12b] Furthermore, Spark is open source and freely available with considerable documentation and support, provides apis for 3 different languages - Java, Python and Scala, and puts no restrictions on the underlying hardware, making it well suitable for current work.

Cluster overview Spark's structure is in many ways similiar to MapReduce. Spark requires a cluster manager and a distributed storage system. Cluster manager's job is to deal with resource allocation, Spark supports standalone (native Spark cluster), Apache Mesos and Hadoop YARN cluster managers. For storage, Spark supports a wide range of different systems, such as HDFS, Amazon S3 and several others. The program specified by the programmer to use Spark is called a Driver program and contains a SparkContext object that connects to the cluster manager tell Spark to acquire executors on nodes for processing and storage, refer to figure 1. Application code is distributed to executors and SparkContext assigns tasks for the executors. Executors can be thought of as independent processes that when connected to the manager, make up the Spark cluster.

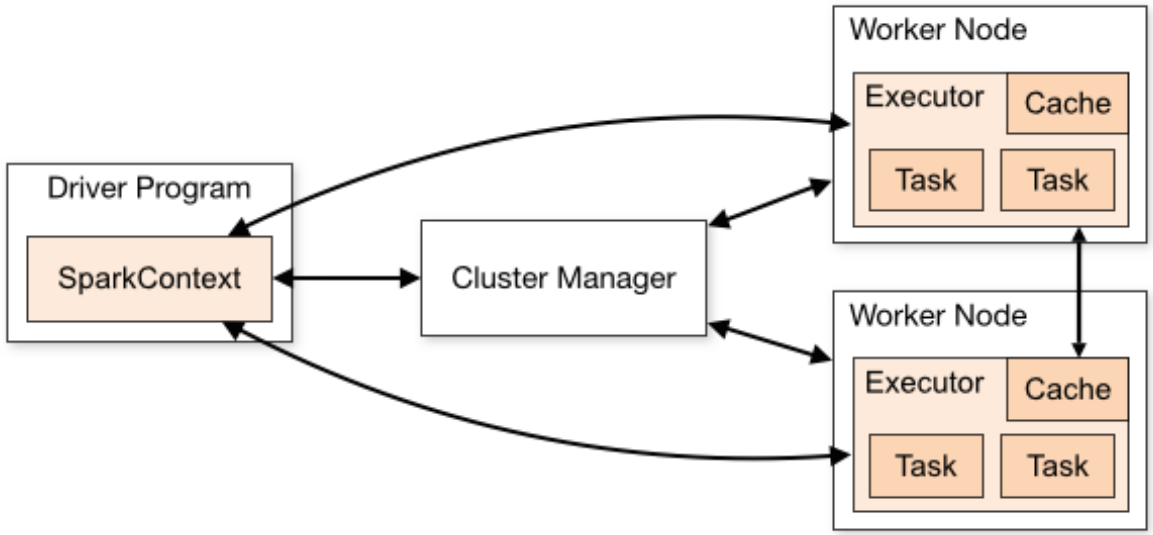

Figure 1: Spark cluster overview `https://spark.apache.org/docs/latest/cluster-overview.html`

Resilient distributed dataset The central abstraction in Spark is that of resilient distributed dataset (RDD) which is an immutable collection of elements partitioned across multiple machines that can be operated on in parallel. RDDs act as a storage primitive and a restricted form of coarse-grained memory as well as a programming abstraction. In Spark, RDDs can be created through operations on either data on the file system or other RDDs, by converting a collection to an RDD or by changing the persistence of an existing RDD. Since RDDs are stored in memory during execution, operations on them are considerably faster than maps in MapReduce for example. RDDs are aggressively thrown out of memory if they are not immediately needed so if a certain RDD is needed for multiple operations, appropriate cache or persist method has to be explicitly called. [ZCD+12c]

Programming model To use Spark, programmers need to provide a driver program that controls the high-level processing flow of their application. This is done by first creating an RDD, usually by reading some data from the file system, and then applying various parallel operations on these RDDs. In Spark programs, RDDs and operations are represented as language specific [2] objects and methods. RDD operations fall into two main categories: transformations and actions. Transformations are lazy functions that define new RDDs and actions prompt the actual computation by either returning a value or exporting data to storage.

Fault tolerance Fault tolerance is provided without replication, instead Spark tracks computation and any lost data is recomputed from the base data on disk. This means that elements of an RDD need not exist on the file system explicitly but instead contain the RDD's lineage - transformations used to derive a particular RDD from disk or other RDDs.

[2] Scala, Java or Python.

2.2.2 Hadoop Distributed File System

As a distributed storage platform, the Hadoop Distributed File System(HDFS), a scaleable and fault-tolerant distributed file system written in Java, was chosen. Since HDFS is part of the Hadoop stack, it works well with Spark and has great resources and support available.

In HDFS, data is stored in nodes. Metadata and the actual application data are kept separately with the former being stored on a single dedicated node called the NameNode and the latter on other nodes called DataNodes.

Files in HDFS are split into large blocks, by default 128MB in size, that are replicated accross multiple DataNodes, mapping of blocks to DataNodes and files, as well as management of the namespace hierarchy, is handled by the NameNode.

Once HDFS is properly set up, DataNodes send regular heartbeats (3s) to the NameNode to confirm that the DataNode is in operation as well as provide the NameNode with information regarding their storage capacity and ongoing data transfer. NameNode uses replies to heartbeats to send various instructions to DataNodes, such as the commands to replicate certain blocks to other nodes or to shut down the node. Faul toleration is one of the main principles of HDFS and it is mostly achieved by constant replicatio. Faults are considered the norm rather than the exception. Figure 2 provides a broad overview of HDFS architecture.

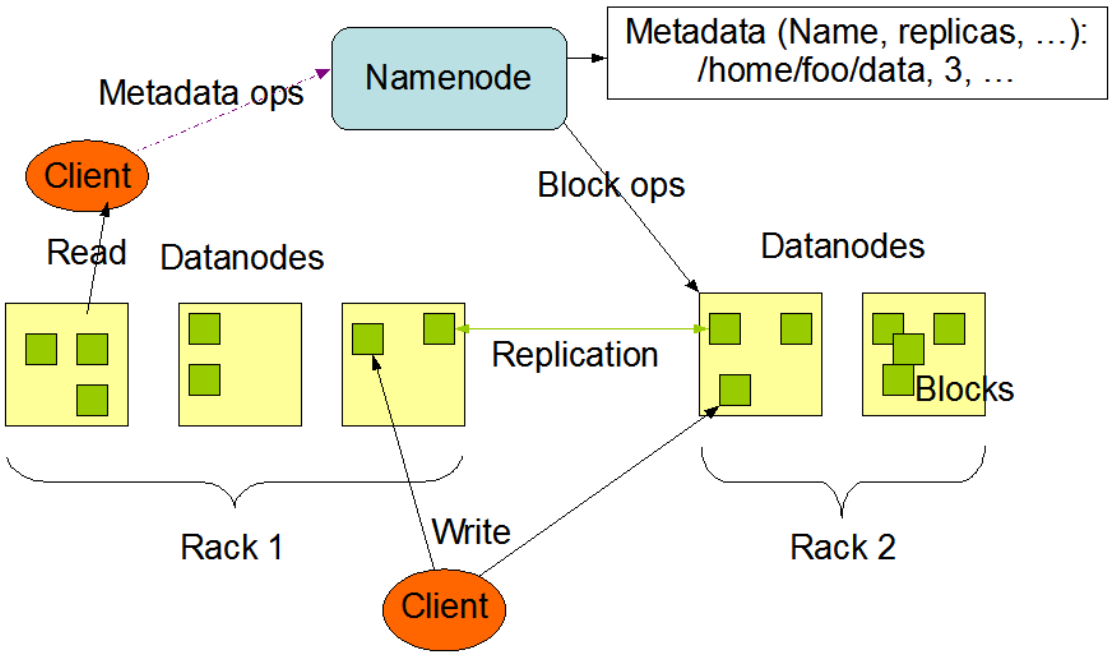

Figure 2: HDFS Architecture `http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

2.3 Synthetic Aperture Radar

2.3.1 Radar imaging basics

Typically radar imaging techniques work by emitting radio waves (called radar signals) in some direction and collecting the waves that are reflected back. Quality of the reflected signal is determined by properties of the object that the radio wave came in contact with. Radar signals are reflected especially well by metals, seawater and wet ground - materials that conduct electricity. Different materials have different reflection qualities and some cause the signal to scatter instead of reflecting back, this means that generally radar imaging techniques do not explicitly state that a certain object is from a particular material but rather convey the differences between different areas covered meaning that with additional information the actual material can be reasonably guessed.

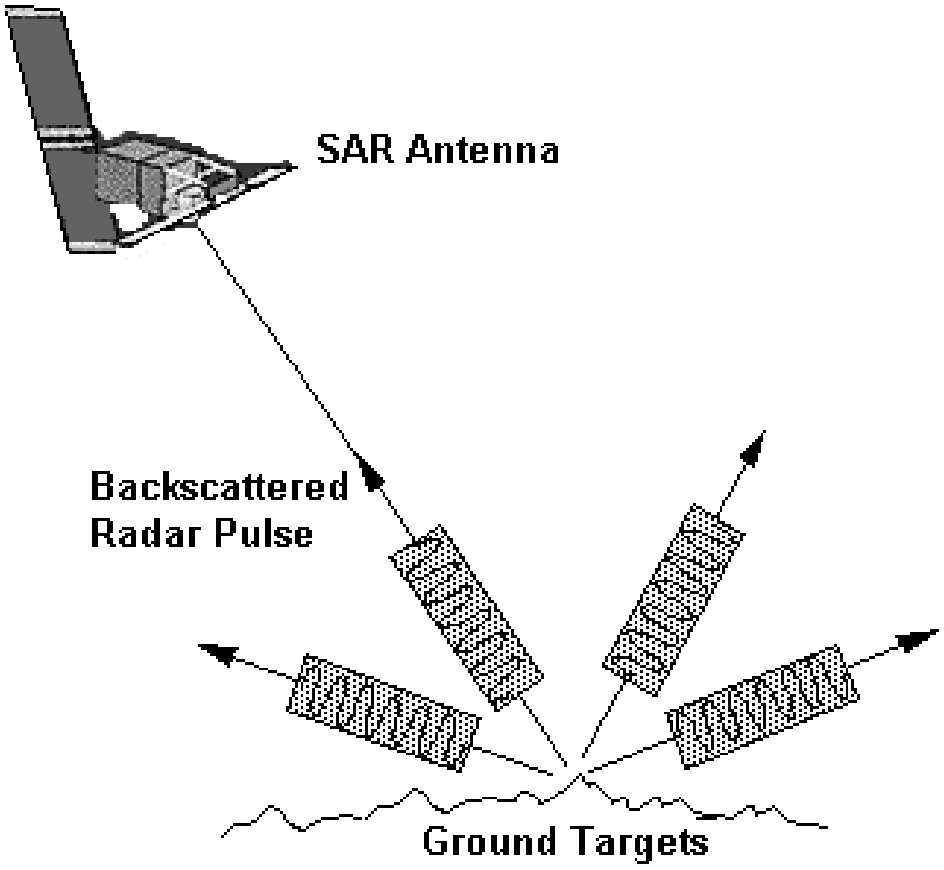

Figure 3: Scattered and reflected signals, `http://www.crisp.nus.edu.sg/~research/`

2.3.2 SAR specifics

What sets Synthetic Aperture Radar apart from regular radars is that a SAR is constantly moving at a high speed rather than remain stationary. This allows SAR radars to produce fine spatial resolution images while having a small physical antenna aperture. A SAR continuously transmits radio waves to target area and gathers the reflected signals as it moves over land, figure 3 depicts the relection and scattering of those signals. The radar movement means that several recordings of the same area from multiple antenna

locations can be combined - this forms the synthetic antenna aperture. This also allows SARs to also create 3d representations of an area. Since for each unit of land covered, the radar collects multiple differing signals, the pixel (or voxel - 3d pixel) values are usually represented in terms of probability (density) of a reflective surface being at a certain location. [Oli89] Since multiple differing signals cannot usually be coherently collected, raw SAR images contain a lot of random interference effects called "coherence speckle" which from an image-processing standpoint is considered random noise. For raw SAR images, information regarding the radio wave's waveform is also preserved, most importantly - phase. This means that in addition to typical image processing algorithms, signal processing methods where processing is done on radio waves can also be used. SAR images can potentially have a very fine spatial resolution, airborne radar systems can provide resolutions to about 10 cm and less. Satellite images though have a worse spatial resolution and the images provided by the Sentinel 1 satellite, which are used in this work have a spatial resolution between 5 by 5 and 20 by 40 meters. [Age]

2.3.3 Common uses

Since SAR images are produced by using radio waves, they are not affected by clouds and work all the same in day and night meaning they can be reliably used for various land and sea monitoring tasks. Indeed, one of the main uses of SAR imagery is to track and study polar ice movements and deforestation and farm-land usage. Since precipitation and other various weather phenomena affect the ground they are over, the most obvious example being flooding, the images can also be used to better coordinate natural disasters? relief efforts. Other areas where SAR image processing is becoming more used is tracking of smaller-scale events, shipping routes and urban development for example.

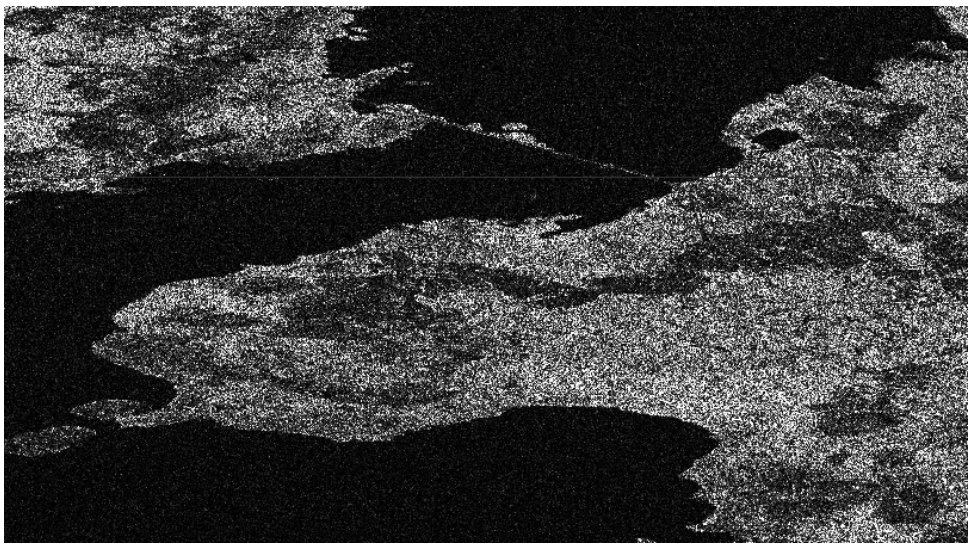

Figure 4: Bridge between Muhumaa and Saaremaa `https://scihub.esa.int/`

2.3.4 Image representation

Since the Sentinel 1 radar employs a technique called TOPSAR where the produced images are two-dimensional, this thesis does not examine three-dimensional SAR image processing possibilities. The images used for this thesis are of two types, first type consists

of compressed and unfocused SAR raw data, called Sentinel 1 level-0 products, figure 4 depicts a typical level-0 product. Images of the second type have been preprocessed and calibrated by ESA and are therefore more useful for higher-level processing, these are called Sentinel 1 level-1 products. Figure 5 depicts a typical level-1 product, note that the sideways-look present in product-0 images is gone. Level-1 products however do not contain phase data, meaning only conventional image processing can be conducted. For Level-0 products, each pixel contains a 32-bit integer value which can be operated on for the purposes of most algorithms. Level-1 products can essentially be considered as just grayscale tiff images.

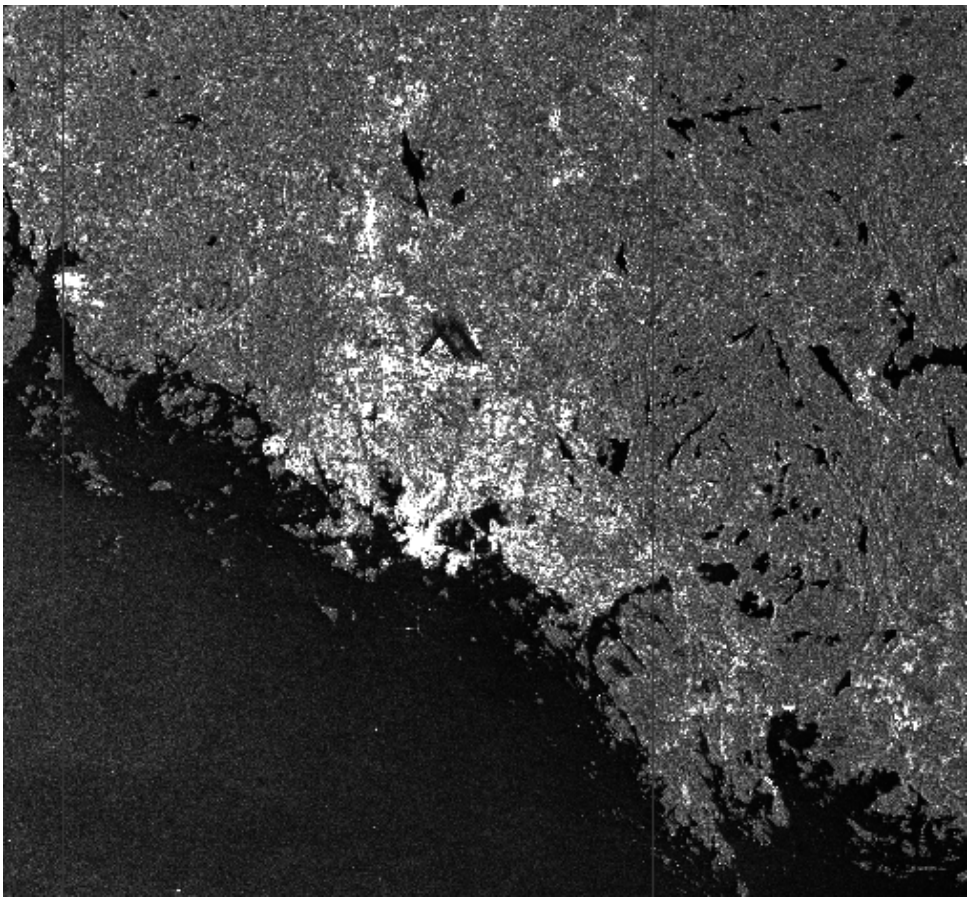

Figure 5: Helsinki

3 Approach

3.1 Theoretical

To best tackle the issues of adapting SAR image processing algorithms to parallel computing environments and to ensure that the proposed solutions would be as general as possible, a certain classification of potential algorithms is needed. The classification has to provide an answer to the question, which SAR data processing algorithms can be efficiently parallelized and needs to consider the architecture and properties of both the distributed storage system (HDFS) and the computing framework (Spark) used, as well as the characteristics and use cases of SAR images themselves.

3.1.1 Considerations

The inherent nature of parallel processing dictates that to gain the biggest increase in processing speed, the processing task needs to be dividable into several independent computing tasks. In the context of Spark, this means that the code running on parallel executors needs to be executable at the same time and one executor's processing should not depend on some other's progress. Indeed, in Spark, operations on RDDs are defined so that the user-defined transformation or action operates on some chunk of data with the information regarding other chunks being not available by default and the order of execution unspecified, meaning programmers are forced to define their functions so that the execution of code on chunks can potentially be done in any order. This means that RDD operations are inherently parallel. To gain the largest increase in processing speed, it is then necessary to maximize the number of potential parallel executors. For this work specifically, that is to maximize the number of objects in an RDD representing a large image. This can be done by splitting the original image into a multitude of smaller ones.

However, if done so, during the execution of a single transformation, distant pixel values are unknown. This in turn means, that algorithms that require access to the whole image, such as non-local means denoising with window size as the whole image, where to set a particular pixel value, the mean of all pixels in an image is taken and weighted by how similar those pixels are to the target pixel, need multiple passes on the data. Since doing multiple passes on the data is fast in Spark due to intermediate results not needing to be written to disk, the algorithms' classification should consider what processing needs to be done inside a single executor.[3]

Satellite SAR images do generally not have a very fine resolution, that is, a single pixel represents an area in tens of meters. This combined with the fact that new images are quickly available, means that SAR images are most suitable for detecting large scale changes on the Earth's surface. Two groups of typical algorithms ran on SAR data, are thus change and edge detection algorithms. Since a typical characteristic of SAR images is speckle noise, another essential group of algorithms is denoising algorithms. Because many denoising and edge detection algorithms operate on or inspect a neighbourhood of pixels the algorithms' classification should cover sliding window approaches. The requirement for change detection algorithms, multi-temporal images, means that the classification should also cover algorithms with multiple input images.

Since the goal of the classification is to help compare algorithms' potential in terms of suitability for parallelization, it is important to note what is meant by parallel speedup. For this work, parallel speedup refers to how close the parallel algorithm's implementation can get to the theoretical maximum speedup defined by Amdahl's law. For problems that can be easily separated into parallel tasks[4] , this is close to the number of parallel executors used.

[3]This is specified in the user-defined RDD operation function.

[4]Called embarrasingly parallel problems

3.1.2 Classification

The proposed classification based on previously described considerations therefore makes two distinctions. The first is based on the number of input images take algorithm takes as arguments:

- Single image input
- Multiple image input

The second is based on the size of the necessary visible area of a single step of the algorithm:

- Pixel-based
- Window-based

If both distinctions are considered, the SAR processing algorithms to be potentially parallelized are divided into four broad classes:

1. Single image, pixel-based
2. Single image, window-based
3. Multiple image, pixel-based
4. Multiple image, window-based

The first class is the simplest one and includes algorithms that apply various single-pixel filters or otherwise increase/decrease the pixel values. The second class covers typical neighbourhood filters with many denoising algorithms falling into this category. The third class covers change detection algorithms, where the images' pixel values are directly compared or otherwise used, to train some model for example. The fourth class covers various object search and detection algorithms. It is important to note, that while many image processing algorithms fall into one of the described classes in their entirety, many do not. However, the majority of algorithms of interest can at least be reduced to only contain sub-algorithms from the four mentioned classes.

3.2 Practical

With the algorithms' classification in place, it is necessary to study, which of these can be parallelized using HDFS and Spark, what is the most efficient way to do so, and how do algorithms from the four classes compare to one another in terms of processing speed gain.

3.2.1 Single image pixel-based algorithms

Single image, pixel-based algorithms can easily be parallelized due to their inherent nature. Since operating on a single pixel in no way affects other pixels, the number of parallel executors can potentially be as large as the total number of pixels in an image. From a practical standpoint, the number of parallel executors obviously cannot be as large and for HDFS and Spark specifically, it would be reasonable to have a single node process at least roughly a block size of data. So for a 1200MB image with HDFS block size 128MB, having 10 parallel executors would likely yield a good processing time win. Adapting these algorithms to parallel environments can be done with no change to the program's logic.

3.2.2 Single image window-based algorithms

Single image, window-based algorithms require that for each pixel its surrounding area with arbitrary size, be visible. Parallel processing for these types of algorithms can be thought of as scanning multiple areas of the same image at the same time. Since different windows can have overlap, cases may arise where several executors need to access the same pixel at the same time. In Spark, situations like these are handled by the execution framework by replicating data to the nodes that need them. Replicating data during execution is also needed when the window spans the HDFS block boundary, that is, when processing pixels near the borders of a sub-image in a particular block. Part of pixels in the window reside in one block and part in another one and thus a portion of the data needs to be fetched across the network from the correct node. The larger the window size and number of parallel executors used, the more data is needed to be fetched. Since compared to processing speed, network traffic is slow, this incurs a significant time penalty on the whole task.

To parallelize this group of algorithms most efficiently it is therefore necessary to minimize the amount of data needing to be exchanged between nodes. Maximal efficiency can be achieved if no additional data is exchanged, that is, if every executor node contains all the data it needs for computations from the start. The notion of assigning executors to data already on the nodes and needing no additional data transfers is known as data locality and is one of the main principles of processing in Hadoop and Spark.

In the case of SAR data processing, maximal data locality for this class of algorithms can be attained if the sub-images on nodes are stored with certain overlap so that the logical split boundary and the actual HDFS split boundary are different. This means that pixels that are outside the logical boundary but inside the actual block boundary are explicitly stored on multiple nodes. [5] In 6 the dotted lines repsent the logical boundaries while the green area represents an example sub-image actually stored on the node. This way of storing images obviously carries with itself additional storage overhead depending on the overlap size and the number of splits the original image is divided into. In HDFS, to keep storage overhead low while still maintaining data locality, the number of splits should be chosen so that the actual split size would be roughly the same as the HDFS block size.

The parallel implementations of this class of algorithms need to be aware that the input images contain overlaps and are therefore larger than the area the algorithm needs to logically process. The simplest approach would be to have the algorithm code running on parallel executors to require two additional parameters, one for overlap size in the horizontal axis and one for vertical axis. The only modifications to the non-parallel algorithm would be to use the overlap parameters to specify the area of the image where changes might occur and to use the whole image on the node only for the search window. In figure 7 the area of pixels for which values might potentially change is in blue and the overlap is in white. Dotted square represents the starting position of the search window.

3.2.3 Multiple image pixel-based algorithms

Multiple image, pixel-based algorithms' parallel executors need to have access to several input images at once. The base case for these algorithms is for an executor to need access to two images. The non-parallel implementations can be thought of as looking at and

[5] Even if the HDFS replication factor is 0

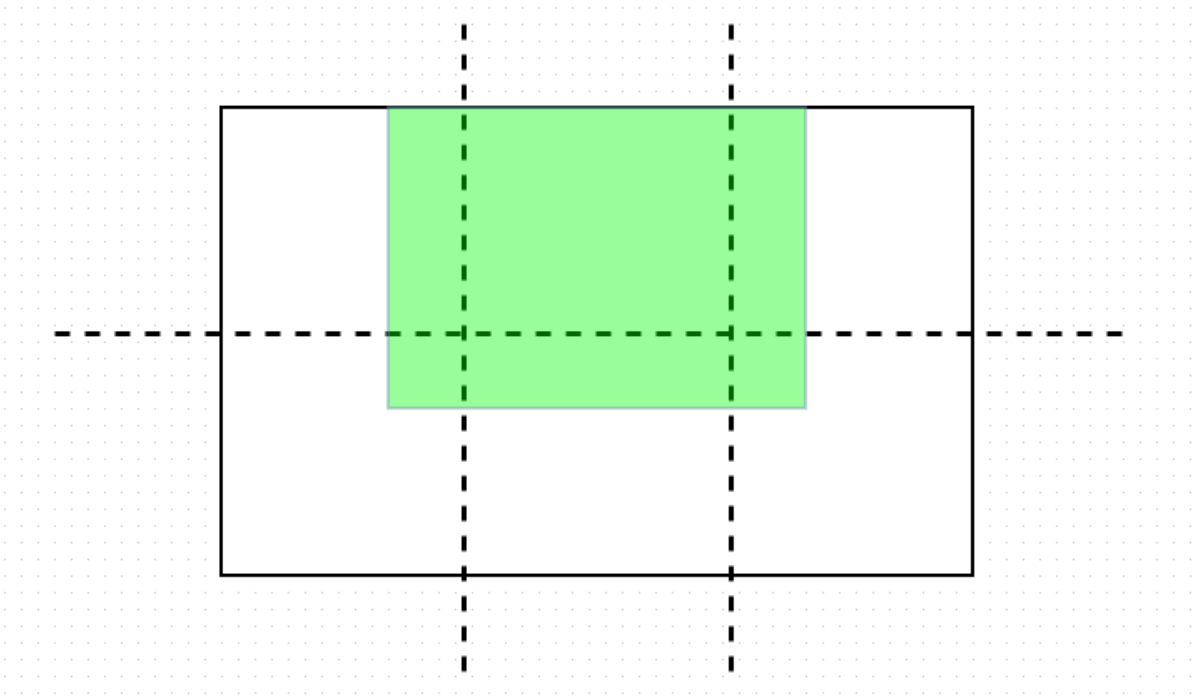

Figure 6: Large image

processing each pixel of both images simultaneously. Any order of going through the pixels is valid so long as the pixel's coordinates are same for each image. This means that if an executor has access to both images, parallelising the algorithm can be done in the same fashion as for single image pixel-based algorithms, that is, with no change in the non-parallel algorithm's logic.

Since the image splits that correspond to each other can potentially be on any nodes, one of them [6] likely needs to be fetched from some other node during execution. This means that data locality is lost and the parallel speedup of the whole task is smaller than potentially achievable, however it can still be considerable and therefore worthwhile to study. To preserve data locality for this class of algorithms, it is necessary to change the underlying file system's block placement policy, that is, how different blocks of data are placed on different nodes on the cluster. Although this is also possible in HDFS, it requires a more low level configuration of HDFS and therefore outside the scope of this work. Algorithms that require access to more than two input images are different only in the way that the number of images potentially needed to be fetched is larger. If the algorithm operates on n images, each divided into blocks in same fashion, the number of splits needed to be fetched for each executor can be as large as n - 1. This means that unless the distributed file system's block placement policy has been overridden so that splits for the same area of each image are placed on the same node, the amount of data transferred over network during execution grows large and the speedup of the whole task diminishes rapidly as the number of input images grows.

[6] For the case with two input images.

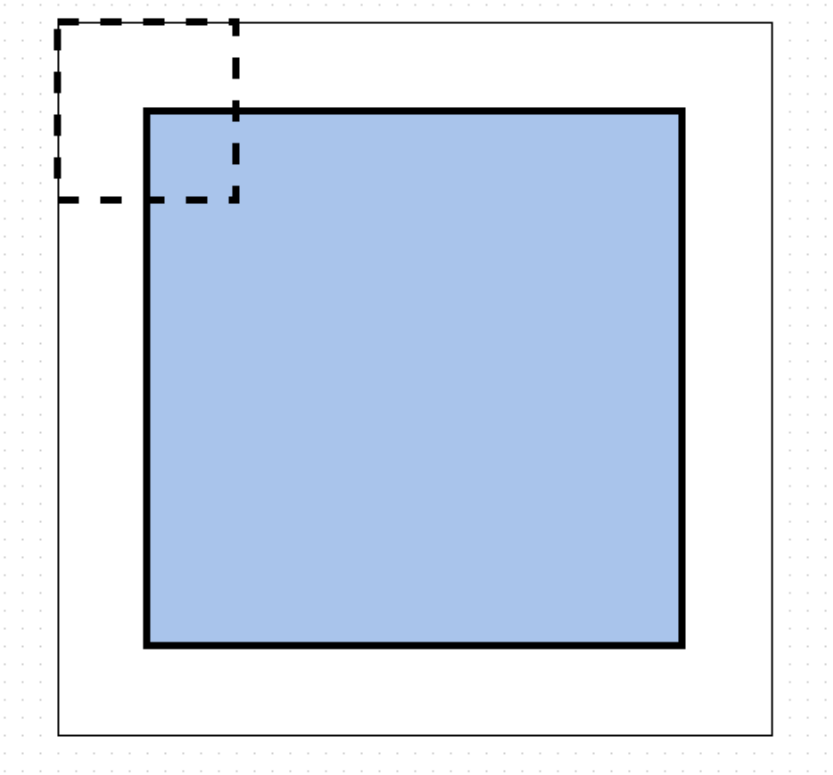

Figure 7: Single image split

3.2.4 Multiple image window-based algorithms

Multiple image, window-based algorithms suffer from the same parallelisation issues as multiple image pixel-based algorithms, that is, if the number of images the algorithm needs access to grows large, the amount of data needed to be exchanged between nodes imposes a large penalty. However, par a few concrete cases, algorithms belonging to this class are not particularly common in image processing. The most widely used group of algorithms in this class are various object detection and search algorithms. In those cases, the algorithm takes two images as input with one, the object representation, being considerably smaller. In those cases the smaller image can be broadcasted to all the nodes with no significant time penalty. The larger image would be split in the same fashion as for single image window-based class of algorithms. The overlap would need to be with the dimensions of the smaller image minus one pixel in both dimensions. If the images are split correctly and the object representation is broadcast to all nodes, each parallel executor can run the non-parallel algorithm with no change in program's logic.

3.3 Implementations

For the specific example algorithms to implement, an algorithm was chosen from both the second (single image, window-based) and third (multiple image, pixel-based) classes. The first class (Multiple image, window-based) was left out of implementations since it can be described in terms of the second class with special case of overlap being zero. The fourth class of algorithms was left out since currently the main uses of SAR images do not include many algorithms belonging to this class. However, they are certainly intereseting research topics and should be considered for future work. It should be noted that the

language chosen to implement the algorithms is Java, but Spark also supports Scala and Python and implementations in those languages should be similar.

3.3.1 Interacting with the file system

The first practical problem that needs to be solved is splitting the images, storing them in the HDFS and making sure they can be properly read in the application code. Since SAR images are in tiff format, they cannot be read using the standard Java library. The library chosen for this task is called Java Advanced Imaging(JAI) library[7] and it is suitable since it provides a simple high-level api for manipulating images, making it easier to focus on the algorithms' implementations while developing. Since JAI extends the standard Java library it is reliable and integrates easily into Java programs.

The splitting of an image into multiple splits as described in the previous section is done as a preprocessing task. The splits are then written to HDFS with each split being in a separate HDFS block. Since Spark is primarily used with textual data, some custom additions are needed to work with tiff images, mainly for the writing of images to HDFS from the executors. The reading of images is done by using JavaSparkContext.binaryFiles method which returns an RDD of type <String, PortableDataStream> with the string representing the path to a particular binary file and the stream containing its contents. By applying a map transformation to this RDD, the data stream can be directed to an ImageIO.read method which, if the necessary JAI libraries are present, returns a BufferedImage object representing a concrete tiff image split which are then manipulated depending on the needs of the algorithm implemented. For writing the BufferedImage objects back to HDFS, custom implementations of certain Hadoop classes need to be provided. Those classes are Writable, RecordWriter and OutputFormat. For this work, the implementations just delegate various read and write methods to the corresponding ImageIO methods.[8] Figure 8 depicts a SAR image before and after applying median filter with 1 pixel neighbourhood. Note that denoising does not necessarily mean that the image is more clear to humans.

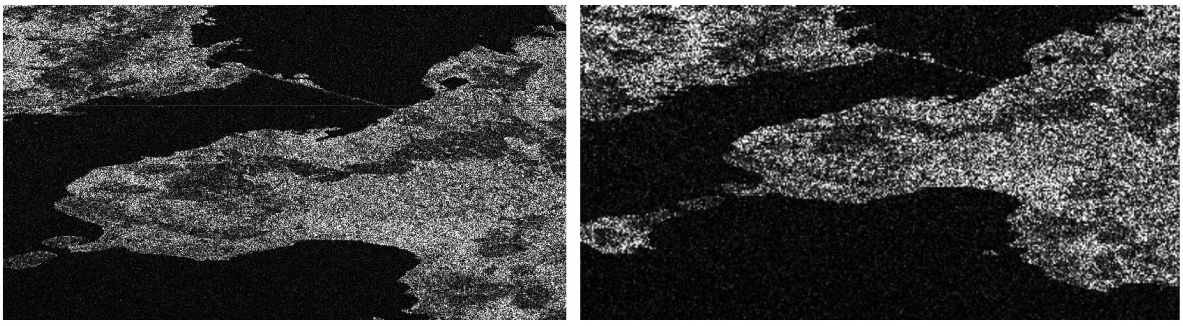

Figure 8: Before and after median filter

[7] `http://www.oracle.com/technetwork/articles/javaee/jai-142803.html`

[8] Refer to the repository in appendix to view the implementations.

3.3.2 Median filter

The algorithm chosen to represent single image window-based class of algorithms is a simple neighbourhood filter algorithm called the Median filter. The Median filter algorithm is as follows:

- Iterate over each pixel of the image
- For a single pixel, inspect its some neighbourhood
- Replace the original pixel value with the mean of the pixel values in the neighbourhood

For implementation purposes the neighbourhood for each pixel is defined by two parameters: x and y. The neighbourhood is a rectangle with dimensions x + 1 by y + 1 and centered on the pixel being processed. To retain data locality it is important that neither the x or y parameters are no larger than the image overlaps in the corresponding axes. The classification previously proposed states that for this class of algorithms, the only necessary change in logic from the non-parallel implementation is that only the non-overlapping area be considered for changing the pixel values. So in Spark, for an RDD of images, the algorithm is specified by a map function which acts as the non-parallel version of the algorithm but instead of iterating over each pixel of the image, only the area that is not part of overlap is iterated over. Since changing the pixel values during processing should not affect other pixels that have this pixel in their neighbourhood, the image needs to be copied first, meaning that memory usage for each executor is at least twice the size of the split.

The imagesplits that have gone through denoising can then be used as input for other algorithms. If it is necessary to inspect the original image after denoising, all imagesplits need to be downloaded and and then joined together as a post-processing task. Note that the location of each split on the image must be preserved throughout denoising to succesfully join the splits back together later, this can be done by having the split filenames contain the necessary information

3.3.3 RGB composite change detection

The algorithm chosen to represent multiple image, pixel-based algorithms does not have a specific name but the technique is common. [RAAKR05] Several single-band images are composed to a multi-band image. For SAR images this can be done by considering the pixel values as single grayscale color values. The algorithm can then be described as follows:

- Iterate over the pixels of 2 separate images simultaneously
- For each pixel pair assign first image's grayscale pixel value as the red band of the composite image and the second image's corresponding pixel value as the green band
- Assign the average of both pixel values as the blue band

Parallel executors need to have access to the correct images that make up the composite. In Spark, this can be achieved by using the join operator (e.g. RDD1.join(RDD2)), this creates an RDD of type <K,<V,V» from two RDDs of type <K,V>. Values with

same keys are joined together. [sf] This means that the easiest way to implement this algorithm for multitemporal SAR images in Spark is to have two input folders, each containing images of the same locations on different times with images of the same area having the same name. According to the classification, as long as the executor has access to both images, the non-parallel implementation logic of the algorithm can be used for the parallel executors as well. This is indeed the case, when calling map function on the RDD where values contain two images, the user-defined function can be ported from non-parallel environments and the correct output is produced for each image in the RDD. Since the composite image is a typical rgb image, it is reasonable to save it as a png instead of tiff to save storage. For this, custom implementations of Writable, RecordWriter and OutputFormat were specified. They are nearly identical to the tiff implementations but the ImageIO methods called refer to png images instead.

Figure 9 depicts the output for RGB composite change detection algorithm that was run on two images from December 2014 with 3 weeks time between them. The earlier image was assigned to red band and the later to green band. Since pixels that did not change in value over time have all red green and blue values the same, they are in grayscale. Pixels that had stronger magnitude in the first image are shifted to red values and pixels that had stronger magnitude in the second, to green values. This means that in the context of this image red-ish areas represent the loss of snow and ice and green areas the accumulation of additional ice and snow.

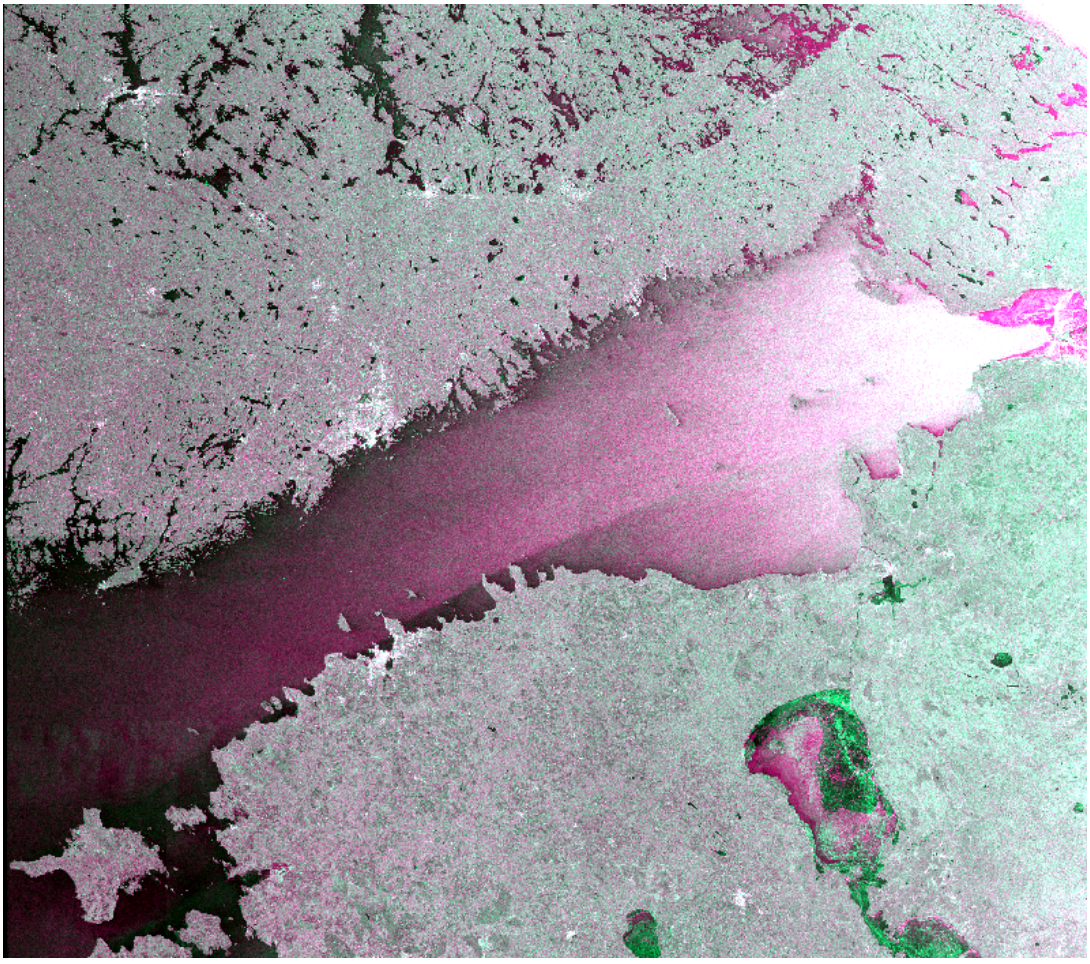

Figure 9: RGB composite change detection

4 Results on cluster

All tests were ran on a 4-node cluster with each node having 2 processor cores, 4GB of memory and 20GB of SSD storage.

According to the theoretical approach described in the previous chapter, the implemented median filter denoising algorithm's parallel speedup should be reasonably close to the theoretical maximum since the algorithm itself has no strictly serial part and because data locality is maintained, neither HDFS nor Spark impose a considerable overhead.

This is supported by the results of running the algorithm both on the cluster and locally. Running median filter on one 1.3GB image split into 16 splits with 1 pixel overlap took 374 seconds locally and exactly 100 seconds on the 4-node cluster. The actual parallel speedup is 3.74 while the theoretical maximum is 4. With double the data, 4-node cluster ran the algorithm in 213 seconds, assuming non-parallel implementation scales linearly, the speedup is 3.51 With triple the data the cluster's time was 298 seconds meaning 3.76 speedup. Since runs with more data are generally more representative, it is reasonable to think that with additional data the speedup would stay roughly the same or increase. Figure 10 shows how input data size and parallel execution time are correlated for the Median filter algorithm and the theoretical best case scenario.

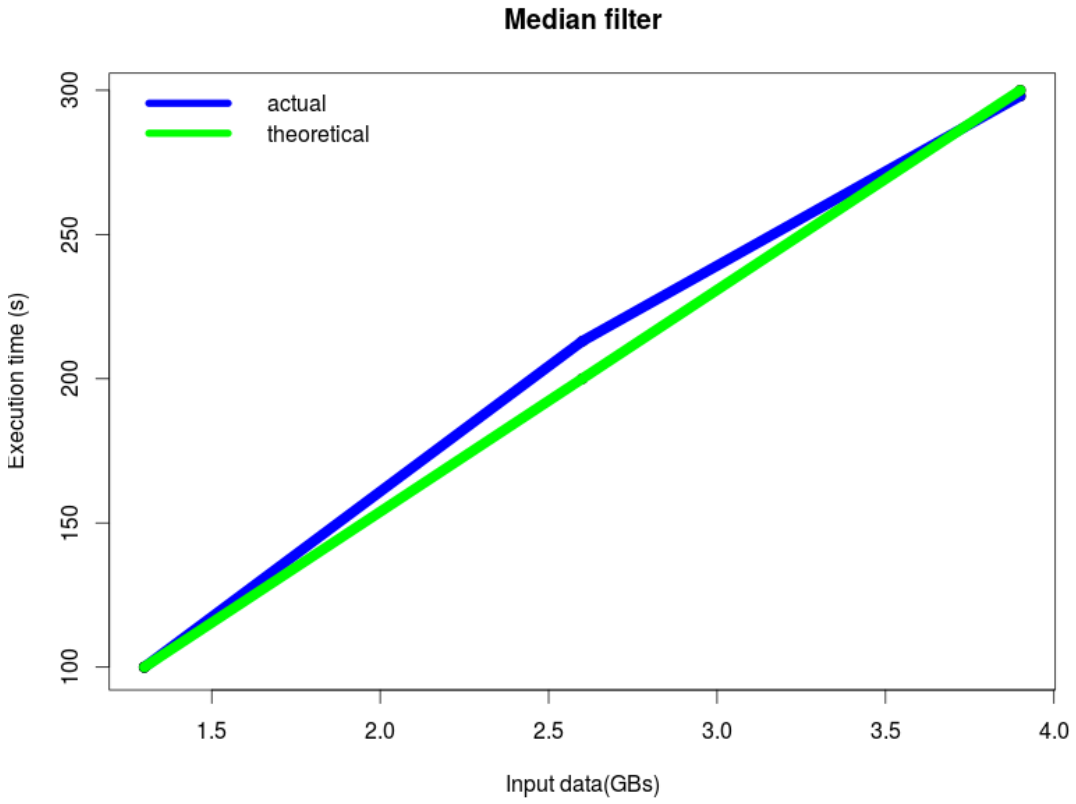

Figure 10: Median filter data and execution time dependence

For the second algorithm, the classification states that the speedup should be smaller, especially when the number of images gets large. Composing two 200MB images on local machine took 70 seconds and on cluster 19 seconds, this means a 3.7 parallel speedup which indicates that transferring 200MB over the nodes is too small amount of data to have cause a noticeable data transfer overhead. With double the data, the algorithm ran

for 66 seconds meaning the speedup is 2.15 With quadruple the data, the algorithm ran for 199 seconds meaning 1.42 speedup. Figure 11 shows how input data size and parallel execution time are correlated for the change detection algorithm and the theoretical best case scenario.

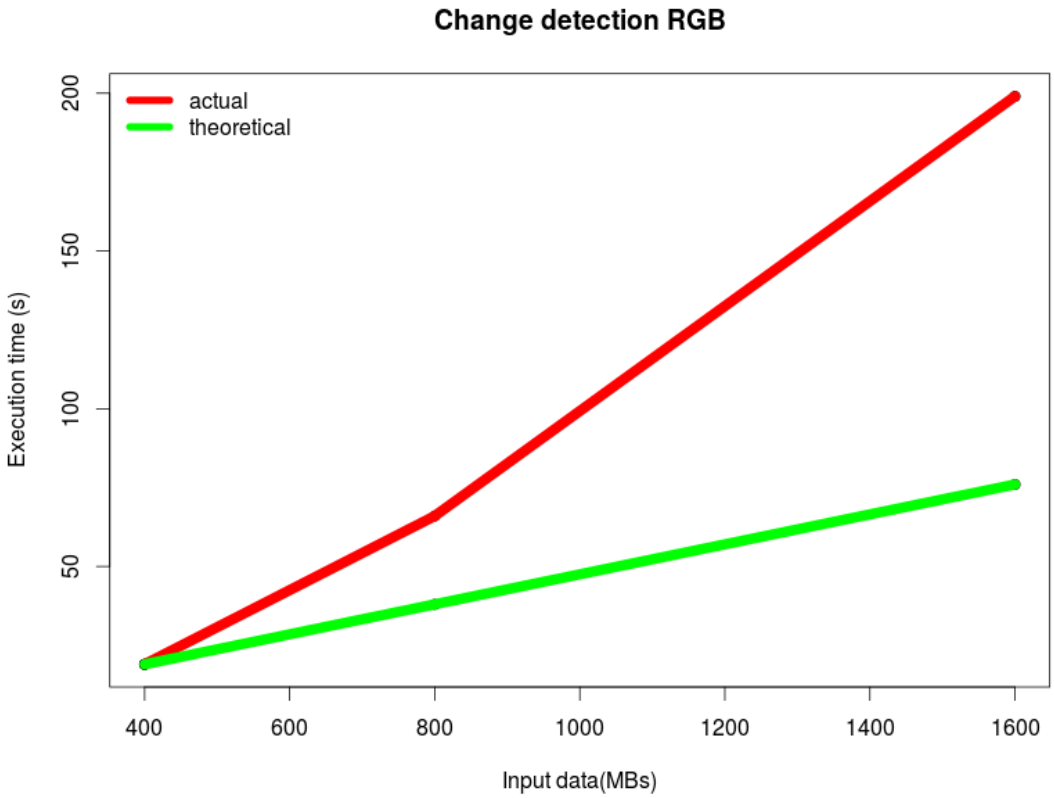

Figure 11: RGB change detection input data size and execution time dependence

5 Conclusions

5.1 Conclusion

The present work proposed a classification of synthetic aperture radar image processing algorithms in which the feasibility of parallelizing concrete algorithms can be determined. Properties of those classes were studied and proved in practice. Example algorithms were implemented and their parallel implementations provided. The example algorithms serve as a good starting point in developing more complex SAR data processing applications and with new monitoring satellites constantly being launched the area of satellite image processing has nearly endless amount of potential applications to be created. A lot of further research is however needed to determine how to best take advantage of the enormous potential large-scale satellite image processing has.

5.2 Proposals

5.2.1 Further research

Although the proposed algorithms' classification can be used to group many SAR image processing algorithms into one of the four classes, another group of algorithms is where the algorithm as a whole does not belong to any class but can be represented as a number of sub-algorithms that each belong to one of the four classes. This means that they can be moved to a parallel environment with relative simplicity and reasonable gain in runtime speed. In many cases reduction to sub-algorithms in the correct form is trivial, however for more complex algorithms it is not. Further work is needed to determine if those complex algorithms can be reduced to the correct form and if it is possible to do so in a reasonable number of steps.

As described in the theoretical approach and evident from the results of running the algorithms on cluster, algorithms that process multiple images simultaneously lose a significant portion of potential speedup when using straightforward implementations due to the amount of data that needs to be transferred over network. This issue can be solved by making sure that the blocks of data executors need access to are placed on the same data nodes which can be accomplished by overriding the distributed file system's block placement policy. It is not clear if this can be done for the general case or if different algorithms require differing block placement policies. It should also be studied if data locality can be preserved in this way for any number of input images.

Another area where further research is needed, is parallel processing of SAR waveform data. Since the splitting, reading and writing of SAR images in this work was all done with no loss of pixel data, further work could use similar techniques for the storage of SAR images.

5.2.2 Applications

Since ESA continuously provides the latest satellite SAR images for anyone to freely download and in this work it was show that SAR data processing can be feasibly done in Spark, it is possible to build a continuous monitoring tool using Spark Streaming. This application could detect notable changes in the images as soon as the images become available meaning problems such as illegal deforestation could be detected more rapidly.

Although algorithms for this work were implemented in Java, Spark additionally supports Python and Scala which both also have interactive shells for Spark. This means an interactive and fast data processing/analysis tool can be developed using either Python or Scala Spark shell. This would make it easier to perform exploratory data analysis by trying many different processing algorithms in quick succession. The tool would be especially useful if it could include some type of visual feedback. Further work is needed however to determine if this is possible in distributed manner.

References

[Age] European Space Agency. Sentinel 1 sar overview. [confirmed on 14.05.2015] `https://sentinel.esa.int/web/sentinel/user-guides/sentinel-1-sar/overview`.

[Oli89] CJ Oliver. Synthetic-aperture radar imaging. *Journal of Physics D: Applied Physics*, 22(7):871, 1989.

[RAAKR05] Richard J Radke, Srinivas Andra, Omar Al-Kofahi, and Badrinath Roysam. Image change detection algorithms: a systematic survey. *Image Processing, IEEE Transactions on*, 14(3):294–307, 2005.

[sf] Apache software foundation. Spark programming guide. [confirmed on 14.05.2015] `http://spark.apache.org/docs/latest/programming-guide.html`.

[ZCD+12a] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX; login*, 37(4):45–51, 2012.

[ZCD+12b] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCD+12c] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

6 Appendices

6.1 Appendix A: Code repository

All implementations are stored in a git repository at
`https://github.com/ossu54/BSc-Thesis`

6.2 Appendix B: License

Non-exclusive licence to reproduce thesis and make thesis public

I, Oskar Hint (date of birth: 26th of October 1992),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Efficient parallel algorithms for synthetic aperture radar data processing using large-scale distributed frameworks

supervised by Pelle Jakovits

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 14.05.2015