DISSERTATIONES
MATHEMATICAE
UNIVERSITATIS
TARTUENSIS
110

PELLE JAKOVITS

Adapting scientific computing algorithms to distributed computing frameworks

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

110

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
110

PELLE JAKOVITS

Adapting scientific computing algorithms to distributed computing frameworks

1632 UNIVERSITY OF TARTU
Press

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia

Dissertation is accepted for the commencement of the degree of Doctor of Philosophy (PhD) on January 25, 2017 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor:	PhD. Satish Narayana Srirama University of Tartu Tartu, Estonia
Opponents:	PhD. Paolo Trunfio University of Calabria Rende, Italy
	PhD. Vladimir Vlassov Royal Institute of Technology (KTH) Stockholm, Sweden

The public defense will take place on March 14, 2017 at 15:00 in J. Liivi 2-405.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.

European Union
European Social Fund

Investing in your future

ISSN 1024-4212
ISBN 978-9949-77-347-3 (print)
ISBN 978-9949-77-348-0 (pdf)

University of Tartu
http://www.tyk.ee

ABSTRACT

Scientific computing applies computational methods to solve problems in genetics, biology, material science, chemistry etc., where complex real life processes need to be modeled and simulated or where a large amount of data needs to be analyzed. It is strongly associated with parallel programming and high-performance computing (HPC) as it typically requires utilization of a large amount of computer resources from local clusters and grids to supercomputers.

Public clouds can provide these resources on-demand and in real–time but they are often built on commodity hardware and it's not simple to design applications that can efficiently utilize their resources *en masse* and in fault tolerant manner. Frameworks based on distributed computing models such as MapReduce can significantly simplify this work by providing near automatic parallelization and fault recovery. Our first research task was to investigate the suitability of Hadoop MapReduce for more complex scientific computing algorithms and to identify what algorithm characteristics affect the parallel efficiency of the results.

Hadoop MapReduce could easily handle more simple, embarrassingly parallel algorithms, such as trial division or Monte Carlo methods. However, it had serious issues with more complex and especially iterative algorithms, such as the conjugate gradient method. To be able to exploit MapReduce advantages (such as near automatic parallelization and fault recovery) for more complex algorithms, we proposed and investigated three different approaches.

The first approach was to reduce the number of iterations by restructuring the algorithms or using alternative methods that might be less efficient, but would suit the MapReduce model better. The second approach was evaluating alternative MapReduce frameworks (such as Twister, HaLoop or Spark) that are specifically designed for iterative algorithms and analyzing whether they provide the same aforementioned advantages as Hadoop.

The third approach was investigating frameworks from an alternative, Bulk Synchronous Parallel distributed computing model.

Our conclusions were that the first approach would require domain specific expert knowledge of the involved methods and Hadoop MapReduce framework. The alternative distributed computing frameworks investigated in the second and third approaches often produced better results than Hadoop MapReduce, but there is no single framework that suits all different types of scientific computing algorithms.

The efficiency of the result can depend on algorithm characteristics such as the size of data stored in memory or required communication patterns and choosing the best suited distributed distributed computing frameworks can be a very complicated task. One approach would be to choose the most likely framework candidates, implement the algorithm on each of them and perform benchmarking. But this would require relatively large amount of programming and debugging effort and require studying each of the chosen distributed computing frameworks in detail.

This process could be complicated further by the possibility that different distributed computing frameworks might be more suitable for different algorithms part of the same scientific computing application. Emerging technologies (such as Hadoop Yarn or Aneka) can alleviate this issue by enabling on–demand switching between different distributed computing frameworks for different tasks. However, the actual choice between different distributed computing models and their implementations is still left up to the user, which can be a very difficult task when the number of available frameworks is high.

We created a Dynamic Algorithm Modeling Application (DAMA) for simulating the parallel structure of scientific computing algorithms and defined a methodology which uses DAMA to identify the most suitable distributed computing framework for a given scientific computing algorithm.

DAMA is implemented on a number of distributed computing frameworks and can be used to estimate the performance of modeled algorithms by using it as a benchmark in real distributed computing environments. The suitability of distributed computing frameworks and parallel programming libraries can be evaluated without having to implement the given algorithm to any of them and all the programming and debugging tasks can be postponed until the final parallel programming solution has been chosen.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor Satish Srirama for his guidance and support over the years. I would also like to thank all the previous and current members of the Mobile & Cloud Lab and Distributed Systems group of University of Tartu for all the interesting discussions and support.

The work in this thesis was supported in parts by European Social Fund's Doctoral Studies and Internationalization Program DoRa, Archimedes Foundation, Tiger University Program of the Estonian Information Technology Foundation, Estonian Doctoral School of Information and Communication Technology (IKTDK), Centre of Excellence in Computer Science (EXCS), AWS in Education Grant, Estonian IT Academy, Estonian Science Foundation grants ETF9287 & PUT360 and Target Funding theme grant SF0180008s12.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application programming interface
BLCR	Berkeley lab checkpoint/restart
BSP	Bulk synchronous parallel
CG	Conjugate gradient
CLARA	Clustering large applications
DAG	Directed acyclic graph
DAMA	Dynamic algorithm modeling application
DRMA	Direct remote memory access
FFT	Fast Fourier transform
FSM	Finite state machine
GPS	Graph processing system
HDFS	Hadoop distributed file system
HPC	High performance computing
IaaS	Infrastructure as a service
IF	Integer factorization
JNI	Java native interface
JVM	Java virtual machine
LDA	Latent Dirichlet allocation
MPI	Message passing interface
MR	MapReduce
NIO	Java new I/O
PaaS	Platform as a service
PAM	Partitioning around medoids
PLSI	Probabilistic latent semantic indexing
PUB	Paderborn university BSP
RDD	Resilient distributed datasets
SaaS	Software as a service
SLAE	Systems of linear algebraic equations

SPMD	Single program, multiple data
SVD	Singular value decomposition
SQL	Structured query language
YARN	Yet another resource negotiator

CHAPTER 1

INTRODUCTION

Computer resources that are required for large scale scientific computing applications have historically been provided by supercomputers and computer grids, but cloud computing [7] has risen as a competitor for these types of distributed computing resources. Public cloud providers following the Infrastructure as a Service (IaaS) model provide virtually unlimited computing resources on demand and nearly in real-time.

Public clouds also provide a number of additional advantages to scientific computing applications. Users do not have to wait in resource queues like in grids and are not bound by the limits of their local resources. Clouds provide users full access to virtual machines instead of giving limited access to the underlying machine. Users are able to choose the operating system, its configuration, software packages and libraries, allowing them to fully configure the environment their applications run in.

When users are not interested in configuring the hardware and instead want to directly deploy their applications in a pre-configured system (similar to grid job submission systems), they can use the Platform as a Service (PaaS) model of the cloud instead. In PaaS the cloud service provider sets up and manages the platform for the users and users can simply deploy and execute their applications or experiments there. Furthermore, it is also possible to provide existing scientific computing applications using Software as a Service (SaaS) cloud computing model, where applications are deployed in cloud platforms and users can use them directly without any installation or configuration required.

These advantages can be very useful for scientists who do not have consistent access to computing resources or whose peak computing resource needs often greatly exceed their available resources. However, clouds also have several disadvantages. In comparison to many collaborative research grids, public clouds use the utility pricing model and the users have to pay for the used resources. It has also been well documented [8, 9, 10] that the virtualization layer, which is used to simplify and speed up cloud resource provisioning, also increases the computation and communication latencies.

The first disadvantage can be a serious issue, but by providing easier access to computing resources, clouds enable scientists to run larger scientific computing experiments and help them keep up with the ever increasing resource demands of the new advances in sciences. That is, only as long as the users can budget the costs of the experiments. The second disadvantage can't be overcome easily, but the situation has improved steadily with the advances in virtualization technologies and with cloud provides like Amazon giving access to dedicated high performance computing instances that have lower virtualization overheads.

Additionally, having more and more computing resources available also means that the applications must be able to properly utilize these resources. Typical approach is to create distributed applications using the de-facto standard Message Passing Interface (MPI) [11] for synchronizing the computations across a number of machines. However, designing, writing and debugging MPI applications is not a simple task. When using MPI parallel programming library, programmers have full control over how the distributed program is executed. They have to explicitly specify how the input data is partitioned, how the data and processes are synchronized, how to avoid communication deadlocks, etc. Furthermore, these tasks become more complicated as the number of concurrent processes increases.

Another important factor when running resource hungry and potentially very long running scientific applications is fault tolerance. When running applications on hardware consisting of thousands or even hundreds of thousands of processor cores, the likelihood of failures is significant.

MPI applications do not have good means to recover from machine or network failures. There has been extensive work [12, 13, 14] done to introduce fault tolerance and recovery to MPI implementations, but it has not advanced to the stage where it has become a part of the MPI standard or its widely used implementations. Failed MPI applications have to be re-executed, which can become a costly venture in cloud where you have to pay for the used resources whether the experiments were successful or not.

Moreover, when the quantity of computer resources is increased, the frequency of failures also increases dramatically. It has become one of the most important problems with the constant increase in computation power requirements of modern science. In large supercomputers consisting of hundreds of thousands of processor cores, it is not uncommon to have failures every few minutes or even more often [15]. For these reasons, creating reliable distributed applications is often a very complex and time consuming task.

However, a number of distributed computing frameworks have been created that advertise to greatly simplify creating distributed applications by taking care of the most complex parallel processing tasks for the user. This is typically achieved by forcing users to follow specific distributed computing models when implementing algorithms. No longer having full control over how the distributed applications are executed is the main difference between using distributed computing frameworks and parallel programming libraries such as MPI.

One of the most widely used distributed computing frameworks is MapReduce [16] which was first created by Google in 2004 for large scale data processing. Google published in 2008 that they use it to daily process more than 20 petabytes. While the Google MapReduce implementation is proprietary, there exist a number of freely available alternatives of which Hadoop [17] is the most popular in both industry and research.

The main advantages of the Hadoop MapReduce framework are fault tolerance and near-automatic parallelization for algorithms adapted to the MapReduce model. MapReduce input files are divided into smaller blocks, are replicated (3 copies by default) and are divided between machines in the cluster to provide fault tolerance for the data. When a MapReduce job is started, the framework decides which machine processes which block, executes computations in a distributed fashion and takes care of all task and data synchronization issues. When some of the MapReduce tasks fail, framework re-executes them on other machines, thus allowing applications to survive occasional machine or network failures.

When writing MapReduce applications, user only has to define map and reduce methods to specify how input data is parsed, processed, grouped by and aggregated. Everything else is managed by the framework automatically. Section 2.1.1 describes the MapReduce model in more detail.

While near-automatic parallelization and fault tolerance are definite advantages for distributed applications, the MapReduce model is mainly designed for relatively simple data processing algorithms (such as word count or inverse term frequency in documents) which do not require much data synchronization and are generally characterized as embarrassingly parallel algorithms. While the scientific computing field uses a number of embarrassingly parallel algorithms [18, 19, 4], most of the scientific computing algorithms are more complex and require many synchronization steps.

One MapReduce job consists of a single Map and a single Reduce operation and the data synchronization is only performed between these two operations, which makes adapting complex scientific computing algorithms to MapReduce a non-trivial task. How to adapt scientific computing algorithms to MapReduce is discussed in more detail in Chapter 3.

However, the advantages of MapReduce, like automatic parallelization and fault tolerance, are very useful for any distributed application and the scientific computing applications are no exceptions. For this reason we decided to study in more detail what kind of algorithms MapReduce frameworks are suitable for, and whether there are alternative frameworks which are suitable for other types of algorithms.

There have been a number of studies that have examined using MapReduce for solving scientific computing problems, but most often they do not step further from the results of individual algorithms. While some of them provide solutions for more complex scientific computing problems, it is typically in the shape of specifically designed new implementations of the MapReduce model. Examples of such frameworks are Twister [20] and HaLoop [21], which are described in Section 2.1.1. However, these frameworks often give up some of the original MapReduce advantages (such as fault tolerance), which was one of the main reasons why we were interested in this distributed computing model.

Furthermore, several technologies such as Hadoop Yarn [22] and Aneka [23] distributed computing platforms have emerged in recent years which separate the computer resource utilization, task scheduling and execution from the actual computational engine that is used to execute the user defined tasks and allow users to switch between different distributed computing frameworks on the fly. This gives us the freedom to choose different distributed computing frameworks for different types of tasks.

1.1 Problem Statement

Creating distributed applications using parallel programming libraries (such as Message Passing Interface (MPI)) which are typically used in scientific computing field can be a very difficult and time consuming task because users have to explicitly take care of many of the parallelization tasks and deal with any debugging issues.

Distributed computing frameworks (such as Hadoop MapReduce) have potential to significantly simplify creating distributed applications by managing some of the parallelization issues for the user. For example, Hadoop MapReduce framework takes care of input data partitioning, replication and distribution. It manages task delegation, computing resource allocation and re-execution of failed tasks. It also manages how intermediate data is synchronized between concurrently running processes and avoids any deadlocks and race conditions.

Utilizing such frameworks could greatly reduce the time and effort needed to create distributed computing applications. However, distributed computing frameworks typically put some restrictions on their applications, such as having to follow specific distributed computing model or only supporting certain input data types or structures. Thus they can not be expected to be suitable for all types of algorithms. In addition, performance is very critical for scientific computing applications, so distributed computing frameworks must be able to run scientific computing algorithms in an efficient manner.

The main research question we want to answer in this work is: *Can the process of parallelizing scientific computing algorithms be significantly simplified by adapting them to distributed computing frameworks, while not reducing the parallel efficiency and scalability of the result?*

The clarify, the specific expectations from adapting scientific computing algorithms to distributed computing frameworks are the following:

1. **Ease of parallel programming** - Distributed computing framework should manage parallelization tasks for the user. User should be able to concentrate more on the algorithm implementation and less on how to utilize distributed computing resources.

2. **Ease of debugging** - framework should automatically manage the task and data synchronization issues (such as race conditions and

deadlocks) that arise from using multi-core and multi-machine computing environments.

3. **Automatic fault tolerance** - Framework should automatically recover from faults and manage fault tolerance related tasks of the applications.

4. **Good parallel efficiency** - The adapted algorithms should be able to achieve good parallel efficiency. It is difficult to define what "good" parallel efficiency is for different algorithms. Thus, to provide perspective, it should be evaluated in comparison to the parallel efficiency which is achievable using MPI or other parallel programming libraries which are typically used in scientific computing field.

5. **Good scalability** - Framework should be able to efficiently utilize large number of cores. Again, to provide perspective, comparisons should be made in relation to using parallel computing libraries such as MPI.

The main hypothesis of this work is: *distributed computing frameworks can simplify creating parallel scientific computing applications without significantly losing the efficiency and scalability in comparison to using Message Passing Interface (MPI) libraries.*

To test this hypothesis we decided to study which distributed computing frameworks are suitable for parallelizing and scaling up different types of scientific computing algorithms and how good parallel speedup they are able to provide.

We define a suitable distributed computing framework as a framework which fulfills the aforementioned expectations for a given scientific computing algorithm. However, we consider the parallel efficiency to be the most important factor when comparing the suitability of different distributed computing frameworks against each other as performance is very critical for scientific computing algorithms.

The study in this direction lead us to an additional research question which we wanted to answer: *How to choose which available distributed computing framework is the most suitable for a given scientific computing algorithm?* To answer both research questions we defined the following research tasks.

1. Identify what characteristics affect the parallel efficiency and scalability of algorithms adapted to the MapReduce model.

2. Provide alternatives for algorithms for which MapReduce is not suitable.
3. Create a methodology for deciding which of the available distributed computing frameworks is the most suitable for a given scientific computing algorithm.

We chose to initially focus on the Hadoop MapReduce distributed computing framework because it was the most widely used and documented distributed computing framework at the time and was constantly updated with new features.

1.2 Contributions

The contributions of this thesis are the following.

- Scientific computing algorithm classification for MapReduce – A classification for scientific computing algorithms was created, which divides algorithms into different classes based on their structure after being adapted to the Hadoop MapReduce framework and the parallel efficiency of the result.
- Alternative approaches for scientific computing algorithms for which MapReduce is not suitable – Three approaches were proposed and evaluated:
    1. Reducing the number of iterations in algorithms to make them more adaptable for MapReduce.
    2. Using alternative MapReduce frameworks that are specifically designed for iterative algorithms.
    3. Using Bulk Synchronous Parallel distributed computing model as alternative to MapReduce.
- Dynamic Algorithm Modeling Application (DAMA) – A general purpose algorithm simulation application DAMA was created to simplify the process of choosing which distributed computing implementation is more suitable for a given algorithm. DAMA is implemented on MPI, MapReduce, Hama and Spark and it can be used to model the parallel implementations of scientific computing algorithms.

1.3 Outline

Chapter 2 summaries the state of the art and related work in adapting scientific computing algorithms to distributed computing frameworks. First section introduces distributed computing models together with their most widely used implementations and describes their characteristics and design choices. Second section describes the related work in adapting algorithms to distributed computing frameworks and measuring their performance. Last section provides an overview of previous studies which have classified scientific computing and data processing applications based on their characteristics.

Chapter 3 describes the work of adapting scientific computing algorithms to the MapReduce model. It introduces a classification for scientific computing algorithms based on the number of iterations that are required for running the algorithm. One algorithm is chosen from each of the algorithm classes and they are each implemented in the Hadoop MapReduce framework. Each implementation is benchmarked in a cluster environment with varying both the size of the cluster and the length of the execution. The results of these benchmarks are analyzed to find out how efficient Hadoop MapReduce is for parallelizing algorithms from their representative class. The issues that were encountered when adapting different classes of scientific computing algorithms to Hadoop MapReduce are discussed in detail and three different approaches are outlined to solve the aforementioned problems.

Chapter 4 discusses restructuring scientific computing algorithms for MapReduce as the first proposed approach for solving the previously encountered problems. The main idea is to try to reduce the number of iterations or choose an alternative algorithm that solves the same task and requires lower number of iterations. Two scenarios are looked at and analyzed. First is using an embarrassingly parallel linear system solver instead of the highly iterative Conjugate Gradient algorithm to solve systems of linear equations. Second is changing the way PAM clustering is applied on data to cluster it by using parallel sampling of data in the CLARA algorithm, significantly reducing the number of synchronization steps or MapReduce job iterations.

Chapter 5 examines the second proposed approach which is using alternative MapReduce frameworks that are specifically designed or modified for supporting iterative algorithms. The frameworks that we chose to investigate were Spark, Twister and HaLoop. These three frameworks are compared against a MPI library and Hadoop MapReduce framework – the expected best and worst case options for parallelizing iterative applications. Three algorithms (Conjugate Gradient, PAM and CLARA) are implemented in each of these frameworks as benchmarks and executed in a varying size cluster with different dataset sizes. The results of the benchmark experiments are analyzed to find out how well these alternative MapReduce frameworks perform in comparison to MPI and to investigate which algorithm characteristics affect the parallel efficiency of the implemented algorithms in each of the frameworks.

Chapter 6 looks at the third proposed approach which is using frameworks based on the Bulk Synchronous Parallel model as an alternative to MapReduce. The BSP implementations that we considered in this work are BSPonMPI and HAMA. We investigate how suitable these BSP implementations are for parallelizing scientific computing algorithms in comparison to MPI implementations for Java (MPJ Express and MpiJava) and Hadoop MapReduce. To compare their performance we implement both CG and PAM algorithms in each of the implementations and perform benchmarking experiments in a varying size cluster with different dataset sizes. We also investigate whether these implementations provide the same advantages as most MapReduce frameworks do, such as automatic parallelization and fault tolerance. In addition, we propose a new BSP framework – NEWT – for the Hadoop Yarn platform that can be used as an alternative to MapReduce in any existing cluster without reconfiguration. We analyze the performance of the NEWT framework in comparison to BSPonMPI, which we previously identified as the most suitable BSP implementation among the ones that we investigated. We also measure how efficient the NEWT fault tolerance and recovery mechanisms are and discuss what additional features and improvements should be added to NEWT.

Chapter 7 introduces a methodology for choosing the most suitable distributed computing framework for a given scientific computing algorithm. A Dynamic Algorithm Modeling Application (DAMA) is proposed which can model the characteristics of scientific computing algorithms and can be used to estimate the performance of those algorithms when adapted to a number of distributed computing frameworks without having to adapt, implement and debug the algorithms. DAMA is implemented on a selection of the distributed computing frameworks or libraries that were investigated in previous chapters and which have shown to be well suited for scientific computing algorithms. We show how to analyze scientific computing algorithms to identify what are their parallel characteristics, which are needed to configure the benchmark to model them. We describe how to use this benchmark to model scientific computing algorithms and illustrate the process with a number of example algorithms. We also validate the results by applying this approach to model some of the scientific computing algorithms we have investigated in previous chapters.

Chapter 8 concludes the thesis by discussing the contributions and impact of this work and outlines what further work should be done to simplify the process of adapting scientific computing algorithms to distributed computing frameworks and to make the proposed approach more applicable in general.

CHAPTER 2

STATE OF THE ART

This chapter outlines the state of the art in adapting scientific computing problems to distributed computing frameworks. In the first section we introduce the most widely adopted distributed computing models, describe their implementations and summarize what type of algorithms they are designed for. Second section outlines related work in adapting scientific computing algorithms to distributed computing frameworks. Third section gives an overview of studies which have analyzed and classified scientific computing algorithms based on their characteristics.

2.1 Distributed computing models

There exist a number of different distributed computing models but the most widely used models are MapReduce [16], Bulk Synchronous Parallel (BSP) [24] and Message Passing Interface (MPI) [11]. This section introduces these models, outlines their differences and describes a number of their implementations that are freely available for use.

2.1.1 MapReduce model

MapReduce [16] was developed by Google as a distributed computing model and a framework for performing reliable computations on a large number of commodity computers.

An application following the MapReduce model consists of two methods: map and reduce. Its input is a list of key and value pairs and each of

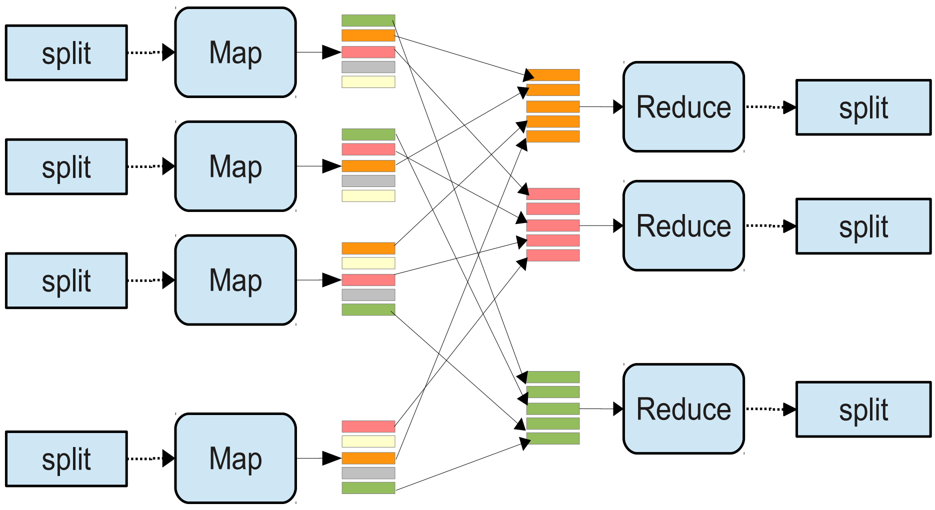

Figure 2.1: MapReduce model

the pairs are processed separately by map tasks which output the result as one or more key-value pairs.

$$map(key, value) \Rightarrow [(key, value)]$$

Map output is partitioned by keys into groups which are in turn divided between different reduce tasks. Input to a reduce method is a key and a list of all values assigned to this key.

$$reduce(key, [value]) \Rightarrow [(key, value)]$$

Reduce method aggregates the output of the map method. It gets a key and a list of all values assigned to this key as an input, performs user defined aggregation on it and outputs one or more key-value pairs.

Hadoop MapReduce

Hadoop MapReduce[16] is the most widely used implementation of the MapReduce model. It was inspired by Google MapReduce implementation, which was first introduced by Google in 2004 [25], but it was not made publicly available for use. Hadoop is managed by Apache and its development is backed by many well-known companies like Yahoo, FaceBook and Twitter, but at the same time it is still open source and freely usable by everyone.

In recent years it has overtaken many of it's alternatives such as Microsoft Dryad [26] and even Google is using Hadoop to provide general purpose large scale data processing service to it's Google Cloud customers. [27] This is likely because their own custom MapReduce implementation is more specialized for specific tasks.

MapReduce framework takes care of everything else from data partitioning, distribution and communication to task synchronization and fault tolerance, greatly simplifying the writing of distributed applications as the user only has to define the content of the Map and Reduce tasks. Near automatic parallelization is achieved by executing map and reduce tasks concurrently across machines in the cluster and partitioning the input data between them.

Hadoop uses Hadoop Distributed File System (HDFS) [28] to store and distribute data to be processed. Data stored in HDFS is automatically divided into smaller blocks (64 Megabytes by default) and replicated on a number (3 by default) of different physical locations in the cluster. The map and reduce tasks in Hadoop are executed where the data is located in HDFS to avoid unnecessary data transfer. Having multiple replicas allows the framework to balance the workload of the machines in the cluster and to achieve automatic fault recovery.

If a map task working on a specific data block fails, the framework checks where are the other replicas of this block located and moves the failed map tasks there. If a reduce task fails, it is re-executed on another location and its input data is transfered again from previously finished map tasks. However, machine or network failure in the reduce stage may also require re-executing some of the map tasks if they were originally also located on the failed node.

It has been shown [29] that Hadoop MapReduce is suitable for many data processing applications from information retrieval and indexing to solving graph problems, like finding graph components, barycentric clustering, enumerating rectangles and triangles. Hadoop MapReduce has also been tested for solving embarrassingly parallel scientific computing problems [30, 19, 4] and it performed well for algorithms such as Marsaglia polar method, integer sort and Monte Carlo methods. However, it had sig-

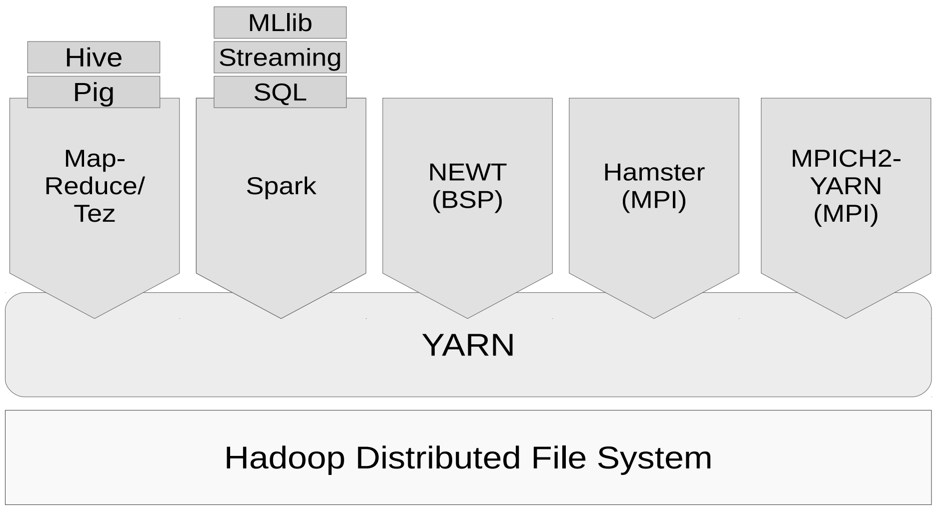

Figure 2.2: Apache Hadoop YARN distributed computing platform

nificant problems with more complex applications and especially iterative methods.

Because of Hadoop MapReduce problems with more complex algorithms and other limitations and disadvantages which are covered in more detail in chapter 3, Hadoop has slowly been redesigned to be more flexible. In Hadoop version 2.0, the computing resource management and task scheduling was separated from the MapReduce job execution with the introduction of YARN (Yet Another Resource Negotiator) cluster management system. YARN transformed Hadoop into a higher level distributed computing platform where it's possible to use multiple distributed computing frameworks at the same time and a number of projects have been initiated to bring BSP or MPI implementations to Hadoop as alternatives to MapReduce.

Figure 2.2 illustrates the YARN platform with a selection of distributed computing frameworks which can be used in YARN. Such as Spark as a MapReduce-like alternative, Tez as advanced MapReduce, NEWT [6] as a BSP framework and Hamster and MPICH2 as MPI implementations. It's entirely possible to create a data processing workflow in a Hadoop cluster which uses different distribute computing frameworks or parallel programming libraries at different stages of the process.

However, while alternative distributed computing frameworks are a step in the right direction, it is a complex task to start using completely dif-

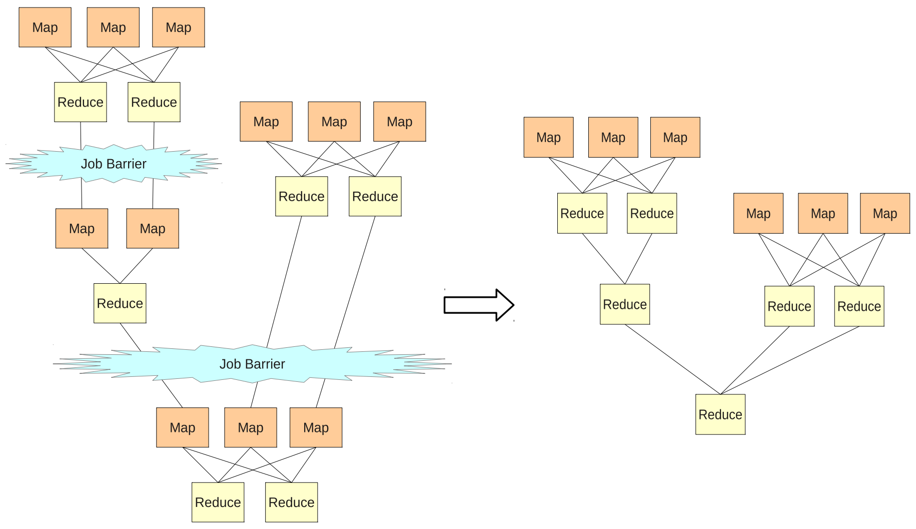

Figure 2.3: Job design - from MapReduce to Apache Tez

ferent parallel programming models and paradigms in existing applications that already use MapReduce. Thus another Hadoop project was started to create a more capable direct replacement for MapReduce - Apache Tez [31], which does not require changing the existing applications.

Apache Tez

Apache Tez [32] is a MapReduce framework which became a part of the Apache software foundation in 2014. It is designed to be a direct replacement for Hadoop MapReduce for higher level frameworks that depend on Hadoop to perform computational tasks, such as Pig and Hive. It moves away from the restrictive MapReduce model consisting of a sequential execution of map and reduce tasks and allows to create a arbitrary size Directed Acyclic Graph (DAG) or user defined tasks. Execution of multiple reduce tasks in a sequence is not efficient in Hadoop MapReduce, which is required for tasks that require regrouping data multiple times. Reduce tasks are required to write data to the HDFS and another reduce task can not be executed without first executing a new Map task and then shuffling and merging the data again.

Figure 2.3 illustrates the conceptual move from MapReduce to Tez. Tez allows to create direct data flow links between sequentially executed tasks and does not restrict their execution order. Everything is executed as a single Tez job with virtually unlimited number of internal tasks.

A Tez job consists of a Directed Acyclic Graph (DAG) of vertices and edges. DAG defines the execution order of data processing tasks. Each DAG vertex represents a single task and describes what data processing operation is applied on the data. Each DAG edge defines how the data is moved between vertices.

Edges have multiple properties that define the behavior of the Tez DAG. A data-movement property defines how the data gets from the edge input (data producer) vertex to the edge output (data consumer) vertex. Scheduling properties define whether the consumer vertex can be executed before the producer vertex is finished. Data-source properties defines how the produced data is stored. Whether it is persisted on disk in a reliable manner or is it only kept in memory in a non fault tolerant manner. Data-movement property can be used to define what data synchronization pattern is applied to move the data, such as one-to-one, broadcast or scatter-gather.

While Tez does not solve all the disadvantages of Hadoop MapReduce it should still significantly improve the efficiency of executing iterative applications in a Hadoop cluster.

Spark

Spark [33] is an open source framework for large scale data analytics. It supports both in-memory and on-disk computations in a fault tolerant manner by introducing resilient distributed datasets (RDD) that can be kept either in memory or on disk. The data kept in the RDD is automatically partitioned and distributed across machines in the cluster and can be replicated to provide data recovery in case of failures.

A computation is defined by applying different Spark defined operations on the RDD's such as map, group-by and reduce. Spark also supports joins, unions, co-grouping and Cartesian products on RDD's and thus extends the framework capabilities beyond the simple model of MapReduce.

Spark is not strictly a MapReduce framework, as it supports many data processing operations other than map and reduce, such as join, filter, cogroup and union. But considering that it's main data processing operation is map, and reduce operation is often implemented using the combination of group–by and map operations, we consider it to be a MapReduce–like framework.

Fault tolerance is achieved by keeping track of operations applied to RDD's and in case of machine or network failures, lost RDD partitions are reconstructed by backtracking the applied operations and rebuilding the RDD partition from the latest intermediate partitions that still exist. In contrast to fault recovery by checkpointing, there is no overhead when there are no failures.

Spark is very actively developed and contains many additional features that extend core Spark, such as Spark Steaming for processing streaming data, SparkSQL for using SQL commands to process and analyze data, SparkR for using Spark inside R applications and GraphX for graph processing.

Twister

Jaliya Ekanayake et al. [20] have studied using MapReduce for data intensive science. They presented their experience with implementing two typical scientific analyzes: High Energy Physics data analysis and k-Means clustering. They also presented CGL–MapReduce, a streaming-based MapReduce implementation. They concluded that the use of iterative algorithms by scientific applications limits the applicability of the existing (at the time) MapReduce implementations like Hadoop.

However, they also strongly believe that MapReduce implementations specifically designed to support such applications would be very valuable tools. Thus they used the proposed CGL–MapReduce design to create a new MapReduce framework for iterative scientific applications, Twister [34]. Twister distinguishes between static data that does not change and normal data that may change at each iteration. It also provides better support for iterative algorithm by enabling long running map and reduce tasks which do not have to be terminated between MapReduce executions.

Twister uses no distributed file system and partitioning the input data is a manual process. It also only guarantees restoring input data that can be reloaded from the file system or static parameters inherited from the main program and any transient information stored in Map and Reduce tasks will be lost in case of failures. The lack of both a distributed file system and a fully working fault recovery mechanism are significant disadvantages.

HaLoop

HaLoop [21] is built on top of Hadoop version 0.20.0 and it directly extends the MapReduce framework by adding support for iterative execution of Map and Reduce tasks, adding various data caching mechanisms and making the task scheduler loop-aware. The authors separate HaLoop from Twister by claiming that it is more suited for iterative algorithms because using memory cache and long running MapReduce tasks makes Twister more prone to failures.

HaLoop can reuse existing Hadoop Map and Reduce code without modifications by specifying how they are executed in iterative manner in an encapsulating iterative job configuration. Users have to define a couple of additional classes from distance measurement for loop ending condition to caching configuration. While creating the applications themselves is not difficult as long as the user has experience with Hadoop, debugging and creating more complex job chains can be a daunting task as the documentation is sparse and the framework itself is no longer in active development.

However, HaLoop has not been updated since June 2012 and is only compatible with Hadoop version 0.20.

Nephele

Daniel Warneke et al. introduced Nephele [35] as an alternative to Hadoop MapReduce. It is a cloud based distributed computing framework for large scale data processing. It is specifically designed to take advantage of cloud characteristics from elasticity to dynamic resource allocation and real-time provisioning.

Nephele uses the cloud computing services to provision computing resources on-demand based on the resource requirements of the user tasks. The authors have compared Nephele to Hadoop MapReduce using typical data algorithms as benchmarks and showed it to be several times faster and able to utilize the cloud infrastructure much more efficiently than Hadoop.

2.1.2 Bulk Synchronous Parallel model

Bulk Synchronous Parallel [24] (BSP) is a distributed computing model for parallel iterative algorithms where the input data is divided between concurrently working tasks and the computation process consists of a sequence of iterative super-steps. Each super-step is divided into three smaller steps:

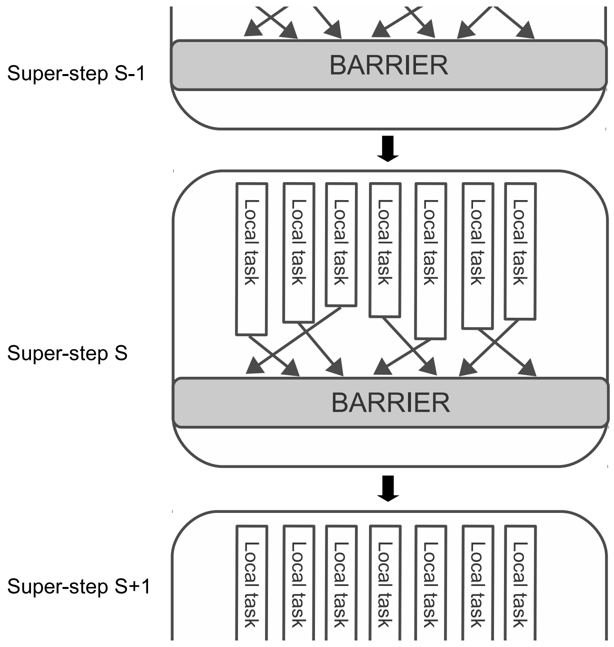

Figure 2.4: Illustration of the BSP execution flow. [1]

local computation, synchronization and global barrier. In the local computation step, a user defined function is applied to every sub-task concurrently, which effectively parallelizes the computation.

At the synchronization step, data which is needed by other sub-tasks is sent to them. However, this data is not available for the destination sub-tasks until the next super-step, which means there will be no deadlocks. As the last step, sub-tasks notify that they have finished working and wait till all other sub-tasks have also done so, before they continue to the next super-step. This process is illustrated on Figure 2.4.

Bulk Synchronous Parallel model works well for iterative applications where the whole computation is divided into sub-tasks of equal size which are then executed concurrently. However, if the computations are divided unequally, the efficiency of the algorithms drops, as the faster sub-tasks have to wait for the slower tasks to finish their work before they can proceed. Choosing the right data partitioning is very important when creating BSP applications.

A BSP algorithm is defined by how the whole computation is divided into sub-tasks, how and what data is synchronized on every super-step and what is the user defined function that is applied to each sub-task at every super-step. Some of the most widely used BSP implementations are introduced in the following sections.

BSPlib

Jonathan M. D. Hill et al. [36] introduced BSPlib as a MPI-like BSP library and demonstrated how it can be utilized for scientific and industrial applications by implementing use cases like Fast Fourier Transform (FFT), sorting, and molecular dynamics. However, the Oxford BSPlib can be considered a legacy implementation as its last update was in 1998. Using it effectively on recent hardware and software platforms and especially on 64 bit systems is very complicated and it is certainly not optimized for them.

The BSPlib library API has been used as an interface in other BSP implementations like BSPonMPI [37] and Paderborn University BSP Library (PUB) [38].

BSPonMPI is a library for creating parallel programs using the BSP model. It implements the BSPlib standard and uses Message Passing Interface (MPI) [11] for the underlying communication. BSPonMPI runs on all the same platforms as MPI, which makes it more usable than Oxford BSP toolset. It also has no fault tolerance and any failed jobs have to be rerun from the beginning.

Paderborn University BSP is also a BSP implementation that follows the BSPlib API. It is written in C, supports on a number of different platforms (PC, Cray T3E, IBM SP/2, Sun workstations) and communication methods (MPI, TCP/IP, Parsytec Parix) but has not been updated since 2002.

Pregel

While the initial implementations of the BSP model (Oxford BSPlib, BSP-on MPI) were not taken into a wide spread use, a new wave of BSP frameworks have been initiated recently after Google published using the BSP model for their new large scale graph processing framework Pregel [39]. It was developed to address MapReduce's inability of supporting iterative graph processing algorithms.

Pregel uses checkpointing to provide implicit fault tolerance. It stores the state of the application in persistent storage between supersteps at user configured intervals. Google's implementation is proprietary and can only be used in-house to solve various graph related problems but a number of Pregel like frameworks have since been created, such as Apache Giraph [40], Stanford GPS [41] and Hama [42].

Apache Giraph

Apache Giraph [40] is an iterative graph processing framework inspired by Pregel. It leverages existing Hadoop infrastructure to run Hadoop MapReduce like jobs that use the BSP model instead. In contrast to Hadoop MapReduce, concurrent tasks are allowed to communicate between each other and the computation is divided into a number of supersteps. Users create a single BSP function which is repeated on each Vertex at every iteration. Creating more dynamic BSP applications where different data processing tasks are executed at different iterations can be difficult.

Giraph also provides fault-tolerance by using Zookeeper [43] as a distributed coordination service for the concurrently running tasks and performing checkpointing with user specified frequency. Every x'th iteration intermediate graph data is stored to HDFS and if any process fails, the computation can be rolled back to the superstep when the latest checkpoint was stored.

Stanford Graph Processing System

Stanford Graph Processing System (GPS) [41] is a Bulk Synchronous Parallel framework developed in Stanford University. It is open source and supports several features not present in either Google Pregel or Apache Giraph. These are support for algorithms that include global as well as vertex-centric computations (whereas the Pregel API focuses only on vertex.centric computations), the ability to repartition the graph during processing and partitioning adjacency lists of high-degree vertices to reduce the amount of communication. Stanford GPS also uses checkpointing to achieve automatic fault recovery.

Pregel, Apache Giraph and Stanford GPS are specifically designed for graph algorithms, and while it is almost always possible to represent other types of distributed applications as graph computations by restructuring them, it would also require extra effort from the programmer and may also lower the parallel efficiency of the result.

Hama

Hama [42] is a distributed computing framework that was initially developed to perform matrix operation in the Hadoop MapReduce framework. However, they have since then moved away from the MapReduce model and started utilizing the BSP model for general purpose computations instead. It is also no longer specialized for matrix computations.

Hama follows the original BSP model relatively closely, yet at the same time it tries to take advantage of the Hadoop framework, and HDFS in particular. In Hadoop, the HDFS is used for data storage, distribution, and as an input and output destination for MapReduce applications. Thanks to using file block replication, HDFS provides fault tolerance mechanism for data.

Hama also exploits these advantages by using HDFS to distribute the initial input of the BSP application and also for its output. However, in comparison to executing MapReduce jobs in an iterative fashion, Hama does not write all the intermediate data to HDFS and thus it does not provide full fault tolerance for running applications.

To write a Hama program, user has to define a $bsp()$ method, which reads input from HDFS and writes its output back. This method is not executed in an iterative fashion as per BSP model. User has to implicitly define how the supersets are executed by creating a custom loop inside the method.

A Hama superstep is defined as any executed code that ends with a barrier synchronization by using the method $sync()$. The $bsp()$ method is executed concurrently on all the machines in the cluster, on which the Hama framework is set up. These concurrent tasks can send messages to each other between the supersteps, however, these messages can only be read after the next barrier synchronization. Input and Output formats and the messages that are sent between concurrent tasks are identical by design to Hadoop MapReduce $Writable$ objects, simplifying the work for users who are familiar with writing MapReduce applications.

NEWT

NEWT [6] is an alternative BSP-inspired distributed computing framework for the Hadoop YARN platform. It was created to support executing iterative applications in Hadoop clusters while retaining automatic fault recovery and achieve near-automatic parallelization. It also does not require any software installation or reconfiguration in the YARN clusters. This framework is described in more detail in section 6.2 together with its modified BSP model, design choices and performance measurements.

2.1.3 Message Passing Interface

Message Passing Interface (MPI) [11] is a language–independent communications protocol used to program parallel computers. It specifies an API for sending messages between concurrently running processes in a cluster. MPI does not restrict applications into a specific distributed computing model, and can actually be used to implement libraries of frameworks for most distributed computing models. MPI has greatly advanced the development of a parallel software industry, and encouraged the development of large-scale parallel applications.

It provides efficient communication between processes working on different physical machines, it gives programmers full control over parallelization and has a number of predefined and optimized collective communication methods to deal with a large number of processes. Over the years it has become the de facto library for communication among processes in parallel applications for these reasons.

However, in comparison to MapReduce, MPI applications are not able to handle machine or network failures by design. Extensive work has been done to introduce fault recovery to MPI, but it has not become a part of the MPI standard or its widely used implementations. [13, 14] Also, in comparison to using previously mentioned distributed computing frameworks in this chapter, writing MPI applications requires very low level programming. Programmers have to specifically take care of input data partitioning & distribution, data & task synchronization, avoiding deadlocks & race conditions, etc. In addition, validating and debugging MPI applications can be as difficult as creating the application itself.

As most of the MapReduce and many of the BSP frameworks use Java programming language and we are interested in comparing their performance to MPI, we are mainly interested in MPI implementations for Java. We chose MPJ Express [44] and MpiJava [45], because they are the two most widely used MPI libraries for Java HPC applications and they both use different underlying message passing implementations, as described in the following subsections.

MpiJava

MpiJava [45] provides an object oriented Java interface to MPI that can work with any native MPI library written in C or C++ that implements the MPI API, such as OpenMPI [46] or MPICH2 [47]. It uses Java Native Interface (JNI) to execute MPI commands from the native MPI library and to pass data between the native language and Java.

OpenMPI is an open source and freely available general purpose MPI implementation which implements the full MPI-2 standard. It's goal was to combine advantages of the previously existing MPI implementations (FT-MPI, LA-MPI, LAM/MPI) into a single consistent and stable open source MPI framework.

MPICH is a high-performance implementation of the MPI standard. Its main goal is to provide an MPI implementation that supports different software and hardware platforms like commodity clusters (desktop systems, shared-memory systems, multi core architectures), high-speed networks (10 Gigabit Ethernet, InfiniBand, Myrinet) and proprietary high-end computing systems (Blue Gene, Cray, SiCortex) [47].

MpiJava has no fault tolerance because the underlying OpenMPI and MPICH2 libraries do not provide any. When using MpiJava the algorithms still run on Java and only the actual MPI messages are transfered through the underlying native libraries. Thus, the performance of concurrent Java tasks is not affected and it is possible to directly compare the performance of the data synchronization and the parallelization process in general.

MPJ Express

MPJ Express [44] is a freely available open source Message Passing Interface (MPI) library written in Java that allows developers to write and execute parallel Java applications on multi core processors, compute clusters and clouds. MPJ Express implements the MpiJava 1.2 API and thus is interchangeable with MpiJava for any Java application that uses this API. In comparison to MpiJava, MPJ Express uses Java New I/O (NIO) for communication instead of depending on JNI to utilize native MPI libraries like MPICH2 or OpenMPI. Similarly with MpiJava, MPJ Express provides no fault tolerance.

	Progr. model	Algorithms	Fault tolerance	Data handling	Limitations	Additional features	Language
Hadoop MapReduce [16]	MapReduce	Embar. parallel, text processing.	All input and output data is replicated. Task state is not preserved. Failed map and reduce tasks are restarted.	All input and output data is stored in HDFS, partitioned into smaller blocks and replicated.	No real support for iterative execution. Long job configure time ($\sim$17 sec).	Speculatively executes a replica of the slowest running tasks on already finished nodes. Many other programming languages other than Java can be used through Hadoop Streaming.	Java
Tez [32]	MapReduce	Embar. parallel.	All input and output data is replicated. Task state is not preserved. Failed map and reduce tasks are restarted.	Uses HDFS for data storage, distribution and replication.	It is mostly designed for other frameworks that compile higher level code into MapReduce and is not as convenient to use as Hadoop MapReduce.	Supports executing multiple reduce tasks in a row.	Java
Spark [33]	MapReduce	Any	Can rebuild missing partitions of RDD's based on the history of applied operations and still existing partitions. Provides an API for more static checkpointing but leaves the decision of which data to checkpoint to the user.	Data stored in RDD is automatically distributed. User chooses when and which data to keep in memory or on disk and whether to use replication.	Hard to estimate the data skewness inside RDD's.	Supports fully in-memory computations, but does not enforce it. Contains many methods other than Map and Reduce, such as filter, join, cogroup and sample.	Scala, Java, Python, R
Twister [20]	MapReduce	Iterative.	Twister client can detect faults and try to restore the computation from the last iteration. Only guarantees restoring inherited static parameters or input data which can be reloaded from file system.	No distributed file system, partitioning data is a manual process.	Input and intermediate data must fit into the distributed memory of the cluster, otherwise program will fail.	Supports fully in-memory computations. Enables long running map and reduce tasks which do not have to be terminated between MapReduce executions.	Java
HaLoop [21]	MapReduce	Iterative.	Re-executing failed Map or Reduce tasks. No real fault tolerance for the long running iterative MapReduce tasks.	Uses HDFS for input/output data replication.	No developments past prototype, only supports Hadoop 0.20.0. Intermediate data between Map and Reduce tasks is still stored to disk.	Loop-aware task scheduling ensures that Map and Reduce tasks process the same partition every iteration. Can directly reuse existing Hadoop MapReduce code. Uses caching for static data.	Java

Table 2.1: Comparison of the distributed computing frameworks

	Progr. model	Algorithms	Fault tolerance	Data handling	Limitations	Additional features	Language
BSPlib [36]	BSP	Iterative applications.	Generally same as MPI.	Explicit	Manually placed barriers when synchronization is needed. Difficult to install on newer hardware.	BSP model is not strictly enforced.	C, Fortran
Pregel [39]	BSP	Graph processing.	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Each process is associated with a vertex value	Proprietary, can not be used directly.	Program consists of a single function, which is continuously executed on each vertex.	C++
Apache Giraph [40]	BSP, Pregel	Graph processing.	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	It can be complicated to create more dynamic applications where different operations are executed at different iterations.	It extends Pregel by introducing master computations, sharded aggregators and edge-oriented input.	Java
Stanford GPS [41]	BSP, Pregel	Graph processing.	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	Uses it's own domain specific programming language, which makes it more difficult to start with.	Extends Pregel with global computations instead of just vertex centric ones. Includes optimizations to graphs repartitioning and reducing network I/O.	Green-Marl
Hama [42]	BSP	Iterative, Graph processing.	Only supports partial fault tolerance by checkpointing messages. Full fault tolerance is still under development.	Uses HDFS to distribute input and output data.	User has to explicitly place barriers when synchronization is needed, similar to BSPlib.	BSP model is not strictly enforced. User does not have to use either BSP loop or barriers.	Java
NEWT [6]	BSP	Iterative.	Checkpoints and recovers intermediary state/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	Creates a separate process for each input file. (Alternative approaches under consideration)	Programs consist of an arbitrary number of labeled functions with user-defined transitions.	Java
MPI (MpiJava [45] & MPJ Express [44])	Generic Message Passing	Any	Not part of MPI standard or its major implementations. Existing transparent solutions store entire task address space.	Data partitioning, distribution and synchronization is explicitly defined by the user.	Can be difficult to debug and optimize. Requires low level programming. Race conditions and can be an issue.	Provides full control over parallelization. Has a number of optimized collective communication methods to deal with a large number of processes.	Java, C, C++, Python

Table 2.1: Comparison of the distributed computing frameworks (continued)

2.1.4 Other distributed computing frameworks

Aneka

Aneka [23] is a distributed computing platform for developing and deploying parallel applications on cloud computing platforms. It is specifically designed to provide users with a development platform for creating distributed applications that can automatically scale on demand. Aneka also tries to take advantage of elasticity and scalability of cloud computing platforms and allows the utilization of multiple different programming models (MapReduce, distributed thread and independent bag of tasks) to provide support for conceptually different application design scenarios from different domains such as engineering, life sciences, and industry.

Another aim of Aneka is to simplify the creation of distributed cloud computing applications by allowing to express the logic of distributed applications in multiple different ways and by providing a framework that takes care of the distributed execution of applications. Users can create static Aneka computing clusters using desktops, servers or even existing clusters and are later able to extend these clusters using public cloud services when the demand raises beyond configured thresholds creating a hybrid cloud.

2.1.5 Summary

Table 2.1 provides an overview of the distributed computing frameworks and parallel programming libraries that were covered in this section. For each of the frameworks, it describes what are their distributed programming model, what algorithms they are specifically designed for, fault tolerance, data handling, limitations, unique features and programming languages.

Most of these solutions utilize HDFS for data storage, partitioning and distribution because it simplifies both creating and using distributed computing frameworks. For example, one of the biggest issues with using Twister is that it does not use a distributed file system and partitioning and distributing input data is a manual process. Users have to manually divide input data into smaller files, and create a file distribution schema, which describes where each block of the data should be located.

While many of the distributed computing frameworks are designed for iterative algorithms, they can just as well be used for non-iterative algorithms, as we can simply set the number of iterations to be 1. However, it is not easy to utilize Pregel based frameworks for anything other than graph processing frameworks, as the Pregel model defines computations using graph notations of vertices and edges. Hama and BSPlib follow the original BSP model and can be used for any type of algorithms.

Apache Spark and Hadoop are the most actively developed distributed computing frameworks in this table. Both have a large number of additional features that extend the core framework and both are constantly update and improved. HaLoop and BSPlib in contrast have not been updated in a significant amount of time.

The most important characteristics of the distributed computing frameworks in the context of this work are support for iterative algorithms, fault tolerance to support long running scientific computing applications and the parallel speedup and efficiency of the adapted algorithms. While support for iterative algorithms and fault tolerance can be evaluated by studying documentation and reading published articles, third characteristic requires more systematic study of the framework and any of its existing evaluation studies. In addition, int the context of scientific computing where performance is critical, it is important to compare the performance of distributed computing frameworks to parallel programming libraries which are typically used in this field, such as MPI libraries.

Next section gives an overview of related work in adapting algorithms to distributed computing frameworks and measuring their performance.

2.2 Adapting algorithms to distributed computing frameworks

This section gives an overview of related work which have studied adapting algorithms to different distributed computing frameworks or investigated the performance of different distributed computing frameworks.

As previously mentioned in section 2.1.1, there are a number of studies which have investigated applying MapReduce for solving embarrassingly parallel scientific computing problems.

Chris Bunch et al. [30] investigated which scientific computing problems are adaptable to MapReduce and which are not. They adapted five benchmarking algorithms from NAS Parallel Benchmarks (NPB) [48] to MapReduce and measured their performance. NAS Parallel Benchmark consists of set of algorithms designed for evaluating the performance of distributed systems and supercomputers.

These algorithms were embarrassingly parallel Marsaglia polar method for generating random numbers, integer sort using linear time bucket sorting algorithm, iterative Conjugate Gradient method for solving systems of linear equations, Fast Fourier transform for solving three-dimensional partial differential equations and block tridiagonal method for solving systems of linear equations.

MapReduce performed well for the embarrassingly parallel algorithms: Marsaglia polar method and integer sort. However, it performed very poorly for Fast Fourier Transform, Block Tridiagonal and Conjugate Gradient algorithms. Authors had to omit the numerical results of running Conjugate Gradient algorithm in MapReduce stating that it took an unreasonable amount of time to execute. The main conclusion of their work is that MapReduce is not well suited for iterative algorithms. They propose that increasing the computation to communication ratio in the adapted iterative algorithms would improve their efficiency but they do not investigate what exactly makes MapReduce unsuitable for iterative algorithms. Investigating these reasons in more detail was one of our main motivations for creating a scientific computing algorithm classification for MapReduce.

Tolga Dalman et al. [19] investigated utilizing Hadoop MapReduce and on-demand resources from Cloud for performing Metabolic Flux Analysis (MFA). They implemented an embarrassingly parallel Monte Carlo simulation method on MapReduce and used the Amazon Elastic MapReduce (EMR) service [49] to execute computations on automatically scalable cloud resources.

Their initial benchmarking experiments in an 64 core EMR cluster only managed to obtain a parallel speedup of 17. Relative parallel speedup measures how many times faster is parallel execution of an application in comparison to running the same application using only a single processor core. They identified that the low parallel speedup was a result of MapReduce framework generating a high number of small files which in turn caused a large number of I/O operations to be performed in the Reduce stage.

However, they managed to increase the parallel speedup from 17 to 48 by optimizing their MapReduce application to make better use of built-in data data types and tuning the size of data split sizes to reduce the number of individual files created by the framework. The result of this work confirm that it is possible to obtain reasonable parallel speedup by adapting embarrassingly parallel algorithms to MapReduce. However, it also indicates that it is not sufficient to simply adapt an algorithm to MapReduce. It is important to know what affects the parallel speedup and efficiency of MapReduce applications and take it into account when adapting algorithms.

Foto N. Afrati et al. [50] studied the trade–off between parallelism and communication cost in MapReduce applications. They modeled the performance of MapReduce applications that can be solved with a single MapReduce job, such as finding triangles in graphs, matrix multiplication and finding strings with a specific Hamming distance.

They identified that replication rate and reducer input size were the main parameters that affected the balance between communication cost and parallelism. Replication rate measures how many map task key-value pair output's are generated for each input key-value pair on average. Reducer size reflects the maximum size of the input value list associated with a unique reduce input key. Increasing replication rate can lower the reducer size and increase the number of reduce tasks and increase the parallelism. Reducing replication rate can increase the reducer size and lower the communication cost. They model the balance between Replication rate and Reducer size values and apply the model to lower the cost of running MapReduce applications.

Their approach is only really applicable for a single MapReduce round of computation and they state that applying their approach to iterative MapReduce algorithms requires further work. While we also investigate algorithms which can be adapted using only a single MapReduce job in this thesis, our main focus is on more complex iterative algorithms which would require more MapReduce jobs to be executed.

Jaliya Ekanayake et al. [51] compared the performance of Hadoop MapReduce to CGL-MapReduce [20], Microsoft Dryad [26] and MPI for a number of typical scientific algorithms and showed that CGL-MapReduce can greatly reduce the overhead of iterative MapReduce applications. CGL-MapReduce was a framework created by the authors of this paper and was later used as a basis for the Twister [34] iterative MapReduce framework. It was specifically designed to be suitable for iterative applications.

The algorithms they used to compare these frameworks were iterative k-means clustering, iterative matrix multiplication, High Energy Physics (HEP) data analysis and Cap3 gene sequence assembler. MPI was used only used for parallelizing the two iterative algorithms: k-means clustering and iterative matrix multiplication. In both cases, it performed the best in comparison to the three MapReduce implementations and CGL-MapReduce was close second. Both Microsoft Dryad and Hadoop MapReduce produced significantly worse result, especially in the case of the iterative k-means algorithm, where they were more than 10 times slower than CGL-MapReduce.

Hadoop MapReduce, CGL-MapReduce and Microsoft Dryad had a very similar performance in the case of Cap3 algorithm, which a simple algorithm only requiring a single map operation. In the case of High Energy Physics (HEP) algorithm, CGL-MapReduce was clearly faster than both Hadoop MapReduce and Microsoft Dryad with Hadoop MapReduce being significantly slower than the other two.

The results show that CGL-MapReduce (a framework specifically designed for iterative algorithms) can greatly reduce the overhead of iterative MapReduce applications in comparison to Microsoft Dryad and Hadoop MapReduce but it still has performance issues in comparison to MPI. These results also confirm that it is important to compare the results of benchmarking the performance of distributed computing frameworks to the performance of MPI. It helps to provide perspective for evaluating the performance results, as a framework being faster than Hadoop or Dryad might not indicate it is a good alternative if it itself is much slower than parallel programming libraries like MPI.

Junbo Zhang et al. [52] proposed and adapted a rough set based method for knowledge acquisition to MapReduce as a single MapReduce job and compared its performance on three different MapReduce Frameworks: Hadoop MapReduce, Twister and Phoenix. Phoenix [53], is a shared-memory MapReduce implementation for multi-core single machine systems.

They used the adapted algorithm to process datasets of different sizes and measured runtime and parallel speedup for each of the frameworks. Phoenix experiments were executed on a single machine with 64GB of memory and 64 cores, while Hadoop and Twister experiments were executed in a cluster, which consists of 8 machines with 8GB to 16GB of memory each. Hadoop was significantly (5 to 19 times) slower than both Twister and Phoenix in all the experiments and Twister was faster than Phoenix in most of these experiments.

Their results show that using Hadoop MapReduce can result in much worse performance even in the case when the algorithm is not iterative. Unfortunately the authors do not identify what exactly causes Hadoop

MapReduce to be 5 to 19 times slower than Twister and Phoenix. This means that it should not be assumed that MapReduce is suitable for all types of non-iterative algorithms.

Yingyi Bu et al. [54] investigated using HaLoop instead of Hadoop for large-scale iterative data analysis by implementing iterative PageRank, descendant query and k-means algorithms in both frameworks and analyzing the results on different datasets. HaLoop performed better than Hadoop in each of their test cases showing that it is more suitable than Hadoop for these types of application.

They conclude that HaLoop greatly improves the overall performance of iterative data analytics applications, however in the case of k-means the difference was very small. They did not compare HaLoop to MPI or other widely used lower level distributed computing libraries or frameworks, so it is hard to put these results in perspective. Without comparing the obtained results to lover level parallel programming libraries such as MPI, it stays unclear how significant the HaLoop performance improvements really are.

Benedikt Elser & Alberto Montresor [55] compared the performance of Apache Hama to Hadoop MapReduce and a number of large scale graph processing frameworks: GraphLab [56], Stratosphere [57] and Apache Giraph [40]. They implement the k-core decomposition graph processing algorithm on each of the frameworks. It is an iterative graph processing algorithm which goal is find subgraphs of the original graph where the minimal degree of vertices is k.

They found that Hama a close competition to GraphLab, generally performs better than Stratosphere and was better than Giraph and Hadoop MapReduce in all their experiments. This indicates that BSP frameworks like Hama can be very potent competitors to MapReduce frameworks. In addition, compared to Giraph and GraphLab, Hama does not enforce using graphs as an input and can be used for more generic algorithms. However, the authors only used a single algorithm as a benchmark to compare these frameworks and did not look further from graph processing algorithms.

There are number of benchmarking suites which can be used to evaluate the performance of distributed computing frameworks. HiBench [58] is a Hadoop benchmarking suite consisting of a number of MapReduce applications for evaluating the performance of a Hadoop clusters. It contains benchmarks from four different categories. Micro benchmarks (Sort, WorkCount, TeraSort), Web Search (Nutch Indexing and Page Rank), Machine Learning (Bayesian Classification, k-means) and HDFS Benchmark (EnhancedDFSIO).

HiBench was initially only implemented for Hadoop MapReduce, so it could not be directly used to compare the performance of different distributed computing frameworks, but it has since been updated to also support Apache Spark. Still, the number of supported distributed computing frameworks is small, there is no support for MPI and k-means is the only iterative benchmark it contains.

Similarily, BigBench [59] is a benchmarking suite for Hadoop, Hive and Spark. It aims to be a end-to-end big data analytics benchmark suite. It consists of 30 data analytics queries which are executed on structured and semi-structured data and they simulate a conceptual customer and sales analysis software stack. BigBench is specifically designed and optimized for online analytics and is less applicable than HiBench in the context of scientific computing algorithms.

One of the most elaborate benchmarking suite for distributed computing frameworks is BigDataBench [60], which is an open source big data benchmarking suite consisting of 33 different benchmarks and 6 datasets. It contains benchmarks from 6 different domains: search engine (Grep, WordCount, Sort, etc.), social networks (K-means, Triangle Count, etc.), e-commerce (Bayes, Join, etc.), multimedia analytics (Face detection, Speech Recognition, etc.) andÂ bioinformatics (SAND, BLAST).

It supports several different parallel programming solutions (MPI, MapReduce, Spark and Flink) and a number of online analytical processing frameworks (Shark, Impala and Hive). However, not all benchmarks are implemented on all the parallel programming solutions. For example, Multimedia analytics benchmarks are almost exclusively implemented on MPI library and E-commerce benchmarks are mostly implemented on online analytical processing frameworks.

BigDataBench is a very versatile benchmark thanks to the large number of different benchmarks and supported frameworks. However, having large number of benchmarks also means that it takes a lot of effort to adapt the whole suite to additional distributed computing frameworks as each of the existing benchmarks would have to be individually implemented on the new framework.

In addition, while BigDataBench contains some iterative algorithms such as k-means, most of its benchmarks are not iterative. Also, many of the available benchmarks are implemented on only a few frameworks, which makes it difficult to use this benchmarking suite to compare the performance of different distributed computing frameworks to each other.

2.3 Analyzing the characteristics of scientific computing algorithms

When adapting algorithms to distributed computing frameworks, it is important to know which algorithm characteristics can affect the efficiency of their parallel implementations. In this section we give an overview of studies which have worked on classifying scientific computing or large scale data processing application and algorithms based on their characteristics.

In 2006, the scientists from University of California, Berkeley studied the parallelization of scientific computing applications and published a technical report [61] which maps the landscape of parallel computing research at the time. Their most interesting contribution in the context of this work was the classification of parallel applications into a number of so called dwarfs. These dwarfs are described as algorithmic methods for capturing the patterns of computation together with communication.

1. **Dense Linear** – Vector-vector, matrix-vector and matrix-matrix operations (Sum, multiplication, dot–product, etc.) on dense matrices.

2. **Sparse Linear Algebra** – Matrix and vector operations on sparse matrices. Most of the values in sparse matrices or vectors are zeros. Data is no longer stored in matrix like structure to avoid having to store all zeros and thus needs different types of methods to perform typical matrix operations.

3. **Spectral Methods** – A number of methods for solving differential equations, where the data is usually from the frequency domain.

4. **N-Body Methods** Collection of methods solving the problem of predicting the movement of objects that are affected by the gravity of other moving objects.

5. **Structured Grids** – Collections of methods that work on a structured grid of data points, where the distance between data points are kept constant. All data–point values are updated together when a

function is applied on data and the resulting value of data–points typically depend on its own value and the values of its spatially located neighbor data points.

6. **Unstructured Grids** – Collections of methods that work on a unstructured grid of data points. All data–point values are updated together when a function is applied on data and the resulting value of data–points typically depend on its own value and the values of its spatially located neighboring data points. In comparison to structured grids, distances between data points can vary greatly, and the shape of the grid is not predefined.
7. **Monte Carlo** – Computationally heavy and embarrassingly parallel methods for estimating the value of numerical functions using repeated randomized sampling.
8. **Combinational Logic** – Class of functions implemented using only boolean circuits. Computations are stateless, meaning no previous results can be recalled.
9. **Graph traversal** – Graph computing methods that require moving through the graph structure using verities and edges when performing computations. The actual computations are not intensive. It may require large number of data synchronization steps when directly parallelizing such methods using data parallel approach. Especially when the movement is spontaneous.
10. **Dynamic Programming** – Method for solving problems by dividing it into a set of smaller problems and avoiding re-computations of smaller problems by only computing them once and memorizing their results.
11. **Backtrack and Branch+Bound** – Recursive methods for finding optimal solutions for search or global optimization problems. They divide the search space into smaller areas, find local optimum results and combine the recursively found optimum results into a global optimal one.
12. **Construct Graphical Models** – Methods like Bayesian networks and hidden Markov models which model the uncertainty and com-

plexity of systems by constructing a graph of all possible states with probabilistic transitions between such states.

13. **Finite State Machine** – Methods where the applied function is defined as computational machine consisting of finite number of possible states and transitions for moving between such states depending on input values.

First seven of these dwarfs (Dense Linear Algebra, Sparse Linear Algebra, Spectral Methods, N-Body Methods, Structured Grids, Unstructured Grids, Monte Carlo) are based on widely used numerical methods in scientific computing. Last 6 dwarfs (Combinatorial Logic, Graph Traversal, Dynamic Programming, Backtrack and Branch+Bound, Construct Graphical Models and Finite State Machine) were added to broaden the distributed computing algorithm classification with a more widely applicable parallel algorithm types.

Authors of this work provided a very insightful overview of algorithms and programming models used in the parallel computing field but they did not investigate the characteristics of algorithm classes in more detail.

In 2014, Berkeley classification was extended by scientists from Rutgers University and Indiana University to also cover more data intensive applications by introducing Big Data Ogres. [62] Their goal was to compare the two extremes of parallel computing field: data intensive Big Data processing and computation intensive scientific computing applications. They studied and analyzed 51 typical use cases that required processing Big Data and designed a multifaceted classification for such applications.

They propose a set of Ogres - a classification of Data intensive core analytics, kernels or skeletons and characterized them from four different viewpoints (so called Ogre facets).

The first Ogre facet is the architecture of the problem after parallelization. It consists of pleasingly parallel (Blast, Protein docking, imagery, Monte Carlo methods), Local Machine Learning (ML or filtering pleasingly parallel as in biological imagery, radar), Global Machine Learning (Latent Dirichlet Allocation (LDA), Clustering etc. with parallel ML over nodes of system) and fusion (Combination of other architectures).

The second Ogre Facet captures the source of data between Structured Query Language (SQL), NOSQL based, other Enterprise data systems, set of files, Internet of Things, streaming and HPC simulations.

The third Ogre Facet is the distinctive system features such as Agents (for example in epidemiology or swarm approaches) and Geographical Information Systems.

The fourth Ogre Facet captures the style of Big Data applications such as are data points in metric or non-metric spaces, maximum Likelihood, F2 minimizations, and expectation Maximization.

The fifth Facet is Ogres themselves classifying core analytic kernels. These Ogres are:

1. **Recommender systems** – Goal is to predict user preference in items by analyzing behavior and pattens of many users. Main approaches are collaborative filtering & content-based filtering methods.

2. **Linear classifiers** – Machine learning approach where objects are classified by using a linear combination of their characteristics. For example support vector machines, naive Bayes and random forests.

3. **Outlier detection** – Methods for detection of outliers in data. Often dealing more with figuring out what to classify as outliers as there is no fully accepted mathematical definition of what counts as outliers.

4. **Clustering** – There are many different clustering methods, such as centroid based (k-means and k-medoid) density based (DBSCAN, OPTICS and Mean–shift) or hierarchical clustering. Many of them are of iterative nature.

5. **PageRank** – Well known ranking algorithm based on incoming links to web cites.

6. **Latent Dirichlet Allocation (LDA)** – It is a generative statistical method for modeling topics and their relations.

7. **Probabilistic latent semantic indexing (PLSI)** – Estimating the probability of object co–occurrence.

8. **Singular Value Decomposition (SVD)** – A method for the factorization of matrices.

9. **Multidimensional Scaling** – Information visualization technique for illustrating the distances of objects in N–dimensional space for human eyes.

10. **Graph algorithms** – Consists of many different graph processing methods, which include finding communities, subgraphs or motifs, diameter, maximal cliques, connected components, etc.

11. **Neural networks** – Family of computational methods inspired from neural networks found in nature, such as nervous systems and brains of animals. Recently more known as deep–learning.

12. **Global optimization** – Methods for finding the global maximum or minimum value of functions over all possible input values. The most well known problem of this type is traveling salesman problem.

13. **Agents** – Computational methods for simulating the behavior of a community of individuals. Often used in epidemiology, social sciences and financial simulation.

14. **Geographical Information Systems** – Many methods dealing with geographical data with the aim to analyze, manipulate and visualize such data.

They propose that these core analytic kernels or Ogres can serve as standard benchmarks for evaluating applications from these two paradigms.

Their conclusion was that the two paradigms: (i) scientific computing – having to deal with many more data intensive workflows; and (ii) Big Data – trying to support more computation heavy tasks; are converging. Scientific computing application are moving towards processing larger and larger amount of data. Big Data applications are moving towards more complex and non-embarrassingly parallel algorithms.

They also conclude that the current benchmarks do not cover all the facets of these two paradigms and new benchmarks should be created to cover them. In addition, it is also needed to agree on the type of datasets with various sizes; correctness for each of the implementation and choosing between written and source code specifications for benchmarks.

2.4 Summary

While there are a large number of distributed computing frameworks to choose from, Hadoop MapReduce was the most widely used framework for dealing with large amount of data when this work was started. While it was clear that the MapReduce model was designed for less complex and easily parallelizable algorithms, there are many different types of scientific computing algorithms and it was important to know more precisely which types of scientific computing algorithms MapReduce is sufficient.

The related works include a number of studies which have investigated the performance of Hadoop MapReduce for both simple and more complex algorithms (including scientific computing algorithms), but there is no clear way of deciding whether Hadoop MapReduce framework or for the MapReduce model in general is suitable for a given algorithm. Different studies of classifying scientific computing algorithms help us to divide them into classes, but there is still no concrete methodology to follow for choosing the most suitable distributed computing framework for a given algorithm or class of algorithms.

The next chapter describes our work in investigating the performance of Hadoop MapReduce framework and classifying scientific computing algorithms based on how suitable Hadoop MapReduce is for parallelizing them.

CHAPTER 3

ADAPTING SCIENTIFIC COMPUTING ALGORITHMS TO MAPREDUCE

This chapter describes the work of adapting scientific computing algorithms to the MapReduce model. While there are a large number of distributed computing frameworks to choose from (as shown in the state of the art chapter) Hadoop MapReduce [17] was the most widely used large scale distributed computing framework when this work was started.

While it was clear from the start that the MapReduce model was designed for less complex and easily parallelizable algorithms, there are many different types of distributed computing algorithms it could still be very useful for. There are a number of related studies that have investigated the performance of Hadoop MapReduce for both simple and more complex algorithms (including scientific computing algorithms), but there is no straightforward method for identifying whether the Hadoop MapReduce framework is suitable for a given algorithm.

For these reasons, we decided to study [2] Hadoop MapReduce in more detail and to try to create a classification for deciding the suitability of the MapReduce model for different scientific computing algorithms. We studied number of typical scientific computing algorithms and adapted them to the MapReduce model to investigate what affects their efficiency and scalability.

The algorithm classes were designed based on the difficulty of adapting scientific computing algorithms to the MapReduce model and what is the resulting MapReduce job structure. The characteristic that clearly had the most effect was identified as the number of iterations that need to be executed. The division of algorithms into different classes is as follows:

1. Algorithms that require a single execution of a MapReduce job.
2. Algorithms that require a sequential execution of a constant number of MapReduce jobs.
3. Iterative algorithms where each iteration consists of a single MapReduce job.
4. Iterative algorithms where each iteration consists of multiple MapReduce jobs.

We chose one algorithm from each of these classes to illustrate our classification approach. These algorithms were factoring integers (first class), Clustering Large Applications (CLARA, second class), Partitioning Around Medoids (PAM third class) and Conjugate Gradient (CG, fourth class). Each of these algorithms are described in the following section together with how they were adapted to the MapReduce model and what was their relative parallel speedup when executed in Hadoop MapReduce cluster.

To measure the performance of the implemented algorithms we set up a MapReduce cluster. The Hadoop cluster was created in a local University of Tartu Cloud built on top of Eucalyptus cloud computing platform. It was composed of one master and sixteen slave nodes. Only the slaves acted as MapReduce task nodes, resulting in 16 parallel workers where the MapReduce tasks could be executed on. Each node is a virtual machine with 2.2 GHz CPU, 500 MB RAM and 10 GB disk space allocated for the HDFS, making the total size of the HDFS 160 GB.

3.1 Algorithms

This section contains the description of the benchmarking algorithms that were used to investigate the parallel efficiency of scientific computing algorithms when adapted to Hadoop MapReduce. In addition to being from a separate algorithm class based on our initial classification, each of these algorithms is also different based on a number of other parallelization characteristics, such as the required number of iterations, size of the input and intermediary data that needs to be kept in memory and the types of communication patterns that need to be applied to synchronize data. These algorithms are described in the reverse order to the previously shown classification, starting with an example for the fourth class.

3.1.1 Conjugate Gradient linear system solver (CG)

Solving systems of linear algebraic equations (SLAE) is a problem often encountered in the fields like engineering, physics, chemistry, computer science or economics. The main goal is different in each of these fields, but the main challenge stays the same. How to efficiently solve systems of linear equations with a huge number of unknowns? For extremely large matrices, it is often unfeasible to find an exact solution for the system of linear equations, either because of time or resource constraints, and an approximation of the solution vector x is found instead.

Conjugate Gradient [63] is an iterative algorithm for solving algebraic systems of linear equations. It solves linear systems using matrix and vector operations. Linear system is first transferred into the matrix form:

$$Ax = b \tag{3.1}$$

where A is a matrix consisting of the coefficients $a_{11}, a_{12}, ..., a_{mn}$ of the system, b is a known vector consisting of constant terms of the system $b_1, b_2, ..., b_m$ and x is the solution vector, made up of the unknowns of the system $x_1, x_2, ..., x_n$.

CG then performs an initial inaccurate guess of the solution x and then iteratively improves its accuracy by applying gradient descent by using the matrix A and vector b values to find the approximate vector x values, if a solution exists at all. The accuracy of the Conjugate Gradient result depends on the number of iterations that are executed.

The input matrices are generally large, but can typically fit into collective memory of computer clusters. The computational complexity of the algorithm is not high and the performed task at every iteration is relatively small which means the ratio between communication and computation is unusually high, especially in comparison to CLARA and PAM. This makes CG a good candidate as an additional benchmarking algorithm for iterative MapReduce frameworks.

Unknowns	24	500	1000	2000	4000	6000	8000
1 node	259	261	327	687	1938	3810	7619
2 nodes	255	259	298	507	1268	2495	4185
4 nodes	255	236	281	360	721	1374	2193
8 nodes	251	251	291	397	563	824	1246
16 nodes	236	240	278	297	338	511	809

Table 3.1: Run times for the CG implementation in MapReduce under varying cluster size. [2]

Adapting CG to MapReduce is relatively complex task as it is not possible to directly adapt the whole algorithm to the MapReduce model. The matrix and vector operations used by CG at each iterations can be reduced to the MapReduce model instead. Every time one of these operations is used in the CG, a new MapReduce job is executed. As a result, multiple MapReduce jobs are executed at every iteration. This is not efficient as it takes time for the Hadoop framework to schedule, start up and finish MapReduce jobs. It can be viewed as MapReduce job latency and executing multiple jobs at each iteration adds up to a significant overhead.

Additionally, in Hadoop, the matrix A is stored on the HDFS and is used as an input for the matrix-vector multiplication operation at every iteration. In Hadoop MapReduce framework it is not possible to cache the input between different executions of MapReduce jobs, so every time this operation is executed, the input must be read again from the file system. As the matrix A values never change between iterations, the exact same work is repeated at every iteration. This adds up to a significant additional overhead.

Experiments were run with different number of parallel nodes to be able to calculate relative parallel speedup. Relative parallel speedup measures how many times the parallel execution is faster than running the same MapReduce algorithm on single node. If it is larger than 1, it means there is at least some gain from doing the work in parallel. Speedup which is equal to the number of nodes is considered ideal and means that the algorithm has a perfect scalability.

We also generated different sized input matrices to investigate the effect of the problem size on the performance. While CG is typically used for larger sparse matrices mostly consisting of zeros, we used dense matrices

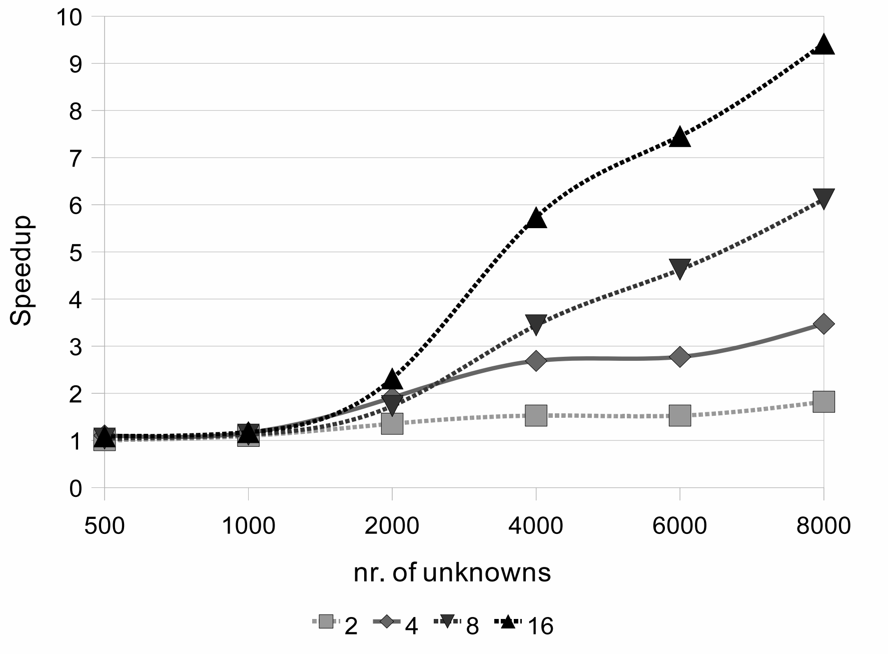

Figure 3.1: Speedup for Conjugate Gradient algorithm with different number of nodes. [2]

to simplify the benchmarking process. Run times for the CG algorithm are shown in Table 3.1 and calculated speedup is shown on Figure 3.1.

It took 220 seconds to solve a system with only 24 unknowns in a 16 node cluster, which is definitely very slow for solving a linear system with such a small number of calculations needed. It indicates that most of the time is spent on the background tasks and not on the actual calculations.

CG MapReduce algorithm is able to achieve much better parallel speed-up when solving larger linear systems. It took almost 2 hours to solve a linear system with 8000 unknowns on one node and 809 seconds on 16 nodes. While these results show that MapReduce is able to take advantage of additional computing resources, it does not mean that the time is efficiently spent on actual computations. To investigate this further, we decided to also adapt CG to Twister, which is a MapReduce framework that is designed for iterative applications and has previously been described in in section 2.1.1. Results for these experiments are provided in section 3.2.

3.1.2 Partitioning Around Medoids (PAM)

Partitioning Around Medoids [64] (PAM) is the naive version of the k-medoid clustering method. The general idea of the k-medoid clustering method is to represent each cluster with its most central element, the medoid, and to reduce all comparisons between the clusters and other objects into comparisons between the medoids of the clusters and the objects.

To cluster a set of objects into k clusters, PAM first chooses random objects as the initial medoids. For each object in the dataset, it calculates the distance from every medoid and assigns the object to the closest medoid, dividing the dataset into k clusters. At the next step, the medoid positions are recalculated for each of the clusters, choosing the most central object as the new medoid. This two step process is repeated until there is no change from the previous iteration. The whole iteration can be adapted as a single MapReduce job:

- Map:
  - Find the closest medoid and assign the object to it.
  - Input: (cluster id, object)
  - Output: (new cluster id, object)
- Reduce:
  - Find which object is the most central and assign it as the new medoid of the cluster.
  - Input: (cluster id, (list of all objects in the cluster))
  - Output: (cluster id, new medoid)

The resulting MapReduce job is repeated until medoid positions of the clusters no longer change.

Similarly to Conjugate Gradient, PAM also has issues with job latency and rereading the input from the file system at every iteration because a new MapReduce job is executed at each time. Table 3.2 provides the runtimes and Figure 3.2 provides the calculated speedup for the adapted PAM algorithm. These results show reasonable parallel speedups, but the runtime is very long, it took almost two hours to cluster 100000 object on a single node. To investigate how much time is actually spent on computations we also implemented PAM using Twister. Results for these experiments are described in Section 3.2.

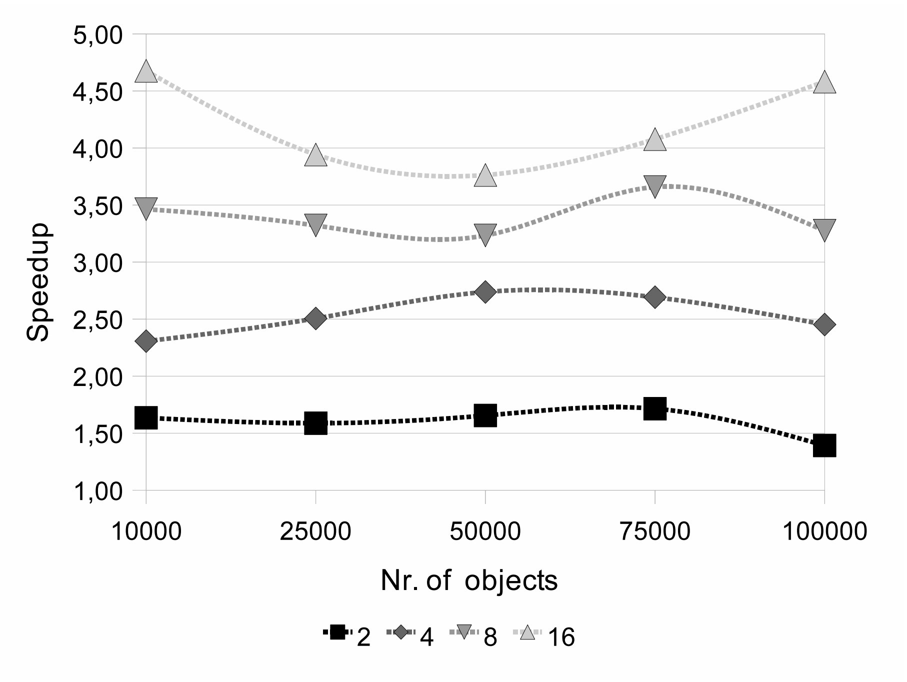

Figure 3.2: Parallel speedup for PAM with different number of nodes. [2]

3.1.3 Clustering Large Applications (CLARA)

Clustering Large Applications [64] (CLARA) is also an iterative k-medoid clustering algorithm, but in contrast to PAM, it only clusters small random subsets of the dataset to find candidate medoids for the whole dataset. This process is repeated multiple times and the best set of candidate medoids is chosen as the final result.

To adapt CLARA to MapReduce, we first have to investigate how to divide the work into parallel tasks that can be executed in concurrent map or reduce tasks. Sampling a dataset S times and using the PAM algorithm on each of the samples to cluster them can be divided into S different tasks, because these results are completely independent of each other and do not have to be executed in a sequence.

The process of evaluating the candidate medoids (obtained by clustering each of the samples using PAM) on the whole data set can be divided into any number of concurrent tasks by simply dividing the data between the tasks. And then check each object one at a time, find the distance from its closest medoid and repeat this for each candidate set of medoids. As a

Objects	10000	25000	50000	75000	100000
1 node	1389	1347	2014	3620	6959
2 nodes	1133	1697	1826	2011	6130
4 nodes	803	782	1156	2562	2563
8 nodes	635	627	1513	1084	1851
16 nodes	297	497	432	761	1029

Table 3.2: Run times for the PAM algorithm. [2]

next step, we can simply group the distances by the candidate medoid sets and calculate the sum of distances for each of them.

As a result, it is possible to ignore the iterative structure of the original CLARA algorithm. Everything can be reduced into two MapReduce jobs, both executing different tasks. First job chooses a number of random subsets from the input data sets, clusters each of them concurrently using PAM, and outputs the results. Structure for the first CLARA MapReduce job would then be:

- Map:
  - Assign a random key to each object.
  - Input: (key, object)
  - Output: (random key, object)
- Reduce:
  - The order of the objects is random after sorting. Read first n objects and perform PAM clustering on the n objects to find k different candidate medoids.
  - Input: (key, list of objects)
  - Output: (key, list of k medoids)

The second MapReduce job calculates the quality measure for each of the results of the first job, by checking them on the whole data set concurrently inside one MapReduce job. As a result of having only two MapReduce jobs, the job latency stays minimal and the input data set is only read twice.

Objects (thousands)	25	50	100	500	1000	5000	10000
1 node	117	118	125	183	261	819	1517
2 nodes	79	84	89	150	215	476	832
4 nodes	61	66	72	120	127	316	486
8 nodes	52	56	61	114	124	218	320
16 nodes	44	50	58	99	98	104	156

Table 3.3: Run times for the CLARA algorithm. [2]

The second CLARA MapReduce job consists of the remainder of the original CLARA algorithm and its structure is the following:

- Map:
  - For each object, calculate the distance from the closest medoid. This is calculated for each candidate sets, and one output is generated for each of them.
  - Input: (cluster, object)
  - Output: (candidate set id, distance from the closest medoid) [One output for each candidate set]
- Reduce:
  - Sum the distances with the same candidate set id.
  - Input: (candidate set id, list of distances)
  - Output: (candidate set id, sum(list of distances))

The result of the second job is a list of calculated sums, each representing the total sum of distances from all objects and their closest medoids, one for each candidate set. The candidate set of medoids with the smallest sum of distances between objects and their closest medoids is chosen as the best clustering.

From the experiment results (Tables 3.2, 3.3 and Figures 3.2, 3.3) it is possible to see that CLARA MapReduce algorithm works much faster than PAM, especially when the number of objects in the dataset increases. PAM was not able to handle datasets larger than 100 000 objects while CLARA could cluster datasets consisting of millions or even tens of millions of objects. It should also be noted that the time to cluster the smallest

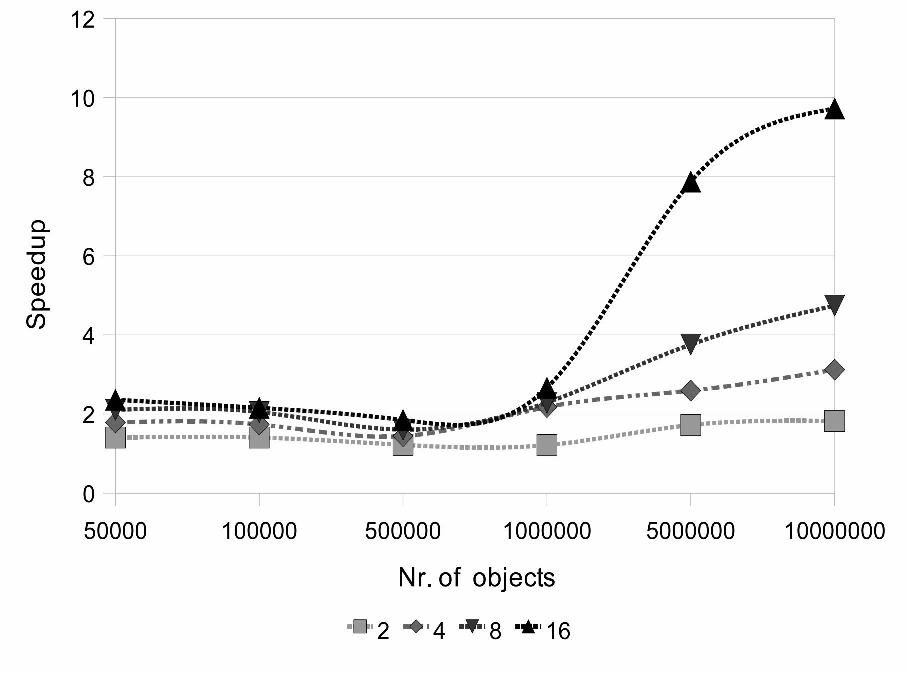

Figure 3.3: Parallel speedup for the CLARA algorithm with different number of nodes. [2]

dataset is quite large for both CLARA and PAM. This is because the background tasks of MapReduce framework are relatively slow to start, so each separate MapReduce job that is started slows down the algorithm. This affects PAM more greatly than CLARA because PAM consists of many MapReduce job iterations while in CLARA only uses two MapReduce jobs.

3.1.4 Factoring integers

Factoring integers is a method for dividing an integer into a set of prime numbers that make up the original number by multiplying them all. For example the factors of a number 21 are 3 and 7. Factoring integers is used for example to break RSA cryptographic system by calculating the secret key value based on the public key.

In this case we chose the most basic method of factoring integers, the trial division. This method is not used in practice, as there exist much faster methods like general number field sieve [65]. But we chose this method purely to illustrate adapting an embarrassingly parallel problem, belonging to the first class, to the MapReduce model as comparison to the other three algorithms.

To factor a number using trial division, all possible factors of the number are checked to see if they divide the number evenly. If one of them does, then it is a factor. This can be adopted to the MapReduce model, by dividing all possible factors into multiple subgroups and checking each of them in a separate map or reduce task concurrently:

- Map:
  - Gets a number to be factored as an input, finds the square root of the number and divides the range from 2 to $\sqrt{number}$ into n smaller ranges, and outputs each of them.
  - Input: (key, number)
  - Output: (id, (start, end, number)) [one output for each range, n total]
- Reduce:
  - Gets a number and a range, in where to check for factors, as an input and finds if any of the numbers in this range divide the number evenly.
  - Input: (id, (start, end, number))
  - Output: (id, factor)

As a result, in contrast to the previous algorithms, this algorithm is reduced to a single MapReduce job, meaning there is no overhead from executing multiple jobs in sequence and why this algorithm belongs to the first algorithm class.

The run times for the integer factorization are given on the Table 3.4 and speedup is shown on Figure 3.4. From the Figure 3.4 it is possible to see that when the factored number is small, there is only little advantage of using multiple workers in parallel. The speedup is slightly above 1 for 2 node cluster and only reaches 2.22 in 16 node cluster. This is because the number of calculations done was relatively small compared to the background tasks of the framework. However, with the increase of the size of the input number, the speedup started to grow significantly. With the input

Digits	17	18	19	20	21
1 node	51	142	361	2058	6767
2 nodes	37	67	188	1117	3271
4 nodes	30	50	120	512	1622
8 nodes	27	36	70	299	887
16 nodes	27	38	59	215	566

Table 3.4: Run times for the integer factorization algorithm. [2]

size of 21 digits, the speedup for two and four node executions was 2.07 and 4.17, showing that there is an ideal gain from using multiple nodes to find the factors when the size of the input is large enough. With larger number of nodes the speedup does not reach the number of nodes, indicating that calculations were not long enough to get full benefit from using 16 nodes. The calculated speedup numbers suggest that this algorithm has a good scalability and that Hadoop MapReduce framework is very suitable for algorithms belonging to the first class.

3.2 Additional Twister experiments

While CG and PAM were able to achieve a reasonable speedup when running on Hadoop MapReduce, it was clear that MapReduce framework introduced significant overhead and slowed down the whole process. To be able to estimate how much it was slowed we decided to compare the performance to other parallel programming solutions (either distributed computing frameworks or parallel programming libraries).

We chose Twister [20] for this task because it is advertised as an iterative MapReduce framework and thus should provide a good comparison to Hadoop MapReduce for iterative algorithms. We implemented CG and PAM for Twister, set up a Twister MapReduce cluster with the same virtual hardware configuration as the Hadoop cluster and ran benchmarking experiments. [2]

Comparing the Twister (Tables 3.5 and 3.6) and Hadoop (Tables 3.1 and 3.2) run times for these algorithms clearly shows that Twister is much more efficient. Twister can solve larger problems in less time and for the same size problems it is 50 to 100 times faster than Hadoop, when running

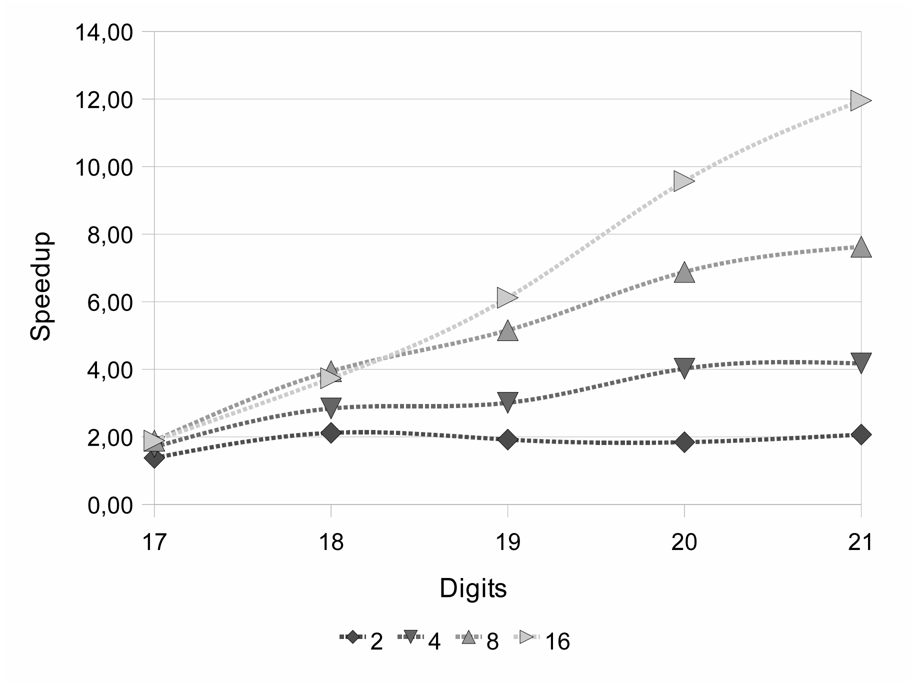

Figure 3.4: Parallel speedup for the integer factorization with different number of nodes. [2]

on 16 nodes. The algorithm structure, how it is adapted to the MapReduce model stays exactly the same in both Twister and Hadoop, yet Twister is much more efficient in handling background tasks for iterative MapReduce applications.

The Hadoop job latency was around 19 to 20 seconds per iteration, while in Twister it is below 3 seconds regardless of the number of iterations. Twister also stores the bulk of the input to the memory and does not need to read it again from the file system at every iteration. In CG, it means being able to store the whole matrix into the collective memory of the cluster and in PAM it means being able to store all the clustered objects.

These results clearly confirm that Hadoop MapReduce is not able to handle iterative algorithms well.

Unknowns	500	1000	2000	4000	6000	8000	10000	20000
1 node	3.19	3.40	3.00	4.96	7.69	11.27	16.22	56.01
2 nodes	3.33	3.40	2.82	3.99	5.72	6.98	9.51	28.15
4 nodes	3.27	3.29	2.76	3.54	4.03	5.29	6.54	16.33
8 nodes	3.38	3.30	2.81	3.56	3.79	4.76	5.44	14.75
16 nodes	3.40	3.42	2.75	3.50	3.56	4.11	4.86	10.05

Table 3.5: Run times for the CG implementation in Twister [2]

Objects	10000	25000	50000	75000	100000	200000	300000
1 node	5.45	20.55	25.00	96.61	204.55	638.56	1888.71
2 nodes	2.93	10.06	22.85	51.19	93.06	359.88	808.96
4 nodes	3.91	7.99	14.63	15.51	91.78	197.15	343.64
8 nodes	4.04	4.93	15.11	31.84	38.13	131.41	355.77
16 nodes	4.25	6.63	11.55	22.26	24.87	85.76	237.43

Table 3.6: Run times for the PAM algorithm in Twister [2]

3.3 Discussion

The results [2] confirm that Hadoop performs very well for more simple algorithms that make up the first class, generally described as embarrassingly parallel algorithms, and also for algorithms belonging to second class, when the number of MapReduce jobs is small. However, it performed much worse for the more complex iterative algorithms belonging to the third and fourth class, which require synchronization sub-steps at every iteration.

Comparing Twister and Hadoop for algorithms belonging to the third and fourth class has shown that Twister is much more suitable for these classes. At the same time, Hadoop may be more suitable for the first class of algorithms, thanks to the fault tolerance it provides, and also for data intensive algorithms in general, when Twister has problems fitting the data into the the collective memory of the cluster.

For less data intensive algorithms belonging to the second class the results are not so clear. When the number of different MapReduce executions is not large, Hadoop can perform well and given the fault tolerance, it should be considered to be more suitable. But because of the short running tasks in Hadoop, which are terminated each time one MapReduce cycle is over, Hadoop loses its efficiency as the number of MapReduce executions increase and Twister should be preferred instead. Thus, the choice of the framework for the second class of algorithms strongly depends on the number of MapReduce steps needed and on how data intensive the task is.

However, Twister also has certain limitations for distributed applications. Significant advantage in using Twister comes from its ability to keep static input data in memory across iterations. But it means that this static data must fit into the collective memory of the machines Twister is configured on. For data intensive tasks this may be quite an unreasonable demand. For example, processing 1 TB of data with Twister would require more than 128 machines with 8 GB of memory each just to store the data into the memory, not to mention the memory needed for the rest of the application, framework itself and operating system it runs in.

Twister also does not have a proper fault tolerance when compared to the fault tolerance provided by Hadoop, which can be a very serious problem when running twister on a public cloud where machines are prone to relatively frequent failures.

The Hadoop MapReduce problems with iterative algorithm arise from the fact that the map and reduce tasks are defined stateless in the MapReduce model and the data flow is strictly one directional. In Hadoop, each new map and reduce task pair requires a separate MapReduce job and there are no means to keep the state of the tasks in memory across multiple sequential MapReduce jobs. As a result, an iterative application consisting of 10 MapReduce jobs must be configured 10 times, read input from the HDFS 10 times, send intermediate data to Reduce 10 times and write any changes back to the HDFS.

We measured the job configuring time to be at minimum of 17 seconds for a typical MapReduce job, regardless of the amount of computations involved. The extra HDFS operations that are required for iterative algorithms also result in additional overhead from the slower I/O operations. We decided to propose and investigate several possible solutions for algorithms for which Hadoop MapReduce framework is not suitable.

One approach is to use alternative embarrassingly parallel algorithms instead of the most efficient iterative algorithms and adapt those to MapReduce instead [4]. For example, instead of using Conjugate Gradient linear system solver, it is possible to use Monte Carlo method for finding the inverse of the linear system‘s matrix form to solve it. This approach is discussed in detail in chapter 4.

Another approach is to use MapReduce frameworks that are specifically designed to provide better support for iterative applications, like

Twister [34], Spark [33] and HaLoop [21]. However, while they are more suitable for complex algorithms, we have also seen in on our previous work [66] that they give up some of the advantages (like fault tolerance as discussed in section 3.2) of MapReduce which are necessary for long running scientific applications. The second approach is discussed in chapter 5.

A third approach is to use an alternative distributed computing model, which are more suitable for large scale iterative scientific applications, and at the same time provide the same advantages as MapReduce. It should take care of most of the parallelization tasks and simplify the creation of distributed applications for users who are not experts in parallel programming. The distributed model that caught our attention is Bulk Synchronous Parallel (BSP) [24] and we decided to evaluate whether the frameworks based on this model are suitable for complex scientific algorithms. The third approach is discussed in chapter 6.

3.4 Summary

In this chapter we investigated the performance of typical scientific computing algorithms when adapted to the MapReduce model and executed in Hadoop MapReduce framework. The results of the study described in this chapter have previously been published in [2]. We documented a number of issues Hadoop MapReduce and the MapReduce model in general has with more complex algorithms and used this result to create a classification of scientific algorithms for the MapReduce model.

While Hadoop MapReduce has definite advantages that are beneficial for scientific computing algorithms it can not manage iterative algorithms well and we proposed three different approaches to deal with such algorithms. The following three chapters describe our work in investigating each of these approaches in detail.

CHAPTER 4

RESTRUCTURING SCIENTIFIC COMPUTING ALGORITHMS FOR MAPREDUCE

One approach for solving the problems MapReduce has with iterative algorithms is to try to reduce the number of iterations, or to find alternative algorithms which perform the same task, but which are more easily parallelizable. Such alternative algorithms might have lower performance in comparison to the iterative algorithms, but it could result in a much better performance in a MapReduce cluster if high parallel speedup can be achieved.

One example of adapting an alternative algorithm for MapReduce is directly using CLARA (Clustering Large Applications) instead of PAM (Partitioning Around Medoids) when clustering objects. It was evident from the previous chapter that the Hadoop MapReduce distributed computing framework is more suitable for CLARA than PAM. However, its clustering accuracy is not guaranteed because it uses random sampling. We decided to investigate this scenario further by implementing CLARA in MPI and run [3] additional experiments to analyze its performance in more detail and to investigate how usable this approach is in general.

In addition, Conjugate Gradient (CG) was the algorithm with which we have had most performance issues when adapting it to the MapReduce model. We decided to look for alternative algorithms for solving systems of linear algebraic equations (SLAE) which would be more easily parallelizable.

While there are many different methods for solving SLAE's, we found that there exists a Monte Carlo method for finding the inverse of the matrix form of a linear system [67], which can be used to solve the original linear system. We decided to adapt [4] the Monte Carlo based algorithm for solving systems of linear algebraic equation to MapReduce to investigate how it performs in comparison to CG when adapted to MapReduce and how applicable can similar approaches be in general.

4.1 Medoid based clustering of objects

We were interested in investigating non-embarrassingly parallel algorithms for which the MapReduce model is suitable and which would not be so easy to parallelize. One of the algorithms we reduced to the MapReduce model that matched this description was CLARA (Clustering LARge Applications) [64] k-medoid clustering algorithm. While the nature of CLARA is a non-embarrassingly parallel algorithm that is relatively complex, it was adaptable to the MapReduce model with little effect on the efficiency and scalability of the result. While it required the restructuring of the algorithm into several stages to change the iterative nature of the algorithm, it did not require changing the core of the algorithm and thus it was relatively easy to utilize MapReduce to achieve parallelism for CLARA. How exactly to adapt CLARA to MapReduce is already described in detail in section 3.1.3.

4.1.1 Analyzing CLARA MapReduce

We set up a 17 node cluster in our SciCloud testing environment to measure the parallel efficiency and scalability of the CLARA MapReduce implementation. Apart from Hadoop we also used Message Passing Interface (MPI) [11] to parallelize and implement the CLARA algorithm in Python to investigate it's performance in comparison to another parallel programming solution.

The cluster is composed of one master and sixteen slave nodes. Each node is a virtual machine with 2.2 GHz CPU, 500 MB RAM and 10 GB disk space allocated for the HDFS. For MapReduce, only the slaves act as task nodes, resulting in 16 parallel workers where the MapReduce tasks can be executed on. For MPI, we used 16 of the 17 available machines. For

Objects (10^3)	25	100	500	1 000	5 000	10 000
1 node	40	62	179	330	1537	3061
2 nodes	20	33	93	171	949	1582
4 nodes	11	17	48	87	398	784
8 nodes	6	9	25	44	202	397
16 nodes	4	5	13	23	107	206

Table 4.1: Run times (sec) for the CLARA using MPI. [3]

MPI, we used *mpi4py* python package, which provides a Python interface to the MPI standard that can work with any native MPI library written in C or C++. For the native MPI library we used OpenMPI [46]. For Hadoop we used version 0.22.0.

We ran the tests with 1, 2, 4, 8 and 16 nodes with a varying size of the input data, which were 25 000, 100 000, 500 000, 1 000 000, 5 000 000 and 10 000 000 objects. The data sets were generated randomly and were clustered into 16 separate clusters. The number of samples was 24 and the sample size was 240. The measured runtimes are shown in the Table 3.3 in section 3.1.3.

These results show that for smaller datasets the MPI implementation is performing better than Hadoop, but as the size of the input increases, Hadoop starts to perform faster. When clustering 5 million objects, MPI implementation is clearly slower than Hadoop and even more so when clustering 10 million objects. However, the tests which are run using 16 nodes are showing less difference and to investigate it, we decided to calculate the parallel speedup for both cases, to be able to see how well the different implementations scale in comparison to each other.

Speedup comparison for both MapReduce and MPI implementations are illustrated on the Figure 4.1. MPI implementation has better speedup numbers than MapReduce, showing that while MPI implementation is slower in this case, it is able to use the resources of additional nodes more efficiently. One of the main reasons why MapReduce shows worse speedup numbers is the job latency.

As discussed in section 3.3, the MapReduce job latency is one of the main problems MapReduce has with iterative algorithms, because it slows down the execution times for each iteration. Job latency when running MapReduce in a cloud environment with multiple nodes has been mea-

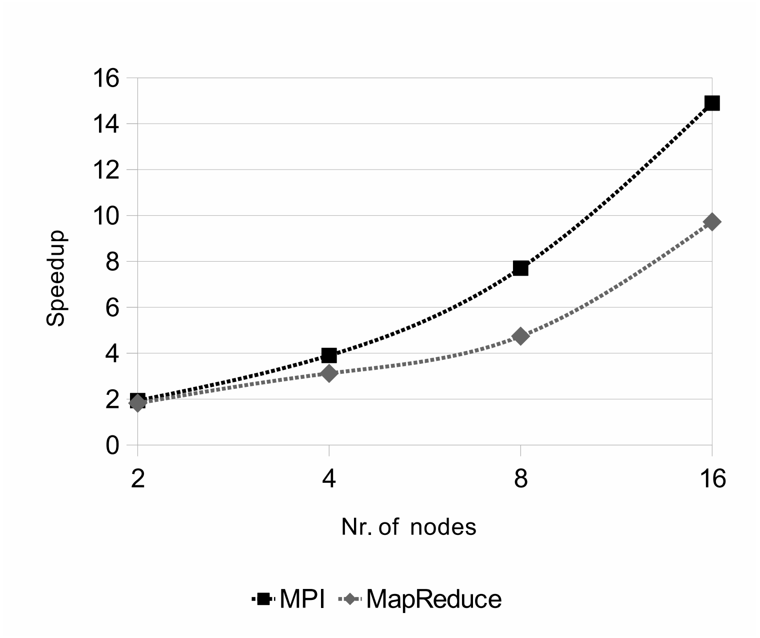

Figure 4.1: Parallel speedup for the CLARA algorithm, comparing MapReduce and MPI. [3]

sured to be at least 15 seconds in our tests, regardless of the input size. For CLARA algorithms, which uses two separate MapReduce jobs that are executed in sequence, this adds at least 30 seconds to the runtime, 15 seconds for each of the executions, which directly affects the scalability of this algorithm.

At the same time, these results show that MapReduce can handle larger amount of objects better, and while the job latency affects the scalability of the MapReduce CLARA implementation the effect is not large thanks to only needing to execute two separate MapReduce jobs. Also, this effect decreases as the input size grows, because the ratio between the job lag and the whole runtime also decreases.

Because the implementations are written in two different programming languages and the general program efficiency is quite different in Python and Java, it is hard to directly compare the run-times of the two different implementations. But, it still provides a useful comparison to judge the scalability of the two different implementations.

However, this does not prove MapReduce is faster than MPI in general. Domain specific solutions, specifically tailored for the problem/application can always be more efficient, but at least the automatically parallelizable framework Hadoop can easily compete with scripting languages like python which use MPI for data distribution and task parallelization.

4.2 Solving systems of linear equations

In section 3.1.1 we introduced Conjugate Gradient (CG) algorithm and adapted it to the MapReduce model. Using the automatic parallelization of the open source Hadoop MapReduce framework to scale the execution of the algorithm on cloud infrastructure did not gave satisfying results [2]. The relative complexity and the iterative structure of the CG algorithm makes it unsuited for Hadoop, which is designed for embarrassingly parallel (i.e. when its trivial to divide the problem into a number of concurrent tasks) data intensive tasks.

As described in previous chapter, CG belongs to the class 4 of algorithms. Adapting CG to Hadoop MapReduce involves adapting each matrix and vector operations to MapReduce separately. As a result, multiple Hadoop MapReduce jobs are executed at every CG iteration. It means that with increasing number of CG iterations, the implementation becomes very inefficient, when taking account of the problems that Hadoop has with chaining MapReduce jobs.

Some of the most widely used embarrassingly parallel algorithms are algorithms based on the Monte Carlo method. Such algorithms are not only almost trivial to parallelize, they are also very easily adaptable to MapReduce model and have been shown [19] to achieve a very good efficiency and scalability when executed in Hadoop framework.

We were interested to know how this approach would compare to both Hadoop MapReduce and Twister CG implementations. For this reason, we decided to study [4] Monte Carlo based linear system solvers, adapt them to the MapReduce model, and compare the resulting parallel efficiency and scalability to the CG implementation.

Branford et al. [67] describe a Monte Carlo based linear system solver algorithm for diagonally dominant matrices. We adapted their proposed algorithm to Hadoop MapReduce and compared its performance to our CG MapReduce algorithm. During the process, we also introduced additional improvements for handling memory usage and lowering the run time. The following section introduces the Monte Carlo linear system solver algorithm for solving linear systems and how it was restructured in the process of adapting it to the MapReduce model.

4.2.1 Monte Carlo SLAE algorithm

The following algorithm uses the matrix inverse method for solving SLAE. Given SLAE as a matrix equation (Eq. 3.1), where A is a square matrix of size n, the solution vector can be found as $x = A^{-1}b$, since we can transform equation 3.1 to $A^{-1}Ax = A^{-1}b$, where $A^{-1}A = I_n$, hence x can be found, should we manage to compute the inverse of A.

The algorithm for finding the matrix inverse presented herein is based on the one described and used in [67, 68]. As with the referenced works we consider only diagonally dominant matrices and use the following algorithm for computing the matrix inverse (Algorithm 1).

Stochastic error of the algorithm

Parameter ϵ influences the number of Markov Chains N, which we compute as shown in [67], the second parameter δ determines their length as shown in step 4 of algorithm 1.
N is computed using the Formula 4.1

$$N = \left(\frac{0.6745}{\epsilon(1 - ||C||)}\right)^2, \tag{4.1}$$

where $||C||$ is the spectral norm of C, ϵ is the measure of accuracy when solving SLAE and has the most effect of the running time of the algorithm as its complexity is $O(N\tau)$ for a single element or row of the solution matrix, where τ, being the chain's length, is generally much smaller. However, when trying to solve SLAE we wish to find all the elements of the solution vector as such the algorithm's complexity becomes dependent on the size of the matrix n. This makes parallelization all-important, since the rate of convergence is nowhere close to CG and, as the wanted accuracy increases, N value quickly becomes very large, as seen from Table 4.2, showcasing the best case scenario of smallest possible matrix norm.

However, the Monte Carlo approach has a unique advantage: it is possible to find a single element or a row of the matrix inverse, allowing to find a specific element of the SLAE solution vector. This process can be parallelized by splitting the N chains needed for its computation among

Algorithm 1 Monte Carlo algorithm for inverse of diagonally dominant matrices [67, 68, 4]

```
Input: Diagonally dominant matrix $D$, parameters $\epsilon$ and $\delta$
Output: Estimate of $D^{-1}$ computed via Monte Carlo method
Step 1. Calculate intermediate matrices $B_1$ and $B_2$
 1: Split $D = B_1 - B_2$, where $B_1 = diag(B)$ and $B_2 = B_1 - D$
Step 2. Calculate matrix $C$ and $||C||$
 1: Compute matrix $C = B_1^{-1} B_2$
 2: Compute $||C||$ and the number of Markov Chains ↩
      $N = \left( \frac{0.6745}{\epsilon(1-||C||)} \right)^2$
Step 3. Calculate matrix $P$
 1: Compute the probability matrix $P$, where $p_{ij} = \frac{|a_{ij}|}{\sum_{k=1}^{n} |a_{ik}|}$
Step 4. Calculate matrix $M$, by MC on $C$ and $P$
 1: For $i = 1$ to $n$
   1.1: For $j = 1$ to $N$
     1.1.1: Initialize $j^{th}$ chain set sum vector ↩
          $SUM[k] = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$
     1.1.2: Set $u = 1$ and $point = i$
     Markov Chain MC Computation
     1.1.3: While $|u| > \delta$
       1.1.3.1: Select a $nextpoint$, based on the transition ↩
            probabilities in $P$
       1.1.3.2: Compute $u = u \frac{C[point][nextpoint]}{P[point][nextpoint]}$
       1.1.3.3: Set $SUM[nextpoint] = SUM[nextpoint] + u$ and ↩
            $point = nextpoint$
   1.2: Assign $M$ elements as $m_{ik} = \frac{SUM[k]}{N}$, $k = 1, 2, \ldots, n$
Step 5. Calculate $D^{-1}$
 1: Compute the MC inverse $D^{-1} = B_1^{-1} + M B_1^{-1}$
```

ϵ	N
1.00E-02	4.55E+03
1.00E-04	4.55E+07
1.00E-06	4.55E+11
1.00E-08	4.55E+15
1.00E-10	4.55E+19

Table 4.2: Values of N as error estimate lessens. [4]

parallel processes, being a very lucrative possibility for applications, that would benefit from such an approach.

4.2.2 Hadoop MapReduce implementation of the algorithm

The Monte Carlo Algorithm 1 is adapted to the MapReduce, resulting in Algorithm 2. It describes four different execution stages in detail, preprocessing, map method which carries most of the weight of the algorithm, reduce which is just the identity function and the post-processing required to finalize the output. To adapt the algorithm to the MapReduce model, we considered several limiting factors that are imposed by the Hadoop framework on prospective algorithm implementations. First is the job initialization overhead and scheduling latency, which can take up to 20 seconds. The second factor is the large quantities of data needed to be transferred between data nodes, since, unlike usual MapReduce operations; we expect to receive the same amount of data as submitted for processing. Due to the aforementioned reasons we divide computation among processes by row blocks, since the output data will need to be retrieved only once as well as having only a single job initialization process, keeping the effect of framework overhead and I/O operations on run time to a minimum.

In parallel implementations, due to the nature of the algorithm, each separate node needs a copy of matrices A and P in order to complete its computations. As matrices grow large it may become unfeasible to satisfy this requirement due to memory constraints on the aforementioned nodes. Thus we consider data splitting approach as explained in [68]. Of the concepts discussed in that work, we chose to use sequential row blocks without supplying overlapping rows and no communication between nodes, as communication between map tasks is practically impossible with Hadoop. It is worth noting from the results in [68] that the error gain introduced by this approach becomes negligible for most matrices as their size grows large.

We split the initial matrix into $< key, value >$ pairs where key is the starting row number in its accompanying matrix row block, which is the $value$ of the pair, which we provide as input for our MapReduce program. The diagonal elements of the given matrix are stored for later use and steps 1 to 4 of the algorithm are executed as part of the map tasks, probability matrix P elements corresponding to the supplied row block are computed and summed for use of binary search in finding $nextpoint$, as suggested in [67].

A row block elements are replaced by values of $\frac{A[point][nextpoint]}{P[point][nextpoint]}$ concurrently to save on the number of operations in step 4. Map output is retrieved skipping the reduce stage and compiled into M for the remaining step of the algorithm, which is done by using M and previously stored diagonal elements to compute the estimate of the inverse.

Once the algorithm was designed, it was analyzed for its performance. Table 4.3 presents results of running the algorithm on a 16+1 cluster of m1.large Amazon EC2 [69] instances with 7.5 GB of memory and 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each) each, running Hadoop version 0.20.2. Tests were performed on randomly generated matrices with algorithm arguments giving an accuracy comparable to the one achieved by running about five iterations of CG.

Experiments were run with different number of parallel nodes to be able to calculate parallel speedup. Run times for the algorithm are shown in Table 4.3 and calculated speedup is shown on Figure 4.2. The results show that the algorithm scales well and the speedup achieved through use of parallelization is nearly ideal for sufficiently large matrices, where Hadoop job initialization and scheduling delay no longer skews results, with data transfer latency over the network being the only overhead, as expected from any embarrassingly parallel algorithm.

To evaluate the real performance gain from using this algorithm we should also compare these results to previous CG experiments with Hadoop and Twister CG implementations. Tables 3.1 and 3.5 in section 3.2 show the results for Hadoop and Twister CG implementations, respectively.

The Monte Carlo method, at the presented measure of accuracy, manages to perform better than the Hadoop CG implementation and has seem-

Algorithm 2 MapReduce Monte Carlo algorithm for matrix inverse. [4]

```
Preprocessing:
  1. Store diagonal elements of the matrix as vector $d$
  2. Split matrix $D$ into sequential row blocks
Map:
  Input $< key, value >$ where:
    $key$ – starting row index
    $value$ – sequential block of rows of $D$
  Execute Steps 1 to 4 of the original algorithm:
      Calculate part of matrix $C$
      $C_{ij} = -D_{ij}\frac{1}{d_i}; j = 1, 2, \dots, n; i = 1, 2, \dots, k$ where $k$ is the ↩
          length of the given row block and $d$ the diagonal ↩
          vector part in this block
      Calculate part of matrix $P$
      $p_{ij} = \frac{|a_{ij}|}{\sum_{k=1}^{n} |a_{ik}|}$
      Calculate part of matrix $M$, by MC on $C$ and $P$
      1: For $i = 1$ to row block length}
        1.1: For $j = 1$ to $N$
          1.1.1: Initialize $j^{th}$ chain sets sum vector ↩
              $SUM[k] = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$
          1.1.2: Set $u = 1$ and $point = i$
          1.1.3: While $|u| > \delta$
            1.1.3.1: Select a $nextpoint$, based on the transition ↩
                probabilities in $P$
            1.1.3.2: Compute $u = u\frac{C[point][nextpoint]}{P[point][nextpoint]}$
            1.1.3.3: Set $SUM[nextpoint] = SUM[nextpoint] + u$ and ↩
                $point = nextpoint$
        1.2: Assign $M$ elements as $m_{ik} = \frac{SUM[k]}{N}, k = 1, 2, \dots, n$
  Output $< key, value >$ where:
    $value$ is the resulting row of $M$ and
    $key$ is its index
Reduce: Identity Reducer is used by default
Postprocessing:
  1. Compile map output into $M$
  2. Step 5 of original algorithm $D^{-1}_{ij} = M_{ij}\frac{1}{d_j}$
```

Unknowns	1000	2000	4000	6000	8000	10000
1 node	35	64	200	312	751	1643
2 nodes	34	36	91	201	424	870
4 nodes	31	35	69	120	227	358
8 nodes	23	27	44	62	112	184
16 nodes	25	31	42	60	84	119

Table 4.3: Run times for Monte Carlo based Matrix Inversion using Hadoop MapReduce.[4]

ingly better scalability. At the problem size of 8000 unknowns the Monte Carlo method is almost 10 times faster than that of Hadoop CG implementation. Thus the analysis shows that Monte Carlo algorithm performs better than the Hadoop CG implementation, when the desired accuracy is low. CG implementation in Hadoop is inefficient because of the inherent problems of Hadoop with iterative algorithms.

However, the slow rate of convergence of the Monte Carlo method, due to rapid growth of the number of Markov Chains as one's desired accuracy for the solution grows, means it loses out to a CG implementation, which does not suffer from framework overhead, such as the Twister MapReduce variant. Twister CG implementation does a much better job with both larger problems and a higher resulting accuracy of the solution. Monte Carlo solution is 20 times slower for solving a linear system with 8000 unknowns and at the same time achieves much lower accuracy.

However, to maintain its efficiency the Twister CG implementation has to have the whole SLAE matrix in memory in the cluster's collective memory at once, an unfortunate drawback, restricting its use for problems of relatively small size. Monte Carlo solution also shares the same problem but with comparatively lesser consequences. When the problem size is huge the matrix is to be read in each iteration in Twister CG implementation, while the Monte Carlo method has to process each block at most just once, making it a bit slow, however would win over Twister at some point. We calculate that this would be achieved at a problem size of $\approx$120000 unknowns, with the collective cluster size memory being 7.5 X 16 GB.

The results show that this algorithm performs better than the Hadoop CG implementation, when the desired accuracy is low. However, the number of Markov chains needed to increase the accuracy grows in quadratic

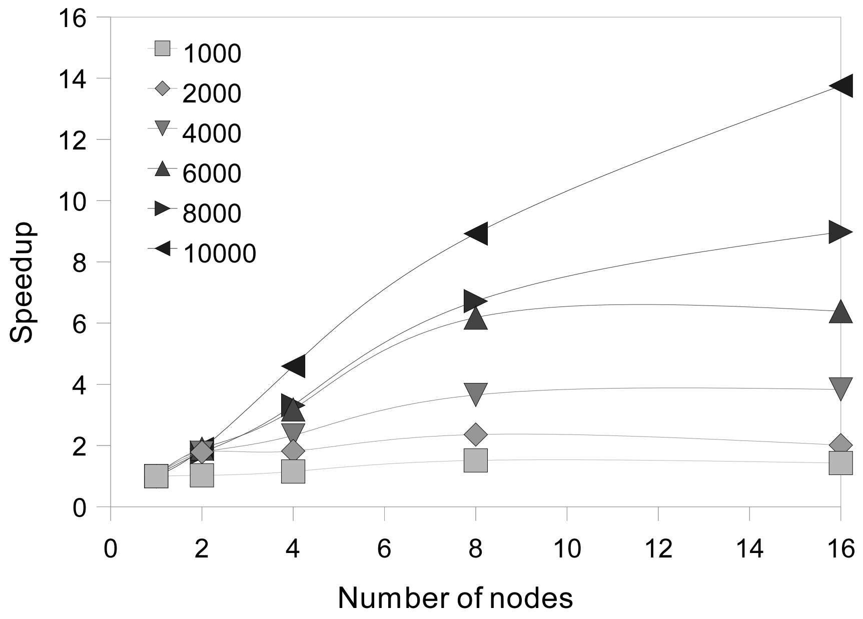

Figure 4.2: Speedup for Monte Carlo Matrix Inversion using Hadoop. [4]

rate and it is computationally very expensive to use the Monte Carlo algorithms when very high precision is needed. CG implementation in Hadoop is inefficient mainly because of the inherent problems of Hadoop with iterative algorithms. CG implementation on an alternative MapReduce framework, Twister, results in a more efficient linear system solver, which is more than 80 times faster than Hadoop Implementation. Comparing the Monte Carlo solver to Twister CG implementation shows that this solver cannot compete with CG, being 20 times slower for solving a linear system with 8000 unknowns and at the same time achieving much lower accuracy.

This algorithm is an example that achieving better performance by trying to find algorithms from different classes for the same problem is possible. Embarrassingly parallel algorithms based on the Monte Carlo method (class 1) are more suited for Hadoop MapReduce framework, thus the idea of solving linear systems with a Monte Carlo method was very intriguing for us. Unfortunately, the results showed that in this case, a CG implementation in an alternative MapReduce framework Twister gives much better results than an alternative algorithm on Hadoop.

4.3 Summary

We investigated an alternative approach for migrating iterative scientific computing application to the distributed computing frameworks like MapReduce. We ran [3] additional experiments to compare the performance of CLARA when adapted to MPI and Hadoop and to investigate how suitable alternative it is to using PAM for the same task of clustering arbitrary objects.

We also implemented a Monte Carlo based linear system solver to investigate whether we can achieve better performance by using alternative algorithms for which MapReduce framework is more suitable. Embarrassingly parallel algorithms based on the Monte Carlo method (Belonging to the first class of our initial classification) are more suited for Hadoop MapReduce framework, thus the idea of solving linear systems with a Monte Carlo method was very intriguing for us.

Unfortunately, the results showed that in this case, a CG implementation in an alternative MapReduce framework Twister gives much better results than an alternative algorithm on Hadoop. It greatly reduces the overall efficiency of the solver [4] and thus is not a good distributed computing framework when solving very large linear systems.

While the approach of using alternative algorithms has a potential to produce very interesting results it can not be used in every case. It also requires domain specific expert knowledge about parallel algorithm design and the internals of the MapReduce framework to be able to optimize the results. Thus our conclusion is that this approach is not a good alternative to adapting scientific computing algorithms to MapReduce unless the user is an expert in parallel programming.

The next chapter describes our work in investigating alternative MapReduce frameworks that aim to solve some of the Hadoop MapReduce problems with more complex and especially iterative algorithms.

CHAPTER 5

USING ALTERNATIVE MAPREDUCE FRAMEWORKS

In this chapter we investigate the feasibility of utilizing alternative MapReduce frameworks for solving resource hungry scientific computing problems. Alternative MapReduce frameworks, such as Spark [33], Twister [34] or HaLoop [21] are specifically designed to provide better support for iterative applications. Each of these frameworks have their own way of extending the MapReduce model to support more complex algorithms. While they may give up some advantages of Hadoop MapReduce to achieve this, they have been shown to provide significant performance improvements from Hadoop. Our goal is to evaluate how well they perform in comparison to both MPI and Hadoop.

We chose a number of typical scientific computing algorithms and adapted [5] them to three alternative MapReduce frameworks. The chosen algorithms were Partitioning Around Medoids (PAM), Clustering Large Applications (CLARA) and Conjugate Gradient linear system solver (CG). The chosen alternative MapReduce frameworks were HaLoop, Twister and Spark. We also compared their performance to the assumed worst (Hadoop MapReduce) and best case (MPI) implementations for iterative algorithms.

To be able to adequately measure the overhead of MapReduce implementations in comparison to MPI without comparing the programming language specific overhead, we chose a Java based MPI implementation MpiJava [45] with MPICH2 [47] as the native MPI library. When using MpiJava the algorithms still run on Java and only the actual MPI messages are transfered through the underlying native libraries. Thus, the performance of concurrent Java tasks is not affected and we can directly compare the performance of the data synchronization and the parallelization process in general.

5.1 Experiment setup

All the experiments were performed in Amazon EC2 public cloud on 32+1 instances. The Amazon EC2 instance type was m1.large with 7.5 GB memory, 2 x 420 GB storage and 2 cores with 2 EC2 Compute Units each. EC2 computing unit represents the capacity of allocated processing power and 1 EC2 unit is considered to be equivalent to an early-2006 1.7 GHz Xeon processor based on benchmarking results. Amazon Inc did not publish the full details of the underlying hardware at the time of running the experiments.

The experiments were conducted in the same Amazon EC2 availability zone (us-east-1b) to affirm that the experiment results were affected as little as possible. The software environment was set up in Ubuntu 12.04 server operating system. MPI tests were performed using MpiJava 1.2.7, MapReduce tests were performed using Hadoop 1.0.3, Twister 0.9, Spark 0.8.0 and HaLoop revision 408. MpiJava internally used MPICH2 1.4.1p1 for the MPI communication and HaLoop used a modified Hadoop 0.20.0.

None of the framework configurations were optimized in detail to avoid giving any of them an unfair advantage. Modifications were only performed to assure that the algorithm executions were parallelized in a balanced manner in each of the frameworks and that the number of working processes was equal to the number of cores in the cluster. This included lowering the HDFS block size to 12 MB, changing the number of input files as needed to force computations in Map tasks to be of equal size and specifying the number of reducers to be equal to the number of available cores.

5.2 Evaluation results

5.2.1 Partitioning Around Medoids (PAM)

Table 5.1 provides the runtime results for the PAM k-Medoid algorithm implementations. We ran all four implementations with data sets consist-

	MPI			HaLoop			Spark			Twister			Hadoop		
Nodes	13333	40000	80000	13333	40000	80000	13333	40000	80000	13333	40000	80000	13333	40000	80000
1	12.7	57.7	213	645	730	938	34	220	811	13.1	62.1	282	1624	1732	1912
2	6.7	29.0	106	427	485	655	29	147	528	7.15	33.0	146	984	1018	1094
4	3.7	14.7	54.1	313	341	439	24	111	388	4.06	18.7	79.4	634	652	706
8	1.9	7.5	28.0	234	256	364	22	97	337	2.33	11.9	41.8	465	472	490
16	1.2	3.9	15.23	231	235	259	21	88	299	2.66	7.21	26.6	369	378	389
32	1.0	3.8	13.31	198	209	252	30	86	281	3.34	5.05	16.5	359	362	378

Table 5.1: Running time (s) of PAM clustering results with different frameworks and data set sizes. [5]

	MPI			HaLoop			Spark			Twister			Hadoop		
Nodes	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mi l
1	222	442	660	294	569	806	612	1197	1783	273.6	530.3	792.4	617	836	1081
2	111	221	330	173	294	428	319	618	918	166.6	323.5	472.2	436	473	647
4	58	113	169	113	228	236	181	344	490	81.8	164.0	244.0	253	349	369
8	29	57	85	87	110	148	106	197	287	50.1	98.6	147.0	180	235	294
16	15	29	43	74	83	101	68	122	178	24.0	49.0	75.9	138	132	161
32	8	15	22	72	72	79	57	61	143	15.8	24.1	36.4	92	101	113

Table 5.2: Running time (s) of CLARA clustering results with different frameworks and data set sizes. [5]

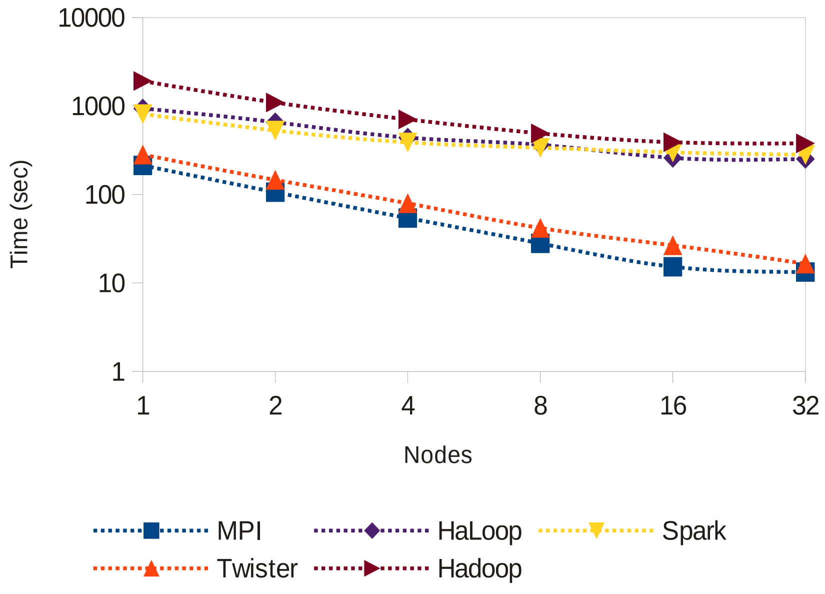

Figure 5.1: PAM runtime comparison when clustering 80000 objects. [5]

ing of 13 333, 40 000 and 80 000 2-dimensional points with double value type coordinates and measured their runtime. The maximum relative standard deviation of the runtime was 2.6% for MPI, 12% for HaLoop, 13% for Spark, 24.8% for Twister and 10% for Hadoop. MPI implementation was the fastest in all cases being clearly the most efficient. Twister performed the best among the four MapReduce frameworks and was rather close to MPI.

HaLoop and Spark clearly performed better than Hadoop in every test case but were significantly slower than either MPI or Twister. For the first two datasets, Spark was several times faster than HaLoop, but it achieved better parallel speedup on the largest dataset and managed to achieve a better runtime when executed on 16 and 32 nodes. Figure 5.1 provides a comparison of PAM implementations when dealing with the largest dataset to better illustrate the differences between them.

5.2.2 Clustering Large Applications (CLARA)

Table 5.2 provides the runtime results for the CLARA k-Medoid clustering algorithm when the data sets consisted of 1 333 333, 2 666 666 and 4 000 000 2-dimensional points. The maximum relative standard deviation was 0.5% for MPI, 11.2% for HaLoop, 13.4% for Spark, 10.4% for Twister and 13.2% for Hadoop. Compared to PAM, these results show how the implementations can manage a larger dataset when the algorithm is not iterative. In all the MapReduce implementations CLARA required 2 MapReduce jobs, first to randomize the data and to apply sequential PAM on smaller sampled datasets and second to choose the best clustering from the ones generated in the previous step. MPI again showed the best results, achieving almost perfect parallel speedup.

Both HaLoop and Twister also performed quite well, being constantly better than Hadoop but it were still constantly slower than MPI. However, Twister scaled much better than HaLoop and was twice as fast on 32 nodes for the largest dataset. Spark was clearly inefficient in comparison to the other four, being the only one slower than Hadoop. Already on a single node cluster it was clearly slower than the other frameworks and also was constantly the slowest framework on the biggest data set.

The main reason for this is that there are tens of millions of small Java objects that are shuffled around in the CLARA randomization stage, where a global random sorting of data is performed. Sparks RDD operations seem to significantly slow down when that many objects are being moved around even when the process is taking place on a single node.

The differences of the CLARA implementations when dealing with largest dataset are illustrated on Figure 5.2.

5.2.3 Conjugate Gradient linear system solver (CG)

Table 5.3 provides the runtime results for the CG algorithm. The dataset consisted of a dense matrix of 4 million, 16 million and 64 million non-zero elements. While CG is typically used for sparse matrices, matrix-vector operations with dense matrices were easier to implement and simplified adapting the benchmark on different distributed computing frameworks.

In comparison to PAM and CLARA, most of the data does not have to be repartitioned between nodes between iterations and can stay local to the original processes. Of course, this does not apply to Hadoop MapReduce

	MPI			HaLoop			Spark			Twister			Hadoop		
Nodes	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil
1	0.10	0.35	1.30	154	851	2809	9.0	10.2	93.7	1.71	3.29	10.19	380	1008	4191
2	0.44	1.61	5.30	151	370	1498	9.3	9.2	28.0	1.39	2.14	6.19	299	533	2291
4	0.47	1.71	6.34	125	200	838	10.3	9.6	12.7	1.73	1.85	3.38	298	388	1197
8	0.53	1.65	7.46	130	176	371	11.3	11.7	9.0	1.44	1.57	3.19	298	303	625
16	0.60	1.73	7.27	130	127	226	11.3	11.3	9.0	1.51	1.57	2.75	297	297	406
32	0.65	1.82	6.60	134	130	181	13.0	12.0	11.3	1.80	2.15	3.16	298	298	300

Table 5.3: Running time (s) of Conjugate Gradient results with different frameworks and data set sizes. [5]

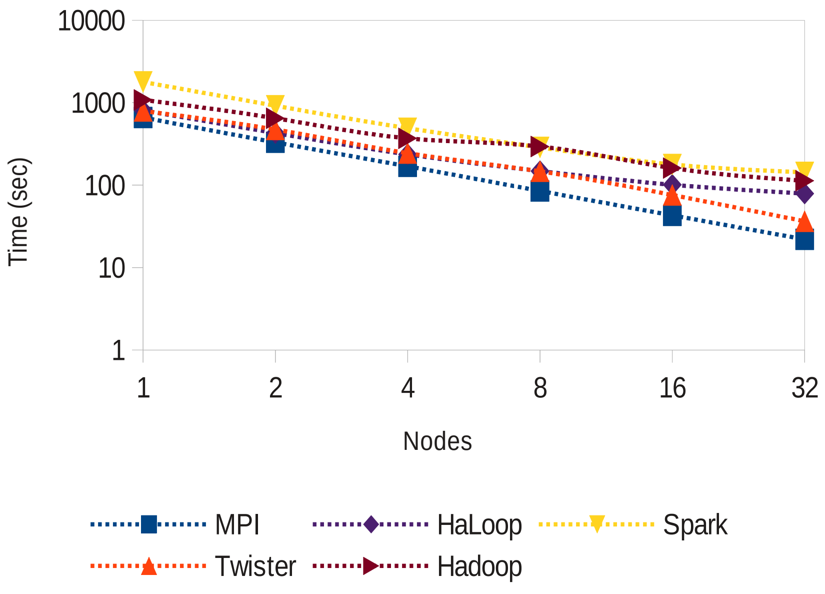

Figure 5.2: CLARA runtime comparison when clustering 3 million objects. [5]

as it has no way of keeping data in memory between iterations. The maximum relative standard deviation was 5.4% for MPI, 15.2% for HaLoop, 20.6% for Spark, 12.5% for Twister and 18.2% for Hadoop.

MPI CG was the fastest implementation for the two smalled data set sizes. Moreover, as long as the dataset fits into the memory of one machine, there is actually no relative speedup from using MPI on multiple nodes as sending messages between processes adds up to a significant overhead. As the number of nodes was increased for the larger dataset, Twister actually performed better than MPI. Considering it was still slower than the MPI's single node execution, it only shows that Twister has less communication overhead and thus scales better. At the same time its runtime is degraded by other framework issues, such as object serialization overhead.

Spark also performed well considering it takes additional time to configure and start Spark processes in comparison to MPI. With first two datasets there is actually no gain from using multiple nodes and most of the total time was actually spent on configuring the Spark processes, loading data to RDD, deploying them and cleaning up after computation was done. Except for the last dataset where Spark actually achieved parallel

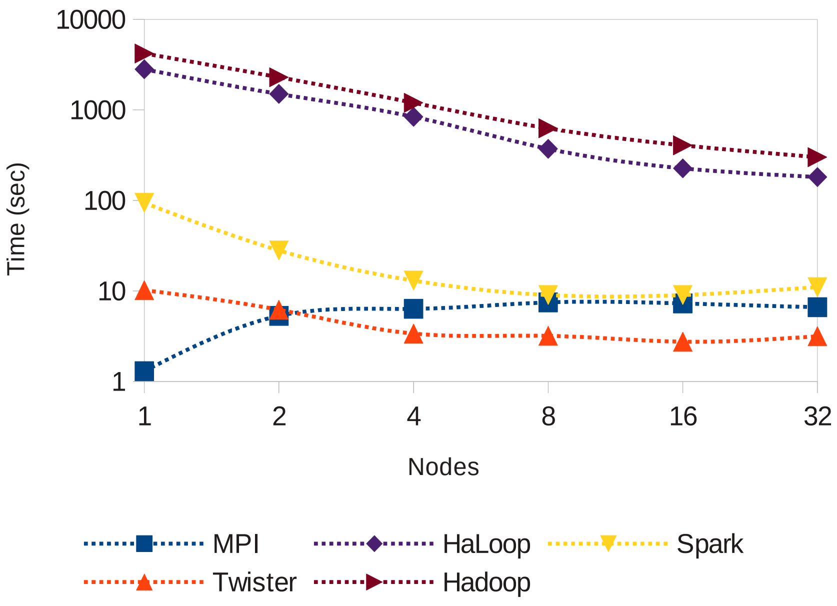

Figure 5.3: CG runtime comparison with data set size of 64 million elements. [5]

speedup up to 8 nodes, after which the computational complexity of a single task became too small.

HaLoop again performed constantly better than Hadoop, but compared to MPI, Twister and Spark the results were still extremely slow, showing that it gains little if any benefit from HaLoop map input caching when large amount of data still needs to processed in reduce tasks and thus need to be transported across machines. The differences between the Conjugate Gradient implementations when dealing with largest dataset are illustrated on Figure 5.3.

5.3 Discussion

From the runtime experiments it is clear that Spark, Twister and HaLoop can deal with iterative algorithms better than Hadoop but not as well as MPI, with Twister being clearly the fastest of them. While this is somewhat expected, the difference is larger than we envisioned for both HaLoop and Spark and it shows that they have great scope for improvements. Another important thing to note is that while the HaLoop results are constantly in same relation with both MPI (always several times slower) and Hadoop (always around 1.5 times faster), Spark results seem to be greatly affected by the characteristics of the benchmarking algorithms and their dataset composition.

In the case of PAM, there is little benefit from caching the input data in memory in both Spark and HaLoop frameworks as the small objects are constantly repartitioned between clusters, which involves regrouping and transporting them between concurrently running tasks at every iteration.

In the case of CLARA, Spark has great difficulties, which can be attributed to the peculiarities of the data set composition as CLARA dataset consists of millions of very small objects (2D points) which are stored in Spark RDD's. It was the same in the case of PAM, but its dataset is much smaller so the problem did not arise.

This problem is especially evident when considering that Spark already has difficulties handling the CLARA dataset when running on a single node and from the fact that is has no trouble with the CG algorithm which requires even more memory in where to store the input data. In addition, the CG objects themselves (matrices) are much larger. Our hypothesis is that Spark RDD's are inefficient when they consist of significantly large number of very small objects. We will investigate it in detail in our future work.

The Conjugate Gradient experiment was the closest Spark could get to MPI results. When processing the largest dataset, Spark results are comparable to MPI results (11 vs 6.6 seconds) on 32 nodes if we take into account that most of the time in Spark was actually spent on framework overhead of scheduling Spark processes, initiation and cleanup, as is evident from the fact that runtime never falls below 9 seconds. Similarly, the MPI runtime is also greatly affected by its communication overhead which depends on the size of the messages and the number of processes, as seen from the increase in MPI CG runtime in Table 5.3 for the largest dataset as the cluster grows.

While Twister is the fastest of the investigated MapReduce frameworks, it is not as usable as Spark when stability and fault tolerance are critical. Twister had the most issues and crashes when running performance experiments. These issues greatly complicated executing the performance tests in an automated manner and its fault recovery system is not ideal for longer running tasks, as only the Map input and data passed by the main program is recovered. Any data kept in memory across iterations would still be unrecoverable.

We should also take into account that there are many advantages to using MapReduce-like frameworks that can significantly simplify adapting algorithms for parallel processing. From the ease of partitioning and distributing data through the Hadoop Distributed File System (HDFS) to near automatic fault tolerance and parallelization. Still, the efficiency of the result can not be ignored for scientific computing applications as performance is one of the main goals when creating and optimizing parallel scientific computing applications.

Executing long running scientific experiments and simulations requires a large number of computing resources and whether we are using supercomputers, grids or cloud resources, the cost and limit of such resources are crucial. Thus the chosen higher level distributed computing framework should at least be comparable to MPI and not be a several times less efficient.

All the chosen benchmarking algorithms were quite different based on their iterative nature, dataset composition and size of input and intermediate data. It provided a good opportunity to compare the involved frameworks from several different perspectives. It is evident from the results that only Twister has a comparable performance to MPI for all the benchmarks and thus is a good candidate for iterative applications when performance is important.

However, Twister loses many of the advantages typically attributed to MapReduce frameworks because it does not use a distributed file system and it is generally unstable. Spark's results depended strongly on the algorithm characteristics and thus Spark can only be used in some of the cases. HaLoop results were only marginally improved over Hadoop in comparison to MPI and as it is in a prototype state, it is not really usable in real situations.

5.4 Summary

This chapter covered the evaluation study of alternative MapReduce frameworks for parallelizing and scaling up scientific computing algorithms. We measured the performance of Twister, Spark and HaLoop in comparison to Hadoop MapReduce and MPI and have shown that the performance of iterative MapReduce frameworks can in some cases be close to MPI and be much better than Hadoop MapReduce in most of the cases. Next chapter investigates the performance of frameworks that use a different distributed computing model, which is Bulk Synchronous Parallel model.

CHAPTER 6

ADAPTING SCIENTIFIC COMPUTING ALGORITHMS TO BULK SYNCHRONOUS PARALLEL MODEL

The first two approaches covered in the previous two chapters mainly considered adapting algorithms to distributed computing frameworks based on the MapReduce model. In this chapter we investigate the third approach, which is adapting scientific computing problems to Bulk Synchronous Parallel (BSP) distributed computing model [24]. BSP model and a number of its implementations have already been described in more detail in section 2.1.2.

In the first section, we describe our work [1] in investigating Bulk Synchronous Parallel (BSP) model to find how suitable this model and its current implementations are for running more complex scientific computing algorithms in comparison to MapReduce and MPI. We adapted two scientific computing algorithms to two BSP implementations and two MPI based implementations and performed benchmarking experiments in a varying size cluster with different dataset sizes to evaluate whether the BSP based implementations are able to achieve the same level of parallel efficiency as MPI implementations. We were also interested to find out whether these implementations provide the same advantages as most of the MapReduce frameworks do, such as automatic parallelization and fault tolerance.

In the second section, we describe our proposal [6] for a new BSP-inspired computation model and its proof of concept implementation for the Hadoop Yarn platform which is called NEWT. It can be used as an alternative to MapReduce in any existing cluster. We analyze the performance of the proposed NEWT distributed computing framework in comparison to the most suitable BSP implementation among the ones that we investigated previously. We measure the efficiency of the NEWT fault tolerance and recovery mechanisms and discuss what additional features and improvements should be added to NEWT.

6.1 Performance evaluation of BSP implementations

Bulk Synchronous Parallel (BSP) is a distributed computing model which is designed for iterative algorithms. Most of its implementations (described in section 2.1.2) target either graph processing (Pregel, Apache Giraph, Stanford GPS) or iterative algorithms (Hama, BSPonMPI). Last ones are the BSP–based frameworks that mainly interest us.

In BSP, computations are executed in a sequence of supersteps (or iterations) with communication operations performed between the supersteps. BSP communication primitives usually consist of sending messages between concurrently working processes. There is also a global barrier between supersteps which means that the next iteration will not start before all concurrently working processes have finished previous superstep and intermediate communication operations. These compulsory barriers and the message-based data synchronization make BSP very similar to using MPI with only collective communication methods (such as All-to-All, Gather and Scatter) which have barrier built into them.

While BSP applications are more structured in comparison to MPI, there is little difference for applications that require this kind of strict structure anyway, such as iterative applications where a portion of data needs to be synchronized between iterations. As a result, implementing iterative algorithms using BSP is very similar to implementing them in MPI.

In addition, having compulsory barriers means that there are specific periods in the algorithm execution process where some processes are waiting for others to complete their tasks and these periods are located at generally predictive time intervals. These barrier moments provide an opportunity to implement fault tolerance in a structured and transparent manner. For example by using checkpointing after every $N'th$ superstep. BSP implementations have potential to provide some of the advantages that have made MapReduce a widely adopted distributed computing framework in large scale data processing field.

Another aspect is the parallel performance of BSP frameworks, which in the case of dealing with resource hungry scientific computing applications is very important. It is important to study how suitable the current implementations of BSP are for parallelizing iterative algorithms in comparison to MapReduce and MPI. We chose two iterative scientific computing algorithms for benchmarking. These are Conjugate Gradient and Partitioning Around Medoids, which are described in chapter 3.

We implemented these two algorithms on two BSP implementations (BSPonMPI and Apache Hama) and on two Java MPI implementations (MPJ Express and MpiJava MPI). All of these implementations were previously described in chapter 2. We used these algorithm implementations as benchmarks and executed performance measuring experiments in the Amazon EC2 cloud. Configuration of these experiments is described in the next subsection.

6.1.1 Experiment configuration

Experiments were conducted in Amazon EC2 public cloud on 16+1 instances. One extra instance was required for both Hama and Hadoop master node, which did not take part in the actual calculations. The first experiment used c1.medium (High CPU Medium) instances with 1.7 GB memory and 2 cores with 2.5 EC2 Compute Units each. The second one used m1.medium (Standard Medium) instances, each with 3.75 GB of memory and 2 EC2 Compute Units (1 virtual core). Thus, each of the experiments had 32 processor cores. EC2 computing unit represents the capacity of allocated processing power and 1 EC2 unit is considered to be equivalent to an early-2006 1.7 GHz Xeon processor.

All instances used Ubuntu 10.10 as the operating system. MPI tests were performed using MPJ Express version 0.38 and MpiJava version 1.2.7, BSP tests were performed using Hama version 0.5 and BSPonMPI version 0.3 and MapReduce test was performed using Hadoop version 0.20.2. Both MpiJava and BSPonMPI used MPICH2 version 1.4.1p1 for the underlying MPI communication.

6.1.2 Partitioning Around Medoids (PAM) results

Nodes	BSPonMPI	MPJ Exp.	MpiJava	Hama	Hadoop
1	24.995	13.26	30.41	17.75	118.7
2	11.882	7.47	15.19	9.92	72.3
4	6.050	3.87	7.37	5.66	49.2
8	2.942	2.15	3.64	3.34	38.6
16	1.593	1.52	1.88	2.37	34.8

Table 6.1: Running time (s) of clustering 80 000 objects with PAM. [1]

The first experiment we performed was PAM, which was parallelized using 5 different distributed computing frameworks or parallel programming libraries: Hadoop, Apache Hama, MPJ Express, MpiJava and BSPonMPI. Each test involved clustering 80,000 objects into 32 clusters and the average time per iteration is shown in Table 6.1. For simplification, we used 32 processes of either MPI or BSP and executed them on the chosen number of machines. This allows us to take full advantage of the 32 virtual cores when we utilize all 16 instances, but it means sending additional messages when running on lower than 16 machines.

MPJ Express achieved the best performance, with BSPonMPI being a close second. MpiJava performed worse than the first two and Hama result was fourth, being more than 60% slower than MPJ Express. We are mainly considering the results on 16 nodes as even on 1 node we still ran 32 MPI or BSP processes because we needed to cluster data into 32 clusters. The results on the lower number of nodes illustrate how well the specific implementations handle the communication between processes running on the same virtual machines, and while this result is interesting and shows the differences of the respective implementations, it does not often affect running real life applications, as common practice is not to have more than a small number of processes per core.

We also executed MapReduce experiments and included the results in the same Table 6.1 as the last column to give a full picture. It is clear that MapReduce performs much worse than any of the MPI or BSP implementations, being at least 14 times slower than any other implementation when running on 16 nodes. It again confirms (as stated in the Chapter 3) that MapReduce is not suitable for iterative applications.

6.1.3 Conjugate Gradient (CG) results

p	MPJ Express	MpiJava	BSPonMPI	Hama
1	476.34	479.15	468.54	453.48
2	276.42	287.15	275.88	377.23
4	160.63	163.89	158.21	273.36
8	90.53	99.70	91.35	206.74
16	55.16	71.01	58.87	204.31

Table 6.2: 3D Heat simulation time (s) on p processes. [1]

Table 6.2 presents the 3D heat diffusion simulation (solving systems of linear equations by applying parallelized Conjugate Gradient algorithm iteratively) runtimes using four different communication libraries - Apache Hama, MPJ Express, MpiJava and BSPonMPI. The resolution of the simulation was kept fixed for all experiments, resulting in a system of linear equations with 8,000,000 unknowns and a total memory consumption of 1 GB. Message size varied from just 8 bytes (dot product and error margin synchronization) to 0.3 MB (matrix vector multiplication synchronization).

Hundred steps of the simulation were performed, each equivalent to one invocation of CG. For the given problem size, 4 CG iterations were necessary to achieve an error margin [1] of 10^{-6}, resulting in around 400 CG iterations in total. Superstep count for Hama was 1337 and quite a bit higher for BSPonMPI, as it requires a synchronization step after each DRMA (Direct Remote Memory Access) registration of an array. As discussed in previous section, running such a large number of Hadoop MapReduce iterations is not feasible, and thus was not included in this case.

These results are similar to the results of the PAM experiment. However, as this experiment has a much higher number of supersteps, it emphasizes the differences between the separate MPI and BSP implementations. They confirm that the BSPonMPI performs very closely to the fastest MPI implementation MPJ Express. Hama performs worse than in the previous experiment, indicating that it has troubles with a larger number of iterations, as in the PAM experiment there was an average of 30 supersteps

[1] Counted as the maximum norm ($||a||_{\max} = \max\{|a_i|\}$) of the residual vector used in the CG algorithm.

(two per PAM iteration), but in the case of this experiment there were at least 1337 supersteps.

6.1.4 Apache Hama performance issues

The results in the Tables 6.1 and 6.2 show that Hama is performing worse than BSPonMPI and both MPI implementations. The likely reason for the slowdown is the object serialization Hama employs when transmitting messages. JBSP [70] is one example of a BSP framework which has been known to be affected by this problem. While Hama authors have performed benchmarks [71] to investigate its performance, they have not directly compared the results to MPI and other BSP implementations. To investigate whether our assumptions are correct, we performed an additional experiment using the same software and cluster configuration as in the CG benchmark.

Both Apache Hama and BSPonMPI were used to run a program that simply performs 100 barrier synchronizations. Further tests transmitted a single message to the master at each superstep, with the message size increasing for consecutive tests. For synchronization times we considered Hama's own TIME_IN_SYNC_MS counter, divided by the number of processes (one for each node in the cluster), and the cumulative duration of each bsp_sync() invocation for BSPonMPI. The results of this experiment are provided in Table 6.3.

Message size	Hama time	Hama sync	BSPonMPI time	BSPonMPI sync
N/A	25.633	10.38	0.26	0.23
10000	31.58	16.03	1.71	1.45
100000	67.77	50.77	10.89	9.80
1000000	280.8	267.1	100.5	100.1

Table 6.3: Benchmark runtime and synchronization times (s) for BSPonMPI and Hama. [1]

For the test case without any messages, in addition to the obvious framework initialization overhead, Hama seems to spend a considerable amount of time just performing the barrier synchronization, resulting in synchronization that is 50 times slower than with BSPonMPI. When a 0.076 MB message (array of 10000 double numbers) is transmitted at each superstep, the advantage of BSPonMPI decreases to being 10-fold. At message size of 0.76 MB, BSPonMPI is only 5 times slower, further decreasing to just 2.5 times at 7.6 MB messages.

While BSPonMPI demonstrates a clear linear increase in synchronization time in relation to message size, Hama's pattern seems to be more erratic, showing a large overhead without a clearly distinguishable source, which is further amplified with the increase to the number of messages being transmitted. The serialization mechanism Hama employs is user defined serialization sequence for all transmitted messages and objects.

It is generally the optimal approach to serialize objects in Java and the additional source of overhead is likely caused by the use of Hadoop RPC (for transfer of messages) and Zookeeper (for barrier synchronization).

6.1.5 Discussion of the evaluation results

Both the PAM (table 6.1) and CG (table 6.2) results show that the fastest BSP implementation BSPonMPI performs only slightly worse than the fastest MPI implementation MPJ Express, but the difference is negligible when the number of nodes in the cluster is 16. The comparison of BSPonMPI and MpiJava is especially interesting for us because they both use the same underlying MPICH2 MPI implementation and access its routines through Java Native Interface (JNI), yet BSPonMPI performs better in both cases.

The slight advantage of BSPonMPI is most likely caused by certain implementation decisions taken by authors of MpiJava, but since the difference in runtimes is in favor of BSPonMPI anyway, it is clear that the BSP model does not impose any significant overhead (at least in case of these algorithms).

The results show us that BSP implementations can be just as excellent as MPI implementations for parallelizing these types of algorithms. As the chosen benchmarking applications represent the typically used complex scientific computing algorithms, this clearly shows that the BSP computing model is a good choice for solving scientific computing problems. However, at the same time, the current BSP implementations are not perfect. BSPlib is a legacy system which is not usable on new hardware.

Hama does not perform well in comparison to BSPonMPI. BSPonMPI is as efficient as MPI in our experiments, but in comparison to MapReduce, it does not provide fault tolerance and does not significantly simplify the creation of BSP applications, which are some of the reasons why we were considering different distributed libraries and frameworks in the first place.

It is clear that the existing BSP frameworks and libraries either do not provide the fault tolerance required by long running applications or are designed for solving specific types of problems, such as graph processing and thus require extra effort for adapting other kind of applications to them. We decided to propose a BSP-inspired parallel programming model (not unlike MapReduce) which enables transparent stateful fault-tolerance and has better support for general-purpose iterative algorithms than currently existing frameworks.

6.2 Fault tolerant BSP framework for Hadoop YARN

NEWT [6] is an distributed computing framework for the Hadoop YARN platform which was designed to replace the default MapReduce framework. It uses a modified Bulk Synchronous Parallel (BSP) model and strives to retain the advantages of MapReduce while supporting a wider range of algorithms. Its goals are to provide automatic fault recovery, retain the program state after fault recovery, provide a convenient programming interface and to support iterative scientific computing applications.

NEWT parallel programming model adopts an approach similar to continuation passing for implementing parallel algorithms and facilitates fault tolerance inherent in the BSP program structure. A NEWT application consists of a number of BSP functions. Each NEWT function can be seen as a standalone BSP function which returns the value of the next function to be applied. One such BSP function is executed at each superstep and it specifies what function will be applied on the next superstep. This allows not only to iterate over multiple different BSP functions (which may be needed for more complex algorithms), but also to dynamically decide the program flow or to decide when an ending condition is met.

In addition to supporting cyclic execution of BSP functions, it also allows to hide the fault tolerance operations from the user by executing checkpoint creation and recovery operations between supersteps. This means that users do not have to modify their NEWT applications (which is often required in MPI) to enable or disable fault tolerance and the complexity of the application is not affected.

We can represent such BSP applications as finite state machines (FSM), where the result of applying each superstep is equivalent to a state in FSM. This leads us to view programs under the BSP model as suitable for an abstract computer, consisting of:

- Memory - contains message queues and the mutable state of the program.
- Label to instruction map - Specifies the relation between instruction labels and instructions.
- Function pointer - holds the label of the next instruction to be executed.
- Communicator - Mediates sending messages between concurrently running BSP processes.

The instructions are user-defined BSP functions and the message queues hold incoming and outgoing messages. Writing a NEWT program is similar to using continuation-passing style from functional programming, except that we allow sending messages as side effects. User writes a number of BSP superstep functions which process received messages, performs local computations to transform the current state, sends messages to other processes and specifies what is the next BSP superstep function to be executed.

The following pseudo code illustrates the inner workings of the described FSM using high-level imperative programming concepts:

```
$state \leftarrow initialState$
$next \leftarrow initialLabel$
while $true$ do
  $next \leftarrow execute(next, state, comm)$
  $barrier(comm)$
  if $next == none$ then
    $break$
  end if
end while
```

The $execute$ call runs the BSP function defined by label $next$ and returns the label of the next BSP function to be executed. The modification of the state and message transfer through communicator $comm$ is achieved as a side-effect of BSP function calls. Typical communication primitives similar to MPI team communication operations are defined to simplify dealing with data synchronization. These primitives are accessible through the communicator $comm$. As per BSP model, $barrier$ initiates communication and synchronizes all the concurrently working NEWT processes.

The state, next BSP function label and incoming messages can be stored into a distributed file system or other non-local persistent storage between invocations of $execute$, so that the full state of the NEWT program can be restored and restarted. This recovery can be achieved seamlessly. When a failure is detected, NEWT processes that failed can be restarted on other nodes and their state can be restored from the previous checkpoint. It is also important to note, that processes that did not fail can simply rewrite their current state from an earlier snapshot and then continue the execution to complete the recovery.

Apart from creating BSP functions, user also has to define the state and assign labels to each of the BSP functions. State of a NEWT program can be defined as the data that is persisted in memory across different BSP function executions. Each BSP function has to explicitly return the label of the next BSP function to be executed. NEWT can be used to easily model any MapReduce program by creating two BSP functions labeled 'Map' and 'Reduce' and defining that the state is empty (statless). Data is passed along only by sending messages at the end of the 'Map' stage and becomes available for processing at the 'Reduce' stage.

Any number of such stages can be defined in NEWT and we can also model iterative programs with an arbitrary number of iterations. Iterative NEWT applications can be defined by BSP functions returning their own labels while iterations are to be continued. In addition, instead of a single BSP function returning its own label, a number of BSP functions could be executed in a cycle to make up more complex iterative applications.

Once the iteration ending condition is achieved, a label for an ending BSP function can be returned which would act as the exit from the iterative cycle. At specific intervals, the current state of the application and the current message queue would be stored to persistent storage. The frequency of this process can be configured based on how long the application is expected to run and how stable is the underlying hardware.

A reliable re-scheduling mechanism and a resilient storage environment are required for successful fault recovery and we chose the following Hadoop components for this purpose. NEWT uses Yet Another Resource

Negotiator (YARN) [22] to manage cluster computing resource allocation, task management and re-executing failed tasks in case of machine or network failures. It also allows to execute NEWT applications inside an existing Hadoop cluster. It also uses Hadoop Distributed File System (HDFS) [28] to store input, output and checkpoint data of NEWT applications. Data in HDFS is divided into smaller blocks and replicated across different physical locations, which is very convenient for storing checkpoint data in a reliable manner.

NEWT coordinator is implemented as a YARN Application-Master, which can request and manage YARN containers [72]. YARN uses containers to separate processes running in Hadoop cluster, assign computing resources to them and re-execute failed tasks. The concurrently working NEWT BSP processes run inside these YARN containers.

Message passing is implemented using Apache MINA [73]. It is a framework which simplifies developing high performance network applications by using Java NIO to provide an event-driven asynchronous messaging API over TCP or UDP. Initially it was planned to use MPI for this, but while there have been developments in making MPI run in a Hadoop YARN cluster [74], there were no working and stable implementations when NEWT was created.

There are three main interfaces that users interact with when creating NEWT applications - *Stage*, *BSPState* and *BSPComm*. *Stage* represents a BSP function in NEWT. A user defined state class, which extends *BSPState* is used to define what data needs to be kept in memory between supersteps and saved in checkpoints.

In this class, user has to define the execute method, which signature is:

- *String* execute(*BSPComm* comm, *BSPState* state)

The *BSPComm* class exposes message passing functionality and it supports the following functions:

- send(*Writable* message, *int* pid)
- send(*Writable* message, *int* pid, *String* tag)
- sendAll(*Writable* message)
- sendAll(*Writable* message, *String* tag)

Messages are subclasses of the Hadoop *Writable* interface. Message passing is race-free and messages can be safely retrieved from the receive queue using the following functions:

- move()
- move(*int* pid)
- getReducedValue(*String* tag, *ReduceOp*<*Writable*> op)

The *move* function returns all messages that were received. These messages are sorted by the sender process ID in the ascending order and by sending order. When a process id is given as an argument, the *move* function retrieves messages from only that process. The *getReducedValue* method mimics the functionality of the *MPI_Reduce* function. It takes advantage of the optional *tag* parameter that can be specified when sending messages by applying a *ReduceOp* aggregation function on all the received messages which have the specified tag attached to them. The framework includes a number of common reduction operations such as sum-of-floats or maximum-of-integers.

Currently Java 1.7 closures are used to define program functions, but the goal is to migrate to Java 1.8 closure syntax in the future as it is much more convenient and natural to write Java 1.8 closures.

6.2.1 Adapting algorithms to NEWT

Parallelizing algorithms using NEWT is similar to parallelization in MPI. In fact, the only significant difference is having to divide the normal application flow into separate BSP functions. When migrating existing MPI applications to NEWT, every communication operation in the original MPI application would be a candidate for the dividing location into separate BSP functions.

A number of single message and collective communication message operations can be used to synchronize data between concurrent NEWT processes. The available operations are identical to their Java MPI API counterparts with the same name.

We describe the adoption of two algorithms to NEWT to illustrate this process. These two algorithms are Conjugate Gradient (CG) linear system solver and PAM clustering method, which were already described in more detail in sections 3.1.1 and in section 3.1.2.

CG is a rather complex iterative algorithm for solving sparse systems of linear equations. The algorithm is typically parallelized using the Single Program Multiple Data (SPMD) model, where all the data is split evenly among the concurrently working processes.

To adapt CG to NEWT, each of the CG iterations need to be split into several supersteps because there are three occasions when data needs to be synchronized between processes. These include two dot product operations and one matrix-vector multiplication which requires synchronizing overlapping portions of vectors between neighbors.

Dividing the algorithm into different stages is visualized on Figure 6.1 :

- Init - Initializes all needed state variables and performs the initial guess for the linear system solution.
- Start of Loop - Defines the beginning of the iteration until the first dot product operations and sends the partial dot product and error value to all other processes.
- Check Ending Condition - Calculates the global dot product and error value. It uses the error value to decide whether the solution is close enough or whether more iterations are required, returning either the label of stop stage or the continue the loop stage.
- Continue Loop - Calculates local values for vector p and sends its overlapping edges to two neighboring processes to prepare for global matrix-vector multiplication.
- Do MatVec - Receives the overlapping p edges and computes the matrix-vector multiplication. It also computes another partial dot product and sends its result to all other processes.
- End of Loop - Calculates the global dot product, performs calculations at the end of CG iteration and returns the label of the 'Start of Loop' stage.
- Stop - This stage is executed when the error value gets below specified margin or the maximum iteration count is reached. It finalizes the result computation and outputs it.

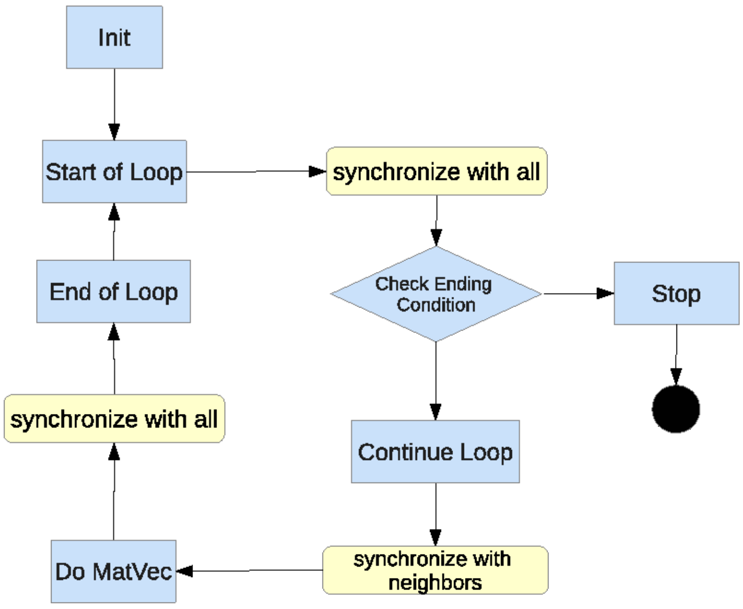

Figure 6.1: Structure of CG under the proposed model. [6]

Apart from having to split each stage into a separate BSP function, the resulting structure is very similar to programs written under common message passing paradigms (such as BSPlib or MPI).

Partitioning Around Medoids (PAM) is an iterative clustering method that divides a set of objects into k clusters using only pairwise distances between objects. It has already been described in more detail in section 3.1.2.

Segregating PAM into NEWT stages is rather straightforward and does not require any restructuring of the original algorithm. There are two main methods in the original algorithm and we simply create a NEWT stage for each of them. This is illustrated on Figure 6.2 and the resulting stages are:

- Init - Defines the global state variables and randomly selects k objects as the initial medoids for the clusters.
- Check Ending Condition - Checks whether the medoids changed from the previous iteration. If they did not then move to 'Stop', otherwise continue to 'Divide Points'.

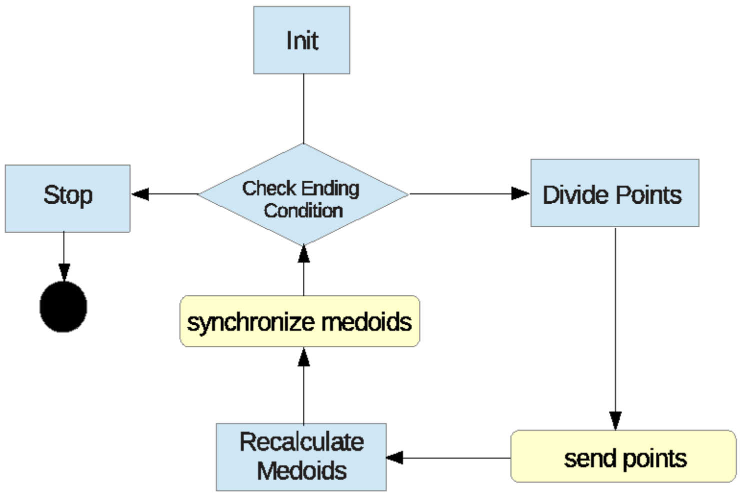

Figure 6.2: Structure of PAM under the proposed model. [6]

- Divide Points - Calculates which medoid is closest for each of the objects in the data set and synchronizes objects that should be moved between clusters.
- Recalculate Medoids - Calculates the new center element - medoid for each of the clusters and each other process of the change.
- Stop - The final stage that completes the clustering process by writing the results to disk.

These examples demonstrate when to split sequential programs into multiple NEWT stages. The most obvious reason is the synchronization requirement. Every time data needs to be received from other concurrently working processes, a barrier is required. In NEWT, communication barriers are performed between stages and thus the original algorithm has to be split into different methods every time that happens.

The second reason arises when the algorithm contains dynamic branching, such as iteration ending conditions, when there are multiple different actions that can be taken that lead to different methods being executed. Another case can be seen from the CG 'Start of Loop' stage, which is executed repeatedly and serves as a loop base for this algorithm. It is a separated stage from 'End of Loop' stage because it must initially be executed separately.

The barriers between NEWT functions are automatic and can cause unnecessary overheads if no actual data synchronization is required. Thus, it would be a straightforward optimization to combine stages when no messages are sent between them. For example, the PAM ’Check Ending Condition’ and ’Divide Points’ would be merged in this fashion.

6.2.2 Performance comparison

A series of performance measurements were performed on the framework to verify that this approach works. The previously described algorithms PAM and CG were chosen as use cases. Both algorithms were implemented in NEWT and their performance was compared to their respective BSPonMPI implementations. BSPonMPI was previously determined (section 6.1.5) to perform as good as MPI for the given algorithms and thus would provide a good comparison for BSP frameworks.

We used Amazon EC2 cloud for the benchmarking environment. A cluster of 16 Amazon’s Standard Extra Large (m3.xlarge) instances was created. Each node had 15 GB of memory and 8 EC2 Compute Units (4 virtual cores) of computing power. They were running Ubuntu Server 12.04 operating system and Hadoop YARN 2.2.0 was installed.

Two types of performance experiments were performed. First we measured the scalability of the algorithm implementations and then we assessed the overhead caused by the fault tolerance procedures of NEWT. In the scalability trials, each of the algorithms was given a fixed input size. A sparse system of 125 million linear equations for CG and 250 thousand objects across 64 clusters for PAM. Only the number of processes p was varied in these experiments to be able to calculate speedup. The results of these experiments are provided in Table 6.4.

It is important to note that these results (table 6.4) also include an approximately 14 second overhead that is induced by YARN for initialization and allocation of process containers, which can not be avoided when running applications in YARN clusters.

In the case of CG, the scaling of NEWT is initially (2-16 cores) better, but starts to decline afterwards (32 and 64 cores). This is somewhat expected because the communication part of the application starts to outweigh the computation part when the number of processes increases and MPI (which BSPonMPI uses internally) is very well optimized for such cases. It indicates that NEWT barrier synchronization should be improved.

	conjugate gradient			k-medoids clustering	
p	NEWT	BSPonMPI	p	NEWT	BSPonMPI
1	4476	4616	1	1889	1873
2	2225	2415	2	1248	1172
4	1245	1221	4	646	601
8	697	689	8	339	330
16	350	346	16	203	185
32	227	219	32	150	153
64	240	207	64	122	151

Table 6.4: Runtime (in seconds) comparison between NEWT and BSPonMPI. [6]

The sequential version (1 process) of CG structured according to the NEWT's model consistently outperformed the sequential MPI implementations. Our current working assumption is that it is related to the Java Virtual Machine (JVM) being able to optimize the code better when certain algorithms are structured according to NEWT's model.

The scaling of PAM NEWT implementation is quite close to the BSPonMPI implementation. It suggest that restructuring the algorithm for NEWT does not impose a significant overhead. In addition, NEWT produced significantly better result than BSPonMPI when the number of cores was increased to 64.

6.2.3 Fault tolerance

Periodic checkpointing was enabled in the second performance experiment to be able to measure the overhead of storing checkpoints in HDFS. The Figures 6.3 and 6.4 show three different time lines of the implemented algorithm execution, which are:

1. Default time line with checkpointing disabled
2. Time line when checkpointing is enabled but no faults happen
3. Time line when checkpointing is enabled and a critical fault happens

To be able to simulate machine or network failure in the third time line, one of the Amazon virtual machines was shut down approximately in the middle of the expected running time. An extra cloud instance was added

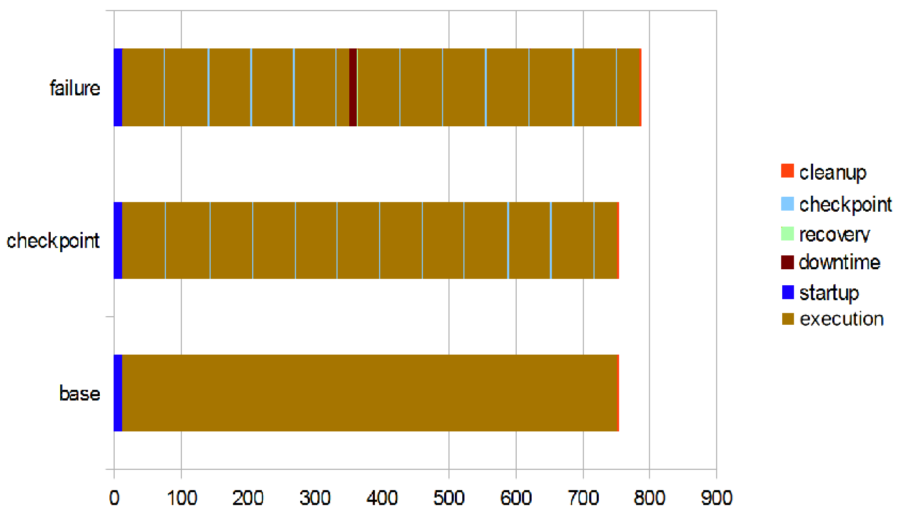

Figure 6.3: PAM performance after enabling checkpointing and on node failure. [6]

to the cluster where the failed task could be moved to. This experiment was executed in a cluster consisting of 16+1 Amazon m1.medium (3.75 GB RAM and 1 virtual CPU core) instances.

Figure 6.3 displays the three different time lines for the PAM fault tolerance experiments. Creating PAM checkpoints every minute had a negligible effect on the whole runtime since the PAM state is very small and storing it on HDFS takes only a fraction of the whole iteration runtime. The PAM state consist of floating point coordinates of the points in each of the clusters and all currently incoming messages. The size of a checkpoint was under one megabyte on average, which was much less than the entire address space of the program. It took approximately 30 millisecond on average to store a single checkpoint to HDFS was.

In the third time line, there was approximately a 10 second period of downtime after one of the nodes failed. It consists of the time it took for YARN to recognize the failure and to allocate a replacement container. After the container was made available, the failed NEWT process was started in it. To restart the whole computing process, each of the process read their current state from the latest checkpoint and all the communication channels were reestablished between them. These last steps took under a second in total.

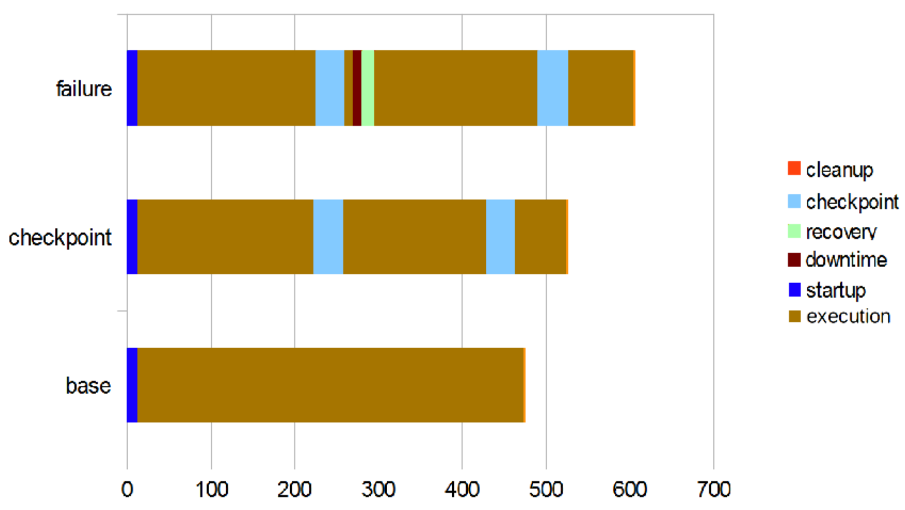

Figure 6.4: CG performance after enabling checkpointing and on node failure. [6]

Figure 6.4 shows the time lines for the three Conjugate Gradient fault tolerance experiments. The size of the checkpoints was 400 megabytes, which was significantly larger than PAM checkpoints. For this reason the frequency of their creation was reduced to three and a half minutes. Storing a 400 megabyte checkpoint took more than 30 seconds and reading them from HDFS took approximately 15 seconds.

Table 6.5 contains the average times for the different types of overhead the framework imposes for both CG and PAM.

	CG	PAM
startup	12	12
checkpoint	34.75	0.1
downtime	11	12
recovery	15	0.3
cleanup	2	2

Table 6.5: Average overhead times (in seconds) imposed by NEWT for CG and PAM. [6]

6.3 Discussion

To analyze the available BSP implementations we compared them to MPI by applying two benchmarking applications, which parallelized using several MPI and BSP implementations. The results of this analysis show that BSP based implementations work at least as well as the MPI based ones and much better than Hadoop MapReduce. As the chosen applications represent the typically used non embarrassingly parallel scientific computing algorithms, it demonstrates that the BSP distributed computing model is a good choice for solving scientific computing problems. We also believe that BSP frameworks have strong potential to offer the same advantages as MapReduce frameworks, like near-automatic parallelization and fault tolerance.

However, the current BSP frameworks and libraries still leave much to be desired. Hama, for example, has problems with performance and BSPlib is outdated and very difficult to use on latest hardware. Libraries such as BSPonMPI are efficient, but like MPI, they do not provide automatic fault tolerance and do not significantly simplify the creation of BSP applications.

BSP frameworks that are based on Google Pregel, like Apache Giraph and Stanford GPS, are designed for graph computations, and while it is certainly possible to adapt non graph algorithms to them, it is not a straightforward task. Still, we have shown that BSP is a good choice for solving scientific computing problems as the performance of the most used BSP libraries is closely comparable to MPI libraries and much better than for Hadoop MapReduce. [17]

These results demonstrates that the Bulk Synchronous Parallel distributed computing model is a good choice for solving iterative scientific computing problems. We also believe that frameworks based on the BSP model have strong potential to offer the same advantages as the MapReduce model, like automatic parallelization and fault tolerance.

To counter these drawbacks we presented a BSP-inspired parallel programming model that enables transparent stateful fault tolerance through checkpointing. To validate the usefulness of the proposed model, we created a distributed computed framework, called NEWT.

NEWT supports a larger range of applications than the current BSP implementations and utilizes Hadoop YARN to perform automatic checkpoint/-restart of programs. We implemented two very different computation kernels on the framework and showed that it performs adequately for coarse-grained algorithms.

However, our results also show that the current barrier synchronization implementation could still be optimized for better support of very fine-grained algorithms. We compared the NEWT checkpointing time requirements to Berkeley Lab Checkpoint/Restart (BLCR) [75] approach and determined that writing NEWT checkpoints to HDFS is as fast as writing BLCR checkpoints to local storage.

6.4 Summary

This chapter outlined our work in evaluating the approach of using Bulk Synchronous Parallel distributed computing model as an alternative to the MapReduce model. We measured the performance of current Java based BSP implementations in comparison to both Hadoop MapReduce and MPI and have shown that the performance of BSP based frameworks is close to MPI, which we consider to be the de facto choice for parallel programming when performance is the main goal.

We identified a number of problems with the current BSP implementations. To solve these issues we proposed a new fault tolerant BSP-inspired parallel programming model and framework NEWT for iterative computations which can run in existing Hadoop Yarn clusters. We described its programming model, illustrated how to adapt scientific computing algorithms to it, and compared its parallel performance to BSPonMPI, which showed the best performance among the pre-existing BSP implementations. Runtime experiments showed that the proposed NEWT framework has similar performance to BSPonMPI.

Next chapter outlines our approach for choosing the most suitable distributed computing framework for a given algorithm when there are many different choices available and more being created every year.

CHAPTER 7

CHOOSING THE MOST SUITABLE DISTRIBUTED COMPUTING FRAMEWORK

We have described three different approaches for adapting scientific computing algorithms to distributed computing frameworks in the previous chapters. In the course of this work, we adapted algorithms to a number of different MapReduce and BSP frameworks, compared their performance to the de-facto parallel programming library MPI to analyze how suitable these frameworks are for solving computation intensive scientific computing problems and evaluated whether they simplify the process of parallelizing and scaling scientific computing algorithms.

We define suitable distributed computing framework as a framework which fulfills our original expectations for adapting scientific computing algorithms to distributed computing frameworks. These expectations (described in Chapter 1.1) are ease of parallel programming, ease of debugging, automatic fault tolerance, parallel efficiency and scalability. When comparing the suitability of different distributed computing framework, the most important factor for scientific computing algorithms is parallel efficiency.

Our conclusion from all of this work is that there is no single approach or distributed computing framework that is suitable for every type of scientific computing algorithm. Hadoop MapReduce is generally more suitable for embarrassingly parallel algorithms when the intermediate data does not fit into the memory of the cluster. Because we have to use some kind of distributed file system to store the data in that case so there is no additional overhead from using file based HDFS to store intermediate data.

Alternative MapReduce frameworks (such as Twister or Spark) are more suitable for iterative algorithms when the intermediate data fits into the total memory of the cluster. BSP based frameworks are designed for either graph processing algorithms (Giraph, Stanford GPS) or also iterative algorithms (HAMA, NEWT). MPI implementations are required for more complex algorithms that require asynchronous or spontaneous communication. Furthermore, existing distributed computing frameworks are constantly updated and new frameworks are introduced every year, which may solve some of the issues we face, but their suitability and parallel efficiency should be evaluated in comparison to existing parallel programming libraries or distributed computing frameworks).

In recent years, new distributed computing platforms (such as Hadoop Yarn and Aneka [23]) have emerged which enable switching between different distribute computing frameworks on-demand. Hadoop YARN separates computer resource utilization, task scheduling and execution from the default MapReduce framework and allows other distributed computing frameworks to be used instead. Aneka allows to extend local clusters with cloud computing resources and execute tasks on different distributed computing frameworks depending on the characteristics of the tasks. Both of these distributed computing platforms were described in chapter 2.

These platforms simplify using the most suitable distributed computing framework for a given algorithm as it allows users to avoid having to set up many distributed computing environments and moving data and tasks between them. However, choosing between different distributed computing frameworks is still left up for the user to decide – which can be a very difficult task. One would have to have an in-depth knowledge of the involved algorithm, each of the available frameworks and the parallelization methods used by those frameworks. Often it would require implementing the given algorithm in a selection of those frameworks and comparing the results and setting up and executing the implementations in a real cluster.

It might also be possible to use previously existing benchmarks to evaluate which of the frameworks would be more suitable for the algorithm under investigation. But that would require finding a benchmark which is very similar to the given algorithm and which is also implemented for each of the distributed computing frameworks that are under consideration. But it is possible that there are no such benchmarks, or that they are similar but use different data types, can not be used with larger amount of data, or are implemented only in some of the available frameworks. It is a potential approach, but we can not assume it is always available and directly applicable.

We decided to make the process of choosing the most suitable distributed computing framework a more straightforward task, as there would otherwise be little benefit from having all these different frameworks available. This approach should not require investigating all the available frameworks in detail or extensive programming and debugging effort to adapt the algorithm for each of them.

To achieve this, we decided to create a Dynamic Algorithm Modeling Application (DAMA) for simulating the parallel structure of scientific computing algorithms and implemented it on a number of available distributed computing frameworks in the Hadoop ecosystem which are widely used or which we have previously evaluated to have a reasonable parallel efficiency in comparison to MPI. We propose a methodology which applies DAMA to simplify choosing the most suitable distributed computing framework for a given algorithm.

The following sections describe DAMA and the proposed methodology in more detail. We also verify that the modeling methodology correctly predicts which distributed computing model and its implementation is more suited for a given algorithm.

7.1 Dynamic Algorithm Modeling Application

Dynamic Algorithm Modeling Application (DAMA) is a generic distributed computing application, which can be reconfigured to model the structure of different algorithms. DAMA follows the single program, multiple data (SPMD) parallelism model and has been implemented using four different parallel programming solutions: Apache Spark, Hadoop MapReduce, Apache Hama and MPIJava.

After configuring DAMA to model the structure of a given algorithm, it can be used directly as an approximate benchmark for this algorithm on any of the supported frameworks or libraries. This process does not require any programming or code debugging steps and can thus significantly simplify the process of estimating the performance of the algorithm on different frameworks as we can avoid implementing the given algorithm on each of the available frameworks and using the implementations as benchmarks.

To be clear, all these steps are still going to be required once the target distributed computing framework is chosen. The goal of DAMA is simply to postpone these steps until it is known which specific framework should give the best result and thus greatly decrease the scope of work that has to be done.

We feel that this kind of approach is critical as the number of available distributed computing frameworks continues to increase in the Hadoop ecosystem and also outside of it. In addition, Santenu Jha et al. [62] also concluded (as covered in the state of the art chapter in section 2.3) that current benchmarks do not cover all facets of scientific computing and Big Data algorithms and new benchmarks should be created to cover them. To support additional frameworks, we need implement DAMA once on them and then we can evaluate their performance for any scientific computing algorithm which DAMA can model.

As previously mentioned, DAMA is a generic distributed computing application, which parallel structure can be changed to model different algorithms. In essence, it is an iterative application which executes a user configured set of communication patterns and computational kernels at every iteration. Communication patterns represent data synchronization tasks and allow us to simulate their effect on the performance of parallel applications. Kernels represent computational operations and are used to simulate the computational complexity of parallel applications. The ratio between communication and computation is one of the most influential parameters on the scalability of parallel applications so it is important to model both of them as accurately as possible.

Almost everything in DAMA is fully configurable by the user, such as the number of iterations, type size and amount of input data, what computational kernels are executed and how much data is synchronized at every iteration. All the available configuration options are outlined in Table 7.2 and Section 7.2.1 describes how to specify these configuration values using four algorithms as examples. The execution flow of DAMA is outlined using pseudo-code in Algorithm block 19

DAMA first checks what communication interface (representing different distributed computing frameworks or parallel programming libraries) to use, generates input data objects of specific size, type and amount and starts the iterative execution process. At each iteration, DAMA checks

Algorithm 3 Pseudo-code for the main operation cycle of DAMA.

```
$init(COMMUNICATION\_INTERFACE)$
$importConfiguration(CONF\_FILE)$
$data = genData(DATA\_OBJECTS, DATA\_TYPE)$
**if** $!DATA\_GENERATED\_LOCALLY$ **then**
  $distributeData(data)$
**end if**
$it = 0$
**while** $it < iterations$ **do**
  **if** $it > 0$ & $STORE\_INTERMEDIATE\_DATA$ **then**
    $data = readIntermediateData(it - 1)$
  **end if**
  $data = callComputationKernel(it, data)$
  $data = syncData(data)$
  **if** $STORE\_INTERMEDIATE\_DATA$ **then**
    $storeIntermediateData(it, data)$
  **end if**
  $processResults(it)$
  $it = it + 1$
**end while**
```

whether intermediate data should be stored and read from the disk, applies computational kernel on the data and performs a set of data synchronization operations which have been configured. Once the specified number of iterations have been completed, the results are either stored in the file system or a number of results are printed out to the standard output.

The currently implemented communication patterns and kernels are described in the following subsections.

7.1.1 Communication patterns

Communication patterns allow us to simulate the effect of data synchronization tasks on the performance of parallel applications. A good representation for typical communication patterns in scientific computing applications are the communication methods defined in the MPI standard [11]. They cover most of the structured communication patterns used in parallel applications when using the single program, multiple data (SPMD) parallelism model. To be clear, SPMD is the only parallelization model that DAMA currently supports.

The data synchronization patterns that we decided to use in DAMA are:

1. **Single Message** – One process sends data to a single other process. This is the basic data synchronization primitive for synchronizing data between concurrently working processes.

2. **Broadcast** – One process sends data to all other processes. Every process gets the same copy of the data.

3. **All-to-All** – All concurrently working processes send data to all other processes. Every process sends a different slice of the data to different processes.

4. **Scatter** – One distributes data between all other processes. Every process gets a different slice of the data.

5. **Gather** – All processes send data to a single process to be collected in one location. In addition there is **All-Gather** pattern, where the results are communicated to all concurrently working processes. Naive way would be to perform **All-Gather** as combining Gather with Scatter or using All-to-All, but there are more efficient ways to implement it as a separate pattern.

6. **To-Neighbors** – All processes send some of the data to a few of the other processes. The processes which to send data to are decided based on a neighborhood in an array, matrix, cube or higher dimensional structured grid.

7. **Reduce** – All processes send data to a single process to be collected in one location. Data objects are aggregated based on a given aggregation function. Similarly to All-Gather, there is also a All-Reduce pattern, where the results are made available to all concurrently working processes.

There are other communication patterns that are utilized in parallel applications, but these are sufficient for a large portion of the scientific computing applications. When configuring DAMA, user has to specify how many times each of these synchronization patterns should be used at every iteration. There are also additional configuration options (outlined in Table 7.2) for each patten to refine how they are applied.

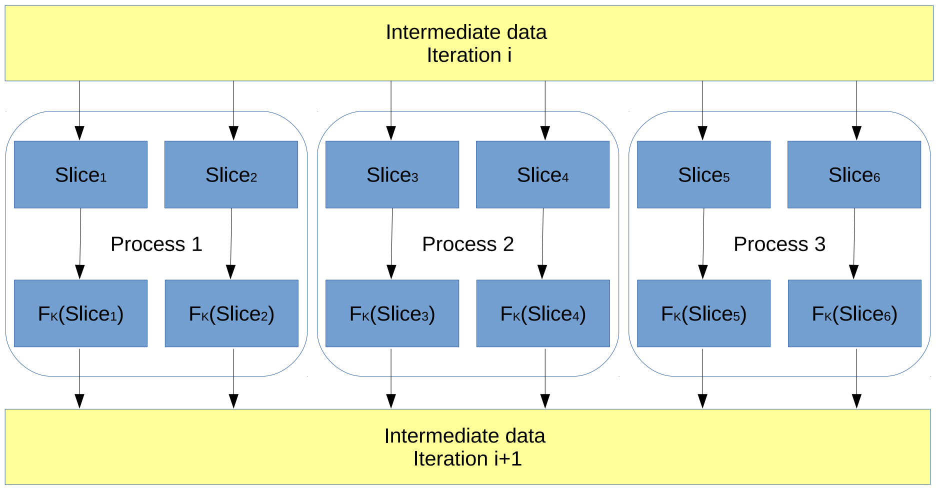

Figure 7.1: Applying computational kernels to data slices.

7.1.2 Kernels

In DAMA, kernels represent the computational operations that are performed in parallel applications and allows us to simulate the computational complexity of algorithms. Figure 7.1 illustrates how computational kernels are applied in DAMA.

Input data of the application (which consists of a list of data objects) is divided into S slices which in turn are divided between N concurrently working processes. At each iteration, a kernel function F_k is applied on every slice, which takes a list of data objects as input and also returns a list of objects. What specific function is applied on the data depends on the kernel that is specified in DAMA configuration.

There are two kernel modes, block kernel mode, when kernel is applied on a slice of data and the basic kernel mode, when kernel is applied on every data object separately. In block kernel mode, $b + n^p$ operations are performed where n is the number of data objects in a slice, p is kernel power and b is kernel base. Table 7.2 illustrates how to specify kernel type, base and power in the DAMA configuration.

The currently implemented kernels are:

1. **Double Distance Kernel** – Calculates Euclidean distance between double numbers.

2. **String Distance Kernel** – Calculates distance between strings.
3. **Double Dot Kernel** – Calculates dot function.
4. **Double Random Kernel** – Generates random double values.
5. **String Random Kernel** – Generates random strings.

It is relatively easy to add additional user defined kernels to DAMA by extending *ee.ut.scicloud.benchmark.kernel.Kernel* class, implementing *Datatype[] execute(Datatype[])* and *Datatype execute(Datatype)* functions for the data types (such as Double, String, or Integer) that the kernel should be applied on and specifying the name of the class as the computational kernel in the DAMA configuration.

The next section describes our proposed methodology which outlines how to apply DAMA for choosing the most suitable distributed computing framework for a given algorithm.

7.2 Methodology

To model scientific computing algorithms we need to consider what algorithm characteristics affect the performance of their parallel implementations. Examples of such characteristics are structure and size of the input data, data synchronization patterns, the ratio between parallelism and communication costs and memory intensity.

These characteristics are used to configure DAMA to model the parallel structure of a given algorithm. DAMA can then be used directly as a benchmark on each of the supported distributed computing frameworks to estimate what would be the performance of the algorithm on each of them.

To clarify, DAMA does not simulate the performance of distributed computing frameworks. User still needs to perform benchmarking experiments in a distributed computing cluster which supports the respective frameworks. The goal of the benchmarking is to calculate parallel efficiency for each of the distributed computing frameworks under investigation and also MPI for comparison. At this point, we assume the user has already filtered out frameworks which are not suitable for their needs based on the other expectations (Ease of parallel programming, Ease of debugging, Automatic fault tolerance) which were defined in Section 1.1 and can choose the most suitable framework based on the calculated parallel efficiency values.

The methodology we propose for the process of choosing the most suitable distributed computing framework for a given algorithm consists of the following steps.

1. Identifying algorithm characteristics.
2. Configuring DAMA to model the parallel structure of the algorithm.
3. Running benchmarking experiments in a cluster.
4. Choosing the most suitable distributed computing framework.

Each of these steps are described in detail in the following subsections.

7.2.1 Identifying algorithm characteristics

The first step in the modeling process is to describe the structure and behavior of the algorithm using a set of pre-defined algorithm characteristics. These characteristics are used as an input for the DAMA application to simulate the parallel implementation of the algorithm on different distributed computing frameworks. The most important characteristics are those that affect the parallel efficiency of the algorithm after it is adapted to one of the distributed computing frameworks.

For example, the number of iterations that must be performed to get the final result will directly affect the efficiency of the parallelization if any amount of data must be synchronized between concurrently working processes at every iteration. Our initial classification (Outlined in chapter 3) only took the number of iterations into account because this was the most influential parallel characteristic for Hadoop MapReduce applications.

But there are other characteristics that need to be taken into account when other distributed computing frameworks are also considered. Such as communication patterns, size of synchronized data, balance between communication and computation, what computation methods are performed at every iteration of the algorithm, etc. To be able to identify such characteristics, the algorithm must be analyzed from the parallelization viewpoint, which can be a challenge for users who are not experts in parallel programming.

Some characteristics are simple to identify, such as the average number of iterations. But specifying which type of synchronization methods or communication patterns should be used inside each iteration can be a complex task for users who are not proficient with parallel computing.

To retain the balance between communication and computations we also need to model the computations that are performed by the algorithm. We use kernels to define what computational functions that are applied on data. To illustrate the process of analyzing algorithms to identify their characteristics, we demonstrate how to analyze four algorithms in this manner in the following subsections.

Integer Factorization

This algorithm applies trial division to find the set of prime number factors for an integer. It has already been described in section 3.1.4, but here we illustrate how to analyze this algorithm to find its distinctive parallel characteristics.

First step of this process is to identify when data synchronization operations need to be performed. This algorithm basically consists of a loop where all possible factors of an integer N are tested to see if they are its divisors. It can easily be parallelized by dividing the range of potential divisors $\sqrt{N}$ into K sub ranges, giving each subrange to different processes and performing K loops concurrently. Once all the loops finish, we need to collect the divisors together in one place and combine them together into a sorted list of prime numbers. The computational complexity of finding the prime factors is $O(\sqrt{N})$ where N is the value of the number to be factored.

Figure 7.2 illustrates the computation steps of Integer Factorization from this perspective. We can see that there are two moments when data needs to be synchronized. First one is required to divide the subranges between concurrently working processes, so each of them know from which range to check divisors from. When the computation is completed, all the results have to be collected together into one location using a Gather data synchronization pattern to be able to aggregate the results.

Depending on which framework this algorithm is adapted to, it may be possible to avoid the first synchronization operation at the start of the computation by providing the starting ranges information to each concurrent task in a more statical manner. For example, if the so called worker tasks need to be spawned explicitly before the computations are started, it

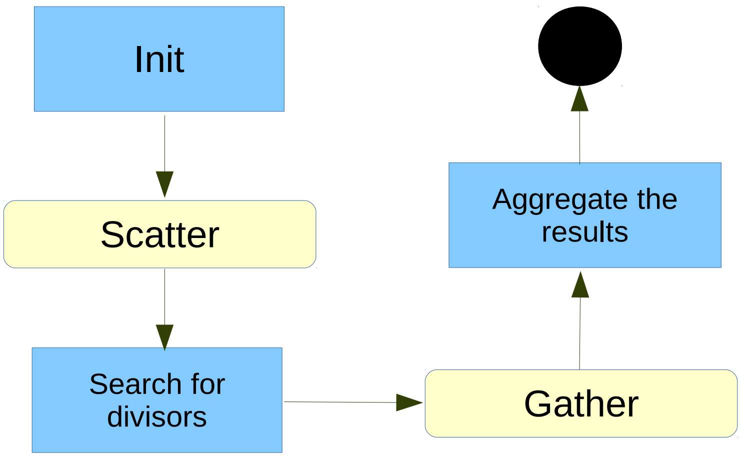

Figure 7.2: Identifying data synchronization breaks in the Integer Factorization algorithm.

may be possible to directly pass on the range each one is supposed to process. But whether this is possible depends on which distributed computing framework is used.

The data that needs to be synchronized is very small, as it is limited by the number of total prime divisors for the given number. Computations consist of generating random number and finding the remainder after integer division. There are no iterations, everything can be computed in one go. Algorithm is not memory intensive, as input is very small and intermediate results do not need to be kept in memory.

Partitioning Around Medoids (PAM)

Partitioning Around Medoids (PAM) [64] is a clustering algorithm which has already been described in section 3.1.2. Its goal is to divide a set of N objects into k different clusters. At the start, k random objects are assigned as the initial medoids (center element) for clusters. An iterative loop is started which consists of two separate tasks. First task looks through the input dataset one object at a time, calculates its distance from each of the medoids and assigns the object to the cluster of the closest medoid. Computational complexity of this task is $O(N * k)$.

Second task recalculates the medoid for each of the clusters by finding the object with the least total distance from all the other objects in the same

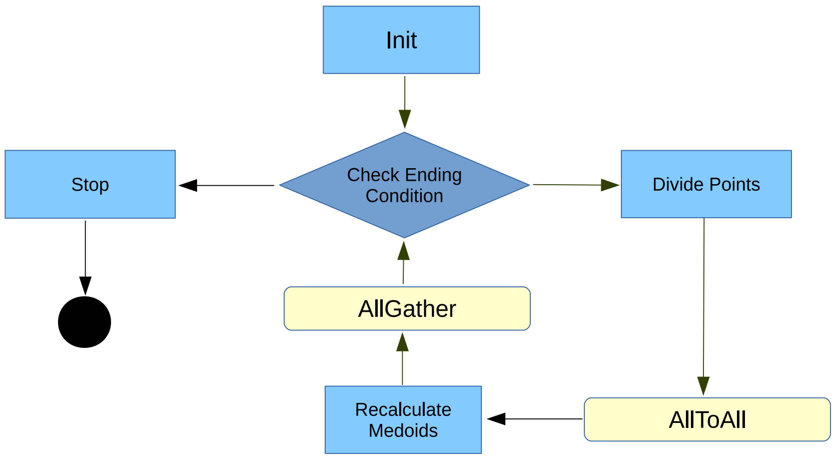

Figure 7.3: Identifying data synchronization breaks in PAM.

cluster. The computational complexity of this task is $O(N^2)$. While the average size of the cluster is N/k, there is no guarantee, that the clusters are of equal size, which would be the best case situation with a complexity of $O(k*(N/k)^2) = O(N^2/k)$ These two tasks are repeated until the medoids of the clusters no longer change. It can take tens of iterations to converge, depending on the size and number of clusters. Convergence may even fail in rare cases.

Figure 7.3 illustrates the computation steps of PAM from this perspective. We can see that there are two moments inside each iteration when data needs to be synchronized. After dividing objects between clusters, all objects assigned to the same cluster must be located together to be able to find the new cluster centers. Thus we need to group objects by their closest medoid and synchronize objects that need to be moved between clusters using All-to-All operation. It may require synchronizing almost all the objects between processes at first, but the number of objects that move between clusters is generally reduced by every successive iteration. After medoids are recalculated for each of the clusters, they also need to be communicated to all other processes using All-Gather operation, but there are only k medoids so the amount of synchronized data is not large here.

PAM is also computationally intensive algorithm mainly because of the medoid recalculation task inside each iteration. It requires pairwise comparison between every object in a cluster, which is performed for every cluster at every iteration.

All the input data needs to be kept in memory until the end, but because PAM can not handle large amount of input data it should not really be considered memory intensive. Adding more CPU resources is always more important than adding more memory resources for PAM and thus it is more computationally intensive algorithm.

The basic computational kernel of PAM consist of finding pairwise distances between objects. How exactly the distance is calculated depends on the type of the object. Euclidean distance can be used for example when dealing with points in n dimensional space. Hamming distance could be used in case of strings.

Clustering LARge Applications (CLARA)

In comparison to PAM, Clustering Large Applications [64] (CLARA) only clusters small random subsets of the dataset to find candidate medoids for the whole dataset. CLARA is described in more detail in section 3.1.3.

CLARA algorithm consists of two main tasks and no iterations. First task chooses t random subsets of size s from the input data of size n as samples, applies PAM on each of the sample, and outputs t set of objects as a candidate medoidsets, each consisting of k objects. The complexity of this task is $O(t * s^2)$.

The second task finds the quality for each of the t medoidsets, by calculating the sum of distances for every object (from the whole input dataset) from its closest medoid for each of the t medoidsets. The computational complexity of this task is $O(n * t * k)$.

The candidate set of medoids with the smallest sum of distances is chosen as the best clustering. The computational complexity of this task is $O(t)$.

Figure 7.4 illustrates the structure of the CLARA algorithm. There are three data synchronization steps. First involves All-To-All operation to randomly shuffle all the data between all the computation nodes. This shuffle together with the All-To-All operation can be avoided if it is known beforehand that data is randomly distributed and randomly sorted. Otherwise, the distributed sampling will not be able to produce samples that represent the whole dataset.

The second step involves All-Gather operation to collect t medoidsets (total of $t * s$ values) on each of the computation nodes, which will then be used to cluster each of the data object calculating the quality of each of the

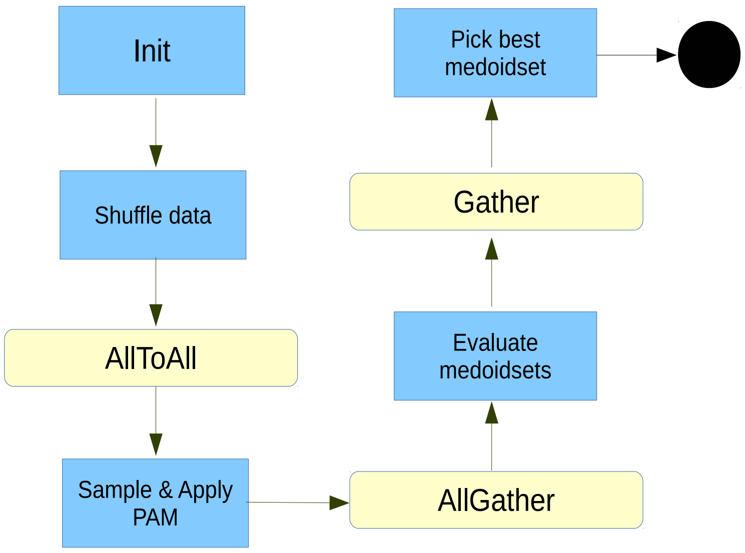

Figure 7.4: Identifying data synchronization breaks in CLARA.

medoidsets. The third synchronization step involves gathering t quality values together in one location and choosing the best result as the final clustering configuration.

The ratio between computations and communication in quite balanced in CLARA. The number of samples and their size affects the time it takes to find candidate medoidsets. The quality of the medoidsets is evaluated using all the input data objects.

All the input data needs to be kept in memory until the end of the work and does not need to be modified or moved after the initial randomization. The basic computational kernels are applying PAM on the sample datasets, finding pairwise distances between objects and summing the final values together when choosing the best medoidset. To simplify the modeling we can replace the PAM kernel with applying pairwise distances as it is also the main kernel of PAM.

Conjugate Gradient linear system solver (CG)

Conjugate Gradient algorithm has already been described in more detail in section 3.1.1. It solves linear systems in the matrix form:

$$Ax = b, \tag{7.1}$$

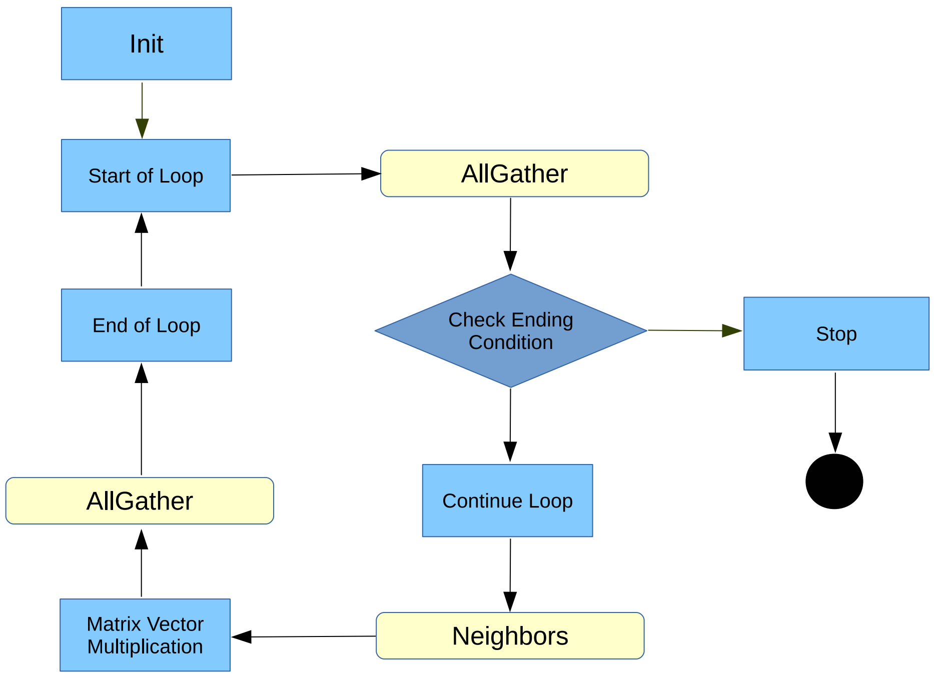

Figure 7.5: Identifying data synchronization breaks in CG.

where A is a matrix consisting of the coefficients $a_{11}, a_{12}, ..., a_{mn}$ of the system, b is a known vector consisting of constant terms of the system $b_1, b_2, ..., b_m$ and x is the solution vector, made up of the unknowns of the system $x_1, x_2, ..., x_n$.

The input matrices are large but can typically be fit into the collective memory of computer clusters. Most of the data does not need to be modified or moved across the iterations.

Figure 7.5 illustrates the computation steps of CG from this perspective. We can see that there are three moments inside each iteration of CG when data needs to be synchronized. One requires synchronization between neighbors (To-Neighbors) to distribute overlapping vector values before the matrix-vector multiplication. Two others require data communication between all nodes in the cluster (using All-Reduce, All-Gather, or All-to-All) to calculate dot product of distributed vectors. The amount of data that needs to be synchronized is very small. The first operation requires synchronizing a small overlap of a vector and the distributed Dot-products require sending only a single value from each process.

CG computational kernels are matrix-vector products and dot-products. To simplify the modeling process, matrix-vector product can be replaced with a number of dot-product operations as long as the number of basic operations is kept the same (Number of dot product operations is equal to the number of rows in the matrix) and the sparsity level of the input matrix is taken into account. Matrix-vector multiplication has the most influential computational complexity out of all the operations performed each iteration, which is $O(n)$, where n is the number of non-zero elements in the input matrix.

The computational complexity of the algorithm is not high and the performed task at every iteration is relatively small which means CG is not computationally intensive. How many iterations are executed depends on the required accuracy. Approximately 4 iterations would result in basic accuracy and each following iteration improves it. Because there are three different data synchronization operations every iteration, CG is considered a communication intensive algorithm.

	PAM	CG	CLARA	Integer Factorization
Balance between Communication and Computation	Computation	Communication	Balanced	Computation
Number of iterations	$\sim 10 - 40+$	$\sim 4 - 15+$	1	1
Complexity (single iteration)	$O(N^2)$	$O(n)$ - n: non-zero elements in A	$O(n * t * k)$ n: objects, t: samples, k: clusters	$O(r)$ r: Random trials
Communication patterns	All to All, AllGather	AllGather, 1D Neighbors	All to All, AllGather, Gather	Scatter, Gather
Size of synchronized data between iterations	Up to 100% (decreasing in time), k	nr_of_slices (All Gather) and 25% (1D Neighbors)	100% if randomize data, $samples * k$	Nr of factors
Memory intensity	No	Yes	Balanced	No
Computation Kernel	Pair-vise distance	Dot-product	Pair-vise distance & Sum	Integer division
Communication type	Synchronous	Synchronous	Synchronous	Synchronous
Dynamic or Static intermediate data	Dynamic	Static	Dynamic & Static	Static

Table 7.1: Parallelization characteristics of the example algorithms.

Summary of the algorithm identification

A selection of algorithm characteristics that were identified and their specific values for the four algorithms are provided in Table 7.1.

User has to identify what are the specific values of such characteristics to model the parallel implementation of a given algorithm using DAMA. It means that at least basic knowledge of algorithm parallelization process is required to follow this methodology. As a result this approach is somewhat limited, but it would still be easier and require much less actual effort in comparison to implementing the given algorithm on each of the potentially suitable distributed computing frameworks.

7.2.2 Configuring DAMA

Once the algorithm characteristics have been identified for an algorithm, DAMA needs to be configured to model its structure. User has to specify a number of configuration values in a configuration file and provide the location of the configuration file as an argument to the DAMA program.

Some of these configuration values are directly related to the previously identified algorithm characteristics, such as the number of iterations, communication patterns or kernels and their options. Other configuration values more related to benchmarking, such as the size and type of the input data. These configuration options are outlined in Table 7.2 with example values for CG, PAM, CLARA and integer factorization algorithms.

Once DAMA has been configured, it can be used directly as a benchmark to estimate the parallel performance of the modeled algorithm on all the supported distributed computing frameworks and MPI

7.2.3 Benchmarking

DAMA does not simulate the parallel efficiency of the modeled algorithm. It is a generic benchmark which models the parallel structure and implementation of an algorithm. Its main advantage is that it can be used directly as a benchmark for the modeled algorithm on any of the supported distributed computing frameworks (Spark, Hadoop, and Hama) and also MPI. DAMA can also used as a sequential single process Java benchmark.

The benchmarking experiments must be performed in a real distributed computing clusters to get meaningful results for estimating which of the

Configuration	CG	PAM	CLARA	IF	Description
DATA_SLICES	32	32	32	32	How many separate blocks will data be split into. Slices are divided between concurrently working processes and kernels are applied to each slice separately.
NUMBER_OF_ITERATIONS	12	31	1	1	How many iterations will be performed.
DATA_TYPE	Double	Double	Double	Double	Type of the basic data objects such as Double, String or Long.
DATA_OBJECTS	3.2 mil	32000	32 mil	32	How many data objects will be generated.
DATA_OBJECT_SIZE	1	1	1	1	Some data objects types (such as Strings) can be of dynamic size.
COMPUTATION_KERNEL	Double Dot	Double Distance	Double Distance	Random Double	What computation kernel is applied to every data slice.
COMPUTATION_KERNEL_-TYPE	Kernel	Block Kernel	Kernel	Kernel	What is the Kernel model. Block kernel is applied on the whole data slice, normal kernel is applied on every data object separately.
COMPUTATION_KERNEL_-POWER	1	2	1	1	Used to specify dynamic computational complexity. Actual effect depends on specific kernel implementation.
COMPUTATION_KERNEL_-BASE	1	1	$samples * clusters$	10 billion	What is the base value for the kernel. If the value of kernel base is b then this is how many operations are executed per data object.
ITERATION_SLEEP_TIME	100	100	100	100	How long should be the default wait time in milliseconds at the end of each iteration. Can be set to 0.
GATHER_FINAL_RESULTS	false	false	true	true	Whether final results should be gathered to a single (master) node
STORE_FINAL_RESULTS	true	true	true	false	Whether final results should be stored on file system.
PRINT_FINAL_RESULTS	false	false	false	true	Whether final results should be printed out to standard output.
PRINT_FINAL_COUNT	0	0	0	1	How many of the final results should be printed out if printing is enabled.
STORE_INTERMEDIATE_ DATA	false	false	false	false	Whether intermediate results should be stored in file system at the end of each iteration
DATA_GENERATED_LO-CALLY	true	true	true	true	Whether data should be generated locally on each of the nodes. If false, data is generated on single node and scattered to others.
TO_NEIGHBORS	1	0	0	0	How many TO_NEIGHBORS operations will be executed at every iteration.
TO_NEIGHBORS_ELEMENTS	100 objects	0	0	0	How many data objects will be synchronized using TO_NEIGHBORS.
TO_NEIGHBORS_NR	2	0	0	0	How many neighbors will data be synchronized with.
GATHER	2	1	2	0	How many GATHER operations will be executed at every iteration.
GATHER_DATA_ELEMENTS	nr. of processes	k	$samples * k, k$	0	How many data objects will be synchronized using GATHER
SCATTER	2	1	1	0	How many SCATTER operations will be executed at every iteration.
SCATTER_DATA_ELEMENTS	nr. of processes	k	$samples * k$	0	How many data objects will be synchronized using SCATTER.
ALL_TO_ALL	0	1	1	0	How many ALL_TO_ALL operations will be executed at every iteration.
ALL_TO_ALL_DATA_RATE	0	100%	100%	0	What percentage of data objects will be synchronized using ALL_TO_ALL.

Table 7.2: Example DAMA configurations for modeling each of the considered algorithms.

distributed computing frameworks are the most suitable for the given algorithm. Thus, access to a cluster where the distributed computing frameworks are installed is required and the size of such cluster limits how large benchmarking experiments can be performed.

The goal of the benchmarking is to measure the runtime of the configured DAMA for each of the distributed computing frameworks and MPI and use the measurements to calculate parallel speedup. Parallel speedup of the respective distributed computing frameworks and MPI should be calculated in relation to the Java sequential runtime of DAMA, as it would clearly show how much faster the parallel execution is in comparison to a non-parallel execution.

Even after accurately modeling the structure of an algorithm, some algorithm characteristics and benchmarking variables can affect the parallel efficiency of the result. For example, the size of the input and intermediate data influences the suitability of some distributed computing frameworks, especially depending on whether the intermediate can be fit into the collective memory of the cluster. Another factor which can significantly affect the performance of distributed computing frameworks, is the number of concurrent processes. Not all distributed computing frameworks handle large number of concurrent processes with the same efficiency.

For these reasons, it is important to perform benchmarking experiments under varying cluster configurations so that the user can see how the performance of distributed computing frameworks is affected by these characteristics and take it into account. At minimum, it is important to change the size of the input dataset and the number of concurrent processes. This is further complicated by the fact that benchmarking in a real cluster can be costly in time and money, and thus users are often not interested in executing benchmarking experiments in large clusters with large amount of data. However, when this issue arises later, DAMA can be used to run additional experiments with different cluster configurations.

7.2.4 Choosing the best candidate

Once the performance benchmarking results have been collected, the most suitable distributed computing framework can be chosen based on the calculated runtime and parallel speedup values.

The calculated parallel speedup of a distributed computing framework shows how much faster this framework was able to execute the modeled algorithm in comparison to a non-parallel algorithm. The framework with the highest parallel speedup is more efficient in parallelizing the modeled algorithm and requires less computing resources than other to achieve the same result.

The best obtained parallel speedup of the distributed computing frameworks should be compared to the parallel speedup of MPI in order to analyze whether there is actually any benefit from using distributed computing frameworks. If the difference is large, it means that much more computing resources are needed to achieve the same runtime. It is hard to concretely qualify what should be the value of the best achieved speedup in comparison to MPI, but the different speedup values can be used to estimate the cost of computing resources needed to achieve the same result.

However, it is also possible that the results are not fully conclusive. Even if one distributed computing framework is technically more suitable based on the parallel speedup values, its actual benefit might be negligible over another, which the user might already be proficient with. The size of the input data can also be an issue. One framework may be more suitable when input data is smaller and not be suitable at all when it increases significantly. In such cases DAMA configuration parameters should be modified accordingly and benchmarking experiments can be repeated to investigate their effect on the performance of the modeled algorithm.

The process of analyzing the benchmarking results is illustrated is Section 7.3 using four example algorithms.

7.3 Validation

To validate the approach and to illustrate its benefits, we apply this approach on the previously analyzed algorithms and investigate the efficiency of the results in comparison to results obtained in our previous work. To be clear, we do not expect that the runtime is same both for simulated and real implementation for each of the frameworks. However, their respective parallel efficiencies should be correlated in the case of increasing the size of the input data set and also the number of concurrent processes in the cluster. Otherwise the modeled algorithm does not represent the behavior of the original algorithm.

The modeled algorithms are Conjugate Gradient (CG), Partitioning Around Medoids (PAM) and Clustering LARge Applications (CLARA) which were described in section 3.1 and which analyzed for DAMA in section 7.2.1. We configured DAMA for each of the algorithms, performed benchmarking experiments for all the supported distributed computing framework (Spark, Hadoop, MpiJava, Hama), calculated parallel speedup and compared the results to previous results obtained in earlier work when implementing the same algorithms on the same frameworks.

7.3.1 Experiment configuration

We set up a Hadoop cluster for the performance experiment where all the required distributed computing frameworks are installed. We chose Cloudera 5.7.0 CDH as the Hadoop distribution as it is easy to install and already contains MapReduce and Spark. We had to install MPI and Hama separately. We set up a cluster in local OpenStack based cloud. The Tartu University OpenStack cluster consists of HP ProLiant DL180 G6 servers, each one has: two 4-core CPU's (Xeon E5606, 2.13 GHz clock speed), 32 GB of RAM; 2x2 TB hard disks and two Gigabit NICs. The Hadoop cluster was set up on top of 4 virtual machines, each containing 13 GB of memory, 220 GB storage and 4 processor cores. Ubuntu 12.04 64 bit was used as the operating system. MPI tests were performed using MpiJava 1.2.7, MapReduce tests were performed using Hadoop 1.0.3, Spark version was 1.6.0 and Hama version was 0.7.0. MpiJava used MPICH2 for the MPI communication internally.

The modeling parameter configuration values for each of these experiments were the same as defined in Table 7.2 except we changed the size of the dataset.

7.3.2 Experiment results

PAM

Table 7.3 shows the benchmarking experiment results for the modeled PAM algorithm and its parallel speedup is shown on Figure 7.6. Parallel speedup is calculated in comparison to the sequential, single process Java execution run time. MPI implementation achieves the best results for all the dataset sizes. Apache Hama also performs well for the largest dataset, but is significantly slower than MPI for the two smaller datasets.

Spark is only able to achieve scale-up in comparison to the single process Java execution when using the largest dataset with 80000 objects. It

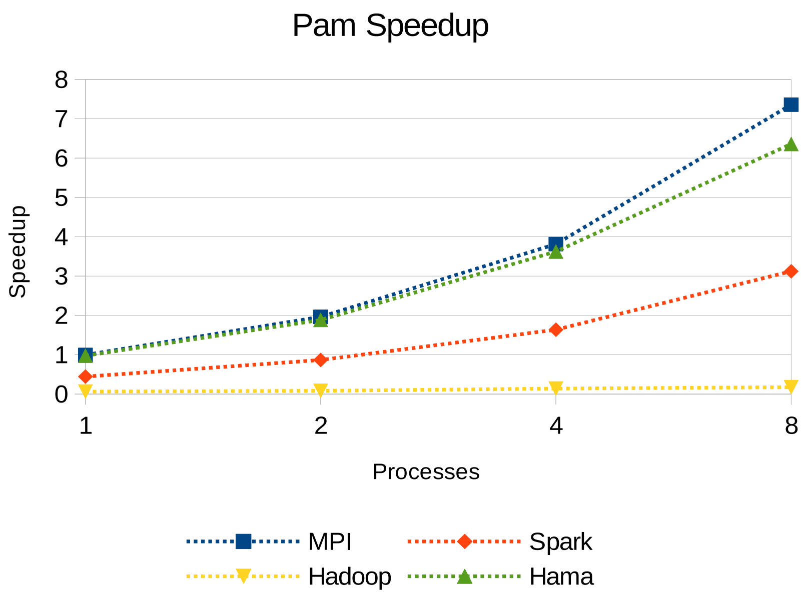

Figure 7.6: Modeled PAM speedup in comparison to Java sequential runtime

achieves no scaling for the 13333 dataset and barely achieves small amount of runtime increase for the 40000 dataset when running on 8 processes. Hadoop results are extremely slow in comparison to all other results as is expected when dealing with iterative algorithms. For the largest data set with 80000 objects Hadoop was 17.8 times slower than Spark and 41.8 times slower than MPI.

In comparison to the previous results [5] provided in Table 5.1 where different MapReduce frameworks were compared to MPI, the respective difference between MPI and Spark is smaller and Spark is able to achieve much better speedup. Modeled Hadoop is actually performing much worse than in the previous results. While the modeled and previous results do not align very well, the analysis of the modeling results would still lead to the same conclusion that MPI is significantly more suitable for PAM than Spark or Hadoop.

Table 6.1 provides the results from our previous work [1], where Apache Hama PAM implementation was compared to Hadoop and MPI. In comparison to other tables, the values in this table are not total runtime but rather the average runtime of iterations. When looking at the 8 process

	Java sequential			MPI			Spark			Hadoop			Hama		
Proc.	13333	40000	80000	13333	40000	80000	13333	40000	80000	13333	40000	80000	13333	40000	80000
1	22.23	181.7	717.2	22.29	182.39	720.42	173.16	512.27	1617.7	9224.5	9698.0	11353	29.59	192.53	735.33
2	-	-	-	14.02	95.21	365.59	109.56	279.24	828.49	7101.3	7412.6	8688.1	21.71	104.45	380.36
4	-	-	-	9.33	50.49	187.64	89.17	200.43	438.17	4248.7	4512.8	5122.7	21.46	60.26	197.96
8	-	-	-	5.53	28.36	97.49	66.72	163.0	229.65	3476.3	3630.5	4076.1	19.40	41.53	112.79

Table 7.3: Running time (s) of modeled PAM clustering with different frameworks and data set sizes

	Java sequential			MPI			Spark			Hadoop			Hama		
Proc.	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil
1	19.0	39.2	58.0	18.89	38.99	57.80	26.74	47.79	68.76	344.7	364.6	381.3	22.69	46.21	65.52
2	-	-	-	11.54	22.88	34.32	15.05	25.61	39.09	265.9	272.8	283.8	11.26	22.10	32.67
4	-	-	-	7.07	13.10	19.59	11.70	19.76	23.46	153.6	163.9	170.5	6.638	12.17	17.56
8	-	-	-	6.18	10.02	13.12	6.66	9.34	12.66	128.2	136.3	145.0	4.769	7.523	9.917

Table 7.4: Running time (s) of modeled CLARA with different frameworks and data set sizes

	Java sequential			MPI			Spark			Hadoop			Hama		
Proc.	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil
1	3.67	11.6	41.2	3.83	11.78	42.16	56.95	64.97	89.09	7578.5	9371.5	16919	8.409	19.90	58.26
2	-	-	-	3.18	6.56	29.12	35.01	40.25	53.75	5651.2	6792.6	11418	7.930	13.51	36.80
4	-	-	-	3.06	5.28	17.12	29.64	28.85	34.21	3302.6	4477.9	9398.8	9.522	12.55	24.74
8	-	-	-	3.76	4.99	9.37	21.71	24.69	28.30	2801.2	4193.2	8666.6	8.791	10.93	16.58

Table 7.5: Running time (s) of modeled Conjugate Gradient with different frameworks and data set sizes

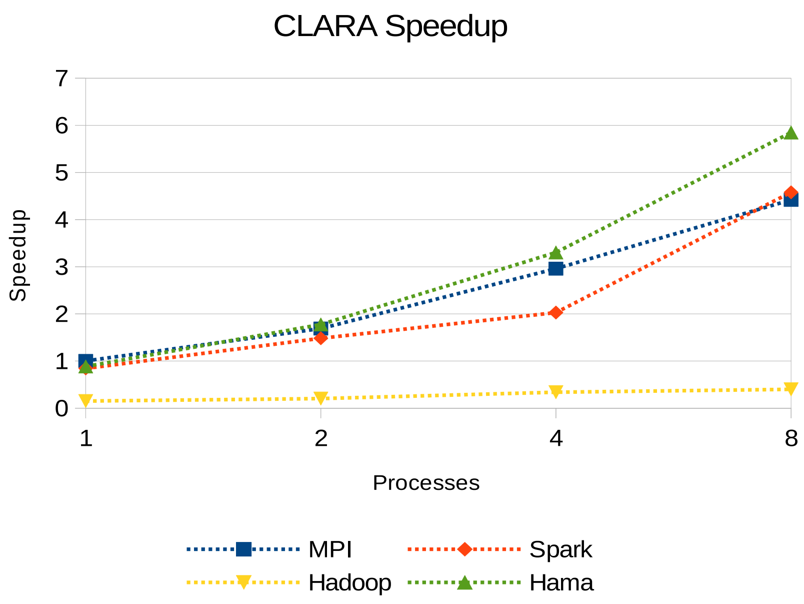

Figure 7.7: Modeled CLARA speedup in comparison to Java sequential runtime

runtime, we can see that Hama is actually performing better than MpiJava, while in the modeled results MpiJava is a little more than 10% faster. Hadoop is again significantly slower.

CLARA

Table 7.4 shows the benchmarking experiment results for the modeled CLARA algorithm and its parallel speedup is shown on Figure 7.7. These results show similar performance for both MPI and Spark. MPI is able to outperform Spark when using 1-4 processes, but Spark achieves better results when using 8 processes for data set sizes 2.7 million and 4 million. Surprisingly, Hama consistently performs better than both MPI and Spark in this case, and achieves significantly better speedup on 8 processes. Hadoop results are again the slowest, but the difference is much less than it was with the previous two algorithms. For the largest data set with 64 million objects Hadoop was 11.5 times slower than Spark and 11.1 times slower than MPI.

The modeled results are quite different in comparison to the previous results [5] provided in Table 5.2. The modeled CLARA is able to perform

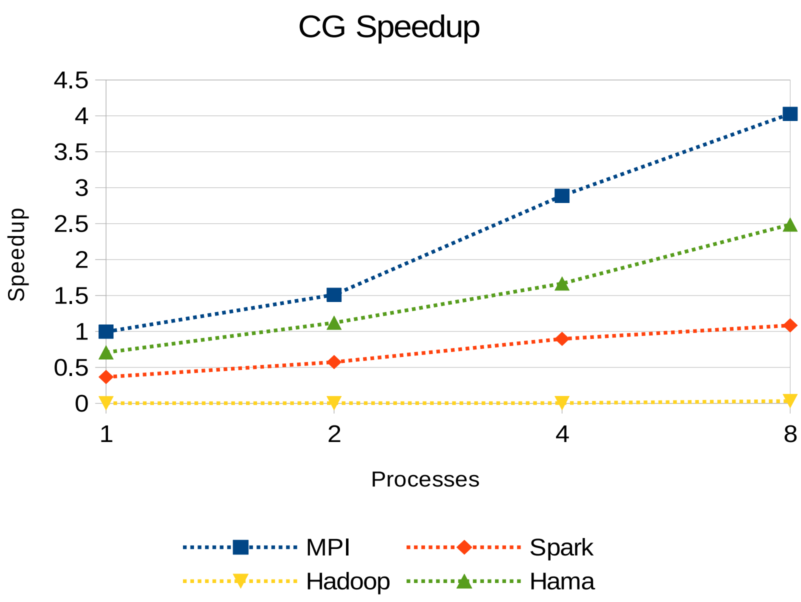

Figure 7.8: Modeled CG speedup in comparison to Java sequential runtime

much faster. Spark is able to achieve similar speedup but both MPI and Hadoop are not able to achieve the same speedup. It indicates that we have not modeled the computational complexity of CLARA well enough.

Conjugate Gradient

Table 7.5 shows the benchmarking experiment results for the modeled CG algorithm and its parallel speedup is shown on Figure 7.8. We can see that again MPI implementation achieves the best results and Hama is second. Spark is not able to achieve any speed-up in comparison to the sequential, single process Java execution even when using 8 processes instead of 1. Hadoop results are again very slow. For the largest data set with 64 million objects Hadoop was 464 times slower than Spark and 1724 times slower than MPI.

When comparing these results to our previous [5] experiments (table 5.3), Spark is again not able to achieve as good speedup and Hadoop is performing worse. In previous results it took almost no time to calculate CG results even for 64 million elements on 1 node. There are a number of causes that affects this, such as having a 100 ms sleep time in our model which counts to additional 1.2 seconds of runtime for CG and also the fact that. Another likely cause is that we simplified the model of CG by replacing matrix-vector multiplication with vector dot product operations.

It is clear that achieving the same runtime in comparison to the original algorithm is extremely difficult, if not practically impossible. However, it is not critical as long as the performance differences between the frameworks stay the same.

Table 6.2 provides the results from our previous work [1], where Apache Hama CG implementation was compared to MPI. When looking at the results of 8 processes, the difference between MpiJava and Apache Hama is very similar to the modeled results. Hama is consistently approximately twice slower than MpiJava.

7.3.3 Discussion of the results

MPI has the best performance for each of these algorithms as was our expectation. Apache Hama performance is quite comparable to MPI. As was the fact that Hadoop performance is horrible as long as there are many iterations. But even in the case of CLARA, where there are only 2 data synchronization operations, it can not compete with MPI and Spark as it still needs to store intermediate data to HDFS and Spark and MPI both perform all computations in memory. The differences between MPI and Spark for the iterative algorithms PAM and CG indicate that Spark is not as efficient as MPI when dealing with iterative computations, but the CLARA results show that it can achieve a comparable performance and speedup otherwise.

The PAM and CG performance comparison results show that the modeled results behave reasonably close to the previous results. In the case of CLARA, the results are quite different, which shows that we did not model CLARA accurately enough. CLARA algorithm includes two completely different computational tasks, first task is applying PAM on randomly sampled smaller data to find s candidate cluster medoidsets and the second is applying s medoidsets on all the input data to find the medoidset that gives the best result.

The results show that we did not model the balance between these two tasks correctly. Both of these tasks perform the same basic operation, but first scales depending on how many samples are extracted and how big they are and the second scales on the input size. Modeling the computational complexity was easier for both PAM and CG as PAM only has one computationally heavy task which is recalculating cluster centers and CG is not computationally demanding algorithm in comparison to the input size, which makes modeling the data synchronization of CG more important.

Our approach worked well in the cases of PAM and CG as the differences in parallel speedup are similar when comparing the performances of the different frameworks. However, CLARA results show that it is not easy to verify that the algorithm has been modeled correctly. We are comparing a manually modeled algorithm to custom implementations of the same algorithms for different distributed computing frameworks.

In addition, the choice of CLARA parameter (such as the number and the size of CLARA samples) values has significant affect on the balance of computations versus communication effort, which is not straightforward to model in DAMA.

7.4 Summary

We proposed a methodology for simplifying the process of choosing the best suitable distributed computing framework for a given scientific computing algorithm. It involves analyzing algorithms from the parallelization viewpoint to identify characteristics that affect the efficiency of their parallel implementations. The values of these characteristics are used to model the parallel structure and execution of the algorithm which in turn is used to estimate the performance of the original algorithm on different parallel programming solutions.

We designed and implemented a Dynamic Algorithm Modeling Application (DAMA) that simulates the parallel structure and behavior of algorithms modeled using this methodology. It can be applied directly as a benchmark to compare the performance of different distributed computing frameworks and libraries. The results of the benchmarking can then be used to estimate which framework would give the best parallel performance for the given algorithm.

There are still a number of issues with this approach. To take full advantage of our methodology, users must be able to analyze the algorithms from the parallelization viewpoint. This requires at least basic knowledge of parallel computing techniques and thus it is not easily approachable for novices in parallelization. A real computing cluster is needed to perform benchmarks, where all the required distributed computing frameworks are installed.

However, it should be clear from our initial results that this approach has great potential to significantly reduce the actual amount of work that is required from the user, such as programming and debugging effort. We feel that it also enables users to consider additional available distributed computing frameworks that they otherwise might ignore. Either because of lack of knowledge about their existence, their apparent complexity or simply because they are already familiar with other frameworks and can't afford to spend time investigating all the available ones.

In addition, it is evident from our own experience that a tool like DAMA would have been extremely useful when we were comparing MapReduce and BSP implementations. In the case of BSP model we implemented 2 algorithms a total of 9 times using 5 different distributed computing frameworks and parallel programming libraries. In the case of MapReduce model, we implemented 3 algorithms using 5 different parallel programming solutions. While there were overlaps, as the algorithms implementations were mostly same for MPI library and Hadoop MapReduce framework, it still required a lot of programming and debugging effort.

To redo the same work in DAMA, instead of implementing one algorithm in a number of different frameworks, we only have to model it once to create a configuration for DAMA. If DAMA does not support the distributed computing framework we are interested in, we only have to implement DAMA itself once for that framework. Furthermore, the framework specific implementation of DAMA would be relatively straightforward, consisting mostly of data synchronization methods, how the kernel is applied to the data slices in parallel and possibly having to implement custom data structures specific to the used framework. The main structure of DAMA would not have to be modified as long as long as it supports Java programming language.

This approach can also be used to estimate whether to move to an alternative distributed computing framework for already implemented algorithms and whether it would give a significant performance gain. It would require additional investment of time and effort, but knowing what is the approximate performance gain may provide strong enough incentive.

CHAPTER 8

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The hypothesis of this work was that *distributed computing frameworks can simplify creating parallel scientific computing applications without significantly losing the efficiency and scalability in comparison to using Message Passing Interface (MPI) libraries.*

It is well known that while distributed computing frameworks based on the MapReduce model (such as Hadoop MapReduce) provide definite advantages for certain types of algorithms, they are not designed to support more complex algorithms. They are mainly designed for processing huge amounts of textual data using embarrassingly parallel algorithms and they store all intermediate data on disk instead of memory.

There are a number of alternative distributed computing frameworks that are designed for specific types or algorithms, such as MapReduce based frameworks for iterative algorithms like Twister and HaLoop, or BSP based graph processing frameworks like Pregel, GPS and Apache Giraph. There are also general purpose frameworks based on these models like Apache Spark and Apache Hama which try to support all types of algorithms, but our experiments have shown that in comparison to MPI they have difficulties with certain types of algorithms.

With the emergence of distributed computing platforms such as Aneka and Hadoop Yarn which allow users to switch between different distributed computing frameworks on-demand, users are able to choose a distributed computing frameworks that best matches their domain specific algorithms without having to switch between different computing clusters.

However, choosing which specific distributed computing framework to use is not an easy task unless the user has extensive knowledge in using each of these frameworks and in creating parallel computing applications in general.

First contribution of this thesis was a classification for scientific computing algorithms for estimating whether Hadoop MapReduce is a suitable framework for parallelizing them. It divides algorithms into different classes based on the their parallel structure when adapted to the MapReduce model. This classification allowed us to partially answer the hypothesis of this work as we could identify what types of scientific computing algorithms could exploit the advantages of Hadoop MapReduce (such as automatic parallelization and fault tolerance) to significantly simplify their parallelization.

Second contribution consisted of three different alternative approaches for scientific computing algorithms for which Hadoop MapReduce is not suitable (based on the classification in the first contribution). These approaches were (i) reducing the number of iterations in algorithms to make them more adaptable for MapReduce; (ii) using alternative MapReduce frameworks that are specifically designed for iterative algorithms; (iii) using the Bulk Synchronous Parallel distributed computing model as alternative to MapReduce.

After evaluating the first approach of this contribution, it was clear that it requires significant expert knowledge of the involved algorithm and MapReduce and could not be successfully applied in every case.

Second and third approaches provided significant advantages for scientific computing algorithms which belonged to classes Hadoop MapReduce has difficulties with. They showed that it is possible to use alternative MapReduce or BSP frameworks for such algorithms to achieve a comparable performance to MPI. However, both of these approaches involve investigating a number of different distributed computing frameworks as potential candidates and require to spend large amount of time and effort for each new algorithm.

The first two contributions confirmed our hypothesis that *distributed computing frameworks can simplify creating parallel scientific computing applications without significantly losing the efficiency and scalability in comparison to using Message Passing Interface (MPI) libraries.* However, choosing which approach to apply for a given algorithm is not a simple task and requires expert knowledge in parallel programming and in the involved distributed computing frameworks. This led us to an additional research question: *How to choose which available distributed computing framework is the most suitable for a given scientific computing algorithm?* Which in turn led us to the final contribution of this thesis.

The third contribution was a Dynamic Algorithm Modeling Application (DAMA) and a methodology which applies DAMA for choosing which distributed computing implementation is more suitable for a given scientific computing algorithm. DAMA allows to estimate the parallel efficiency of algorithms on MPI, MapReduce, Hama and Spark without having to first adapt the algorithm to any of these distributed computing solutions. The proposed methodology was applied on a number of example algorithms to validate it.

These contributions are important for computer scientists who need to design complex distributed scientific computing applications and who would like to know whether their algorithms could take advantage of the latest distributed computing frameworks to simplify their creation, debugging and scaling. It also allows them to estimate how efficient the results would be in comparison to parallelizing the same algorithms using MPI.

8.1 Future Work

There are a number of improvements that we plan to introduce to the Dynamic Algorithm Modeling Application, such as adding support for additional distributed computing frameworks (NEWT, Tez, Twister, etc.). Supporting the execution of multiple different kernels at each iteration would also be important, but it's already possible to create kernels that apply a number of different kernels internally.

We also need to implement additional kernels which are typically used in scientific computing applications. While the process of implementing and adding new kernels to DAMA is not difficult, extending DAMA should not be part of the normal use case.

Another potential future work is to simplify the process of identifying algorithm parallelization characteristics by creating a repository consisting of best practices and guided examples. It is also worth noting that while we concentrate on scientific computing algorithms in this thesis, there is no reason why our approach can not be used for other types of algorithms, such as ones used in graph computing or data intensive applications.

Apart from extending DAMA, we are also interested in collecting data from the benchmarking experiments that users perform. Analyzing data consisting of DAMA modeling configurations, benchmarking results, software versions and cluster configurations could provide a lot of insight. It might be possible to estimate what kind of performance can be expected without running any actual benchmarking experiments, if enough data is collected from users who have already ran similar experiments. We can also analyze such data to find additional patterns between specific algorithm parallelization characteristic values and the performance of different distributed computing frameworks.

DAMA can also be used to analyze the performance of specific distributed computing frameworks more precisely, as it is possible to change all the parallelization characteristics on-demand to execute parameterized tests. Framework creators can use this approach to optimize their framework in comparison to existing competitors.

LIST OF ORIGINAL PUBLICATIONS

1. P. Jakovits, I. Kromonov, and S. Srirama, “Monte Carlo Linear System Solver using Map-Reduce,” in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pp. 293 – 299, IEEE, 2011

2. S. N. Srirama, P. Jakovits, and E. Vainikko, “Adapting scientific computing problems to Clouds using MapReduce,” *Future Generation Computer Systems*, vol. 28, no. 1, pp. 184–192, 2012

3. P. Jakovits, S. N. Srirama, and I. Kromonov, “Viability of the Bulk Synchronous Parallel model for science on Cloud,” in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pp. 41–48, IEEE, 2013

4. P. Jakovits and S. N. Srirama, “Clustering on the Cloud: Reducing CLARA to MapReduce,” in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, NordiCloud ’13, (New York, NY, USA), pp. 64–71, ACM, 2013

5. I. Kromonov, P. Jakovits, and S. Srirama, “NEWT - A resilient BSP framework for iterative algorithms on Hadoop YARN,” in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 251–259, IEEE, 2014

6. P. Jakovits and S. Srirama, “Evaluating Map-Reduce frameworks for iterative scientific computing applications,” in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 226–233, IEEE, 2014

BIBLIOGRAPHY

[1] P. Jakovits, S. N. Srirama, and I. Kromonov, "Viability of the Bulk Synchronous Parallel model for science on Cloud," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pp. 41–48, IEEE, 2013.

[2] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to Clouds using MapReduce," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 184–192, 2012.

[3] P. Jakovits and S. N. Srirama, "Clustering on the Cloud: Reducing CLARA to MapReduce," in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, NordiCloud '13, (New York, NY, USA), pp. 64–71, ACM, 2013.

[4] P. Jakovits, I. Kromonov, and S. Srirama, "Monte Carlo Linear System Solver using Map-Reduce," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pp. 293 – 299, IEEE, 2011.

[5] P. Jakovits and S. Srirama, "Evaluating Map-Reduce frameworks for iterative scientific computing applications," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 226–233, IEEE, 2014.

[6] I. Kromonov, P. Jakovits, and S. Srirama, "NEWT - A resilient BSP framework for iterative algorithms on Hadoop YARN," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 251–259, IEEE, 2014.

[7] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for

delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[8] S. Srirama, O. Batrashev, P. Jakovits, and E. Vainikko, "Scalability of parallel scientific applications on the cloud," *Scientific Programming*, vol. 19, no. 2, pp. 91–105, 2011.

[9] S. Saini, S. Heistand, H. Jin, J. Chang, R. Hood, P. Mehrotra, and R. Biswas, "An Application-based Performance Evaluation of NASA's Nebula Cloud Computing Platform," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 336 –343, 2012.

[10] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931 – 945, 2011.

[11] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.

[12] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[13] J. Hursey and A. Lumsdaine, "A Composable Runtime Recovery Policy Framework Supporting Resilient HPC Applications," technical report, School of Informatics and Computing, 2010.

[14] G. E. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (London, UK, UK), pp. 346–353, Springer-Verlag, 2000.

[15] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *Dependable Systems and*

Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on, pp. 610–621, 2014.

[16] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, 2008.

[17] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[18] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

[19] T. Dalman, T. Dörnemann, E. Juhnke, M. Weitzel, W. Wiechert, K. Nöh, and B. Freisleben, "Cloud MapReduce for Monte Carlo bootstrap applied to Metabolic Flux Analysis," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 582 – 590, 2013.

[20] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pp. 277 – 284, 2008.

[21] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," in *36th International Conference on Very Large Data Bases*, (Singapore), 2010.

[22] Apache Software Foundation, "Apache Hadoop YARN." `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`. Last Visited on August 2016.

[23] X. Chu, K. Nadiminti, C. Jin, S. Venugopal, and R. Buyya, "Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business," in *Applications, Proceedings of the 3 rd IEEE International Conference and Grid Computing*, pp. 10–13, IEEE CS Press, 2007.

[24] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant, "Bulk Synchronous Parallel Computing – A Paradigm for Transportable Software," in *In Proc. IEEE 28th Hawaii Int. Conf. on System Science*, pp. 268–275, Society Press, 1995.

[25] S. Ghemawat and Jeffrey Dean, "MapReduce: Simplified data processing on large clusters," in *Proc. of the 6th OSDI*, 2004.

[26] C. Poulain, "Microsoft Dryad." `http://research.microsoft.com/en-us/projects/dryad/`. Last visited on August 2016.

[27] Google Cloud Platform, "Spark and Hadoop on Google Cloud Platform Documentation." `https://cloud.google.com/hadoop/`. Last visited on August 2016.

[28] Apache Software Foundation, "Hadoop Distributed File System." `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Last Visited on August 2016.

[29] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[30] C. Bunch, B. Drawert, and M. Norman, "MapScale: A Cloud Environment for Scientific Computing," tech. rep., University of California, Computer Science Department, 2009.

[31] Apache Software Foundation, "Apache Tez Introduction." `https://tez.apache.org/`. Last visited on August 2016.

[32] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1357–1369, ACM, 2015.

[33] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *2nd USENIX conf. on Hot topics in cloud computing*, HotCloud'10, p. 10, 2010.

[34] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pp. 810–818, ACM, 2010.

[35] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, p. 8, ACM, 2009.

[36] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, *et al.*, "BSPlib: The BSP programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947 – 1980, 1998.

[37] W. J. Suijlen, "BSPonMPI." `http://bsponmpi.sourceforge.net/`. Last visited on August 2016.

[38] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, "The Paderborn University BSP (PUB) library," *Parallel Comput.*, vol. 29, no. 2, pp. 187–207, 2003.

[39] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, pp. 135–146, ACM, 2010.

[40] M. Han and K. Daudjee, "Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 950–961, 2015.

[41] S. Salihoglu and J. Widom, "GPS: A Graph Processing System." `http://infolab.stanford.edu/gps/gps_tr.pdf`, 2013. Technical report.

[42] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An Efficient Matrix Computation with the MapReduce Framework," *Cloud Computing Technology and Science, IEEE International Conference on*, pp. 721–726, 2010.

[43] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[44] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *J. Parallel Distrib. Comput.*, vol. 69, no. 6, pp. 532–545, 2009.

[45] B. Carpenter, “mpiJava: A Java Interface to MPI,” in *First UK Workshop on Java for High Performance Network Computing, Europar 98*, 1998.

[46] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, *et al.*, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 353–377, 2004.

[47] Argonne National Laboratory, “MPICH2 - a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard.” `http://www.mcs.anl.gov/research/projects/mpich2/`. Last Visited on August 2016.

[48] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, “The nas parallel benchmarks,” *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[49] Amazon Inc., “Amazon Elastic MapReduce.” `https://aws.amazon.com/emr/`. Last visited on December 2016.

[50] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman, “Upper and Lower Bounds on the Cost of a Map-Reduce Computation,” *CoRR*, vol. abs/1206.4377, 2012.

[51] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. Fox, “High Performance Parallel Computing with Cloud and Cloud Technologies,” tech. rep., Indiana University, 2009.

[52] J. Zhang, J.-S. Wong, T. Li, and Y. Pan, “A comparison of parallel large-scale knowledge acquisition using rough set theory on different mapreduce runtime systems,” *International Journal of Approximate Reasoning*, vol. 55, no. 3, pp. 896 – 907, 2014.

[53] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for Multi-core and Multiprocessor Systems,” in *13th International Symposium on High Performance Computer Architecture (HPCA 07)*, pp. 13–24, IEEE CS, 2007.

[54] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The HaLoop Approach to Large-scale Iterative Data Analysis," *The VLDB Journal*, vol. 21, no. 2, pp. 169–190, 2012.

[55] B. Elser and A. Montresor, "An evaluation study of BigData frameworks for graph processing," in *Big Data, 2013 IEEE International Conference on*, pp. 60–67, 2013.

[56] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[57] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl, "Iterative parallel data processing with stratosphere: an inside look," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1053–1056, ACM, 2013.

[58] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, *The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis*, pp. 209–228. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[59] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: Towards an industry standard benchmark for big data analytics," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 1197–1208, ACM, 2013.

[60] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–499, 2014.

[61] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[62] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. Fox, "A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures," in *Big Data (BigData Congress), 2014 IEEE International Congress on*, pp. 645–652, 2014.

[63] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain." `https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`, 1994. Technical report.

[64] L. Kaufman and P. Rousseeuw, *Finding Groups in Data An Introduction to Cluster Analysis*. New York: Wiley Interscience, 1990.

[65] C. Pomerance, "A tale of two sieves," *Notices Amer. Math. Soc*, vol. 43, pp. 1473–1485, 1996.

[66] P. Jakovits, S. N. Srirama, and E. Vainikko, "MapReduce for Scientific Computing - Viability for non-embarrassingly parallel algorithms," in *Applications, Tools and Techniques on the Road to Exascale Computing*, vol. 22, pp. 117–124, IOS Press, 2012.

[67] S. Branford, C. Sahin, A. Thandavan, C. Weihrauch, V. Alexandrov, and I. Dimov, "Monte Carlo methods for matrix computations on the grid," *Future Generation Computer Systems*, vol. 24, no. 6, pp. 605–612, 2008.

[68] C. Weihrauch, "Data Splitting for Parallel Linear Algebra Monte Carlo Algorithms." `http://citeseerx.ist.psu.edu/viewdoc/citations?doi=10.1.1.88.1418`, 2006. Last Visited on August 2016.

[69] Amazon Inc., "Amazon Elastic Compute Cloud." `https://aws.amazon.com/ec2/`. Last visited on December 2016.

[70] Y. Gu, B.-S. Lee, and W. Cai, "JBSP: A BSP Programming Library in Java," *Journal of Parallel and Distributed Computing*, vol. 61, no. 8, pp. 1126 – 1142, 2001.

[71] E. J. Yoon, "Hama Benchmarks." `http://wiki.apache.org/hama/Benchmarks`. Last Visited on August 2016.

[72] Apache Software Foundation, “Constrained environments for launching user’s applications.” https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/SecureContainer.html. Last Visited on August 2016.

[73] Apache Software Foundation, “Apache MINA.” http://mina.apache.org/. Last Visited on August 2016.

[74] Pivotal Software, “Hamster.” http://pivotalhd-210.docs.pivotal.io/doc/2100/webhelp/topics/Hamster.html. Last Visited on August 2016.

[75] P. H. Hargrove and J. C. Duell, “Berkeley lab checkpoint/restart (BLCR) for Linux clusters,” *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006.

KOKKUVÕTE (SUMMARY IN ESTONIAN)

TEADUSARVUTUSE ALGORITMIDE TAANDAMINE HAJUSARVUTUSE RAAMISTIKELE

Teadusarvutus rakendab arvutuslikke meetodeid, et lahendada probleeme geneetikas, bioloogias, materjaliteaduses, keemias jne, kus on vaja modelleerida ja simuleerida keerulisi reaalelu protsesse või analüüsida suurt hulka andmeid. See on tugevalt seotud paralleelse programmeerimise ja suuremahuliste arvutustega, sest see tavaliselt nõuab suure hulga arvutiressursside kasutamist kohalikes klastrites, arvutusvõrkudes või superarvutites.

Avalikud pilved pakuvad neid ressursse nõudmisel ja reaalajas, kuid need on sageli üles ehitatud tarberiistvaral ja ei ole lihtne luua rakendusi, mis saavad hakkama nende ressursside massilise kasutamisega tõrketaluval viisil. Hajusarvutusmudelitel baseeruvad raamistikud, nagu Hadoop MapReduce võivad oluliselt lihtsustada seda tööd pakkudes pea automaatset paralleliseerimist ja rikete kõrvaldamist. Meie esimeseks teaduslikuks ülesandeks oli uurida Hadoop MapReduce sobivust keerulisemate teadusarvutuse algoritmide jaoks ning teha kindlaks millised algoritmi omadused mõjutavad tulemuse paralleelselt efektiivsust.

MapReduce suutis hakkama saada lihtsamate, piinlikult paralleelsete algoritmidega, nagu näiteks katsemeetodil jagamine või Monte Carlo meetodid. Samaaegselt on sellel aga tõsiseid probleeme keerulisemate ja eriti just iteratiivsete algoritmidega, nagu näiteks kaasgradientide meetodiga. Et MapReduce eeliseid (nagu peaaegu automaatset paralleliseerimist ja rikete kõrvaldamist) oleks võimalik utiliseerida ka keerulisemate algoritmide jaoks, otsustasime välja pakkuda ja uurida kolme erinevat lähenemist.

Esimene lähenemine oli iteratsioonide vähendamine algoritmides, kas neid ümber struktureerides või kasutades alternatiivseid meetodeid, mis on vähem tõhusad, kuid sobiksid paremini MapReduce mudelile. Teine lähenemine oli alternatiivsete MapReduce raamistikke (nagu näiteks Twister, HaLoop ja Spark) võrdlus, mis on loodud spetsiaalselt iteratiivsete algoritmide jaoks, ja uurimine, kas nad pakuvad samu, varasemalt kirjeldatud eeliseid nagu Hadoop. Kolmas lähenemisviis uuris alternatiivseid Bulk Synchronous Parallel hajusarvutuse mudelil baseeruvaid raamistikke.

Meie järeldusteks oli see, et esimene lähenemine nõuaks liiga erialaspetsiifilisi teadmisi seotud meetodite ja Hadoop MapReduce raamistiku kohta. Teises ja kolmandas lähenemises uuritud alternatiivsed hajusarvutusraamistikud andsid sageli paremaid tulemusi kui Hadoop MapReduce, kuid ei leidu ühte kindlat lahendust, mis sobiks kõigile erinevat liiki teadusarvutuse algoritmide jaoks.

Tulemuse efektiivsus võib sõltuda algoritmi omadustest, nagu mälus hoitavate andmete suurus või vajalikud kommunikatsiooni mustrid ning kõige sobivama hajusarvutuse mudeli ja raamistiku valimine võib olla väga keeruline ülesanne. Üks lahendus oleks valida kõige tõenäolisemad kandidaadid, implementeerida algoritm kõigile nendest ja teha võrdlusuuringud. Aga see eeldaks suhteliselt suurtes kogustes programmeerimist ja koodi silumist, ning võib nõuda ka iga valitud hajusarvutusraamistiku kasutamise selgeks õppimist.

Selle protsessi võib keerulisemaks teha võimalus, et erinevad hajusarvutuse raamistikud on sobivad erinevatele algoritmidele, mis on osa ühest ja samast teadusarvutuse rakendusest. Arenevad tehnoloogiad (nagu näiteks Hadoop YARN või Aneka) võivad leevendada seda probleemi sellega, et võimaldavad nõudlusel ja reaalaajas sujuvalt üle minna ühelt hajusarvutuse raamistikult teisele. Kuid tegelik valik erinevate hajusarvutusmudelite ja nende implementatsioonide vahel jäetakse siiski kasutaja otsustada, mis võib olla väga keeruline ülesanne, kui olemas olevate raamistike arv on suur.

Me lõime Dünaamilise Algoritmide Modelleerimise Rakenduse (DAMR) teaduslike algoritmide paralleelse struktuuri modelleerimiseks ja defineerisime metoodika mis kasutab DAMRi selleks, et identifitseerida kõige sobivam hajusarvutuse raamistik ette antud teadusarvutuse algoritmi jaoks. DAMR on implementeeritud mitmetel hajusarvutuse raamistikel ning seda on võimalik kasutada modelleeritud algoritmide jõudluse hindamiseks, kasutades seda võrdlusrakendusena reaalsetes hajusarvutuse klastrites. Hajusarvutuse raamistike ja paralleelprogrameerimise teekide sobivust saab hinnata ilma, et peaks algoritmi implementeerima nende kõigi peal. Ning kõik programmeerimise ning silumise ülesanded saab edasi lükata kuni lõplik paralleelprogrameerimise lahendus on väljavalitud.

CURRICULUM VITAE

Personal data

Name	Pelle Jakovits
Birth	January 5th, 1983, Tartu, Estonia
Citizenship	Estonian
Languages	Estonian, English
Email	jakovits@ut.ee

Education

2010–	University of Tartu, Ph.D. candidate in Computer Science
2008–2010	University of Tartu, M.Sc. in Computer Science
2005–2008	University of Tartu, B.Sc. in Information Technology
1989–2001	Tartu Tamme Gymnasium, primary and secondary education

Employment

2013–2016	University of Tartu, Institute of Computer Science, programmer
2012–	University of Tartu, Institute of Computer Science, Assistant in Informatics
2010–2011	University of Tartu, Institute of Computer Science, programmer

ELULOOKIRJELDUS

Isikuandmed

Nimi	Pelle Jakovits
Sünniaeg ja -koht	5. Jaanuar 1983, Tartu, Eesti
Kodakondsus	eestlane
Keelteoskus	eesti, inglise
E-Post	jakovits@ut.ee

Haridustee

2010–	Tartu Ülikool, informaatika doktorant
2008–2010	Tartu Ülikool, MSc. Infotehnoloogias
2005–2008	Tartu Ülikool, BSc. Informaatikas
1989–2001	Tartu Tamme Gümnaasium, põhiharidus ja keskharidus

Teenistuskäik

2013–2016	Tartu Ülikool, Arvutiteaduse instituut, programmeerija
2012–	Tartu Ülikool, Arvutiteaduse instituut, informaatika assistent
2010–2011	Tartu Ülikool, Arvutiteaduse instituut, programmeerija

20. **Jüri Lember.** Consistency of empirical k-centres. Tartu, 1999, 148 p.
21. **Ella Puman.** Optimization of plastic conical shells. Tartu, 2000, 102 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
25. **Maria Zeltser.** Investigation of double sequence spaces by soft and hard analitical methods. Tartu, 2001, 154 p.
26. **Ernst Tungel.** Optimization of plastic spherical shells. Tartu, 2001, 90 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 p.
28. **Rainis Haller.** *M(r,s)*-inequalities. Tartu, 2002, 78 p.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
30. Töö kaitsmata.
31. **Mart Abel.** Structure of Gelfand-Mazur algebras. Tartu, 2003. 94 p.
32. **Vladimir Kuchmei.** Affine completeness of some ockham algebras. Tartu, 2003. 100 p.
33. **Olga Dunajeva.** Asymptotic matrix methods in statistical inference problems. Tartu 2003. 78 p.
34. **Mare Tarang.** Stability of the spline collocation method for volterra integro-differential equations. Tartu 2004. 90 p.
35. **Tatjana Nahtman.** Permutation invariance and reparameterizations in linear models. Tartu 2004. 91 p.
36. **Märt Möls.** Linear mixed models with equivalent predictors. Tartu 2004. 70 p.
37. **Kristiina Hakk.** Approximation methods for weakly singular integral equations with discontinuous coefficients. Tartu 2004, 137 p.
38. **Meelis Käärik.** Fitting sets to probability distributions. Tartu 2005, 90 p.
39. **Inga Parts.** Piecewise polynomial collocation methods for solving weakly singular integro-differential equations. Tartu 2005, 140 p.
40. **Natalia Saealle.** Convergence and summability with speed of functional series. Tartu 2005, 91 p.
41. **Tanel Kaart.** The reliability of linear mixed models in genetic studies. Tartu 2006, 124 p.
42. **Kadre Torn.** Shear and bending response of inelastic structures to dynamic load. Tartu 2006, 142 p.
43. **Kristel Mikkor.** Uniform factorisation for compact subsets of Banach spaces of operators. Tartu 2006, 72 p.

44. **Darja Saveljeva.** Quadratic and cubic spline collocation for Volterra integral equations. Tartu 2006, 117 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
46. **Annely Mürk.** Optimization of inelastic plates with cracks. Tartu 2006. 137 p.
47. **Annemai Raidjõe.** Sequence spaces defined by modulus functions and superposition operators. Tartu 2006, 97 p.
48. **Olga Panova.** Real Gelfand-Mazur algebras. Tartu 2006, 82 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
50. **Margus Pihlak.** Approximation of multivariate distribution functions. Tartu 2007, 82 p.
51. **Ene Käärik.** Handling dropouts in repeated measurements using copulas. Tartu 2007, 99 p.
52. **Artur Sepp.** Affine models in mathematical finance: an analytical approach. Tartu 2007, 147 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
54. **Kaja Sõstra.** Restriction estimator for domains. Tartu 2007, 104 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
57. **Evely Leetma.** Solution of smoothing problems with obstacles. Tartu 2009, 81 p.
58. **Ants Kaasik.** Estimating ruin probabilities in the Cramér-Lundberg model with heavy-tailed claims. Tartu 2009, 139 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
60. **Indrek Zolk.** The commuting bounded approximation property of Banach spaces. Tartu 2010, 107 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
63. **Marek Kolk.** Piecewise Polynomial Collocation for Volterra Integral Equations with Singularities. Tartu 2010, 134 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
65. **Larissa Roots.** Free vibrations of stepped cylindrical shells containing cracks. Tartu 2010, 94 p.