TARTU UNIVERSITY
Faculty of Mathematics and Computer Science
Institute of Computer Science
Computer Science

Ilja Kromonov

Fault Tolerant Distributed Computing Framework for Scientific Algorithms

Master's Thesis (30 ECTS)

Supervisors: Pelle Jakovits
Satish Narayana Srirama, PhD

Fault Tolerant Distributed Computing Framework for Scientific Algorithms

Abstract:

The physical limitations of computing hardware have put a stop on the increase of a single processor core's computing power. However, Moore's law is still maintained through the ever increasing parallelism of the computing architectures. At the same time the demand for computational power has been unrelentingly growing, forcing people to adapt the algorithms they use to these parallel architectures. One of the many downsides to parallel architectures is that with the rise in the number of components, the chance of failure of one of these components increases. When it comes to embarrassingly parallel data-intensive algorithms, Map-Reduce has gone a long way in ensuring users can easily utilize large amounts of distributed computing resources without the fear of losing work. However, this does not apply to iterative communication-intensive algorithms common in the scientific computing domain. In this work a new BSP-inspired (Bulk Synchronous Parallel) programming model is proposed, which adopts an approach similar to continuation passing for implementing parallel algorithms and facilitates fault-tolerance inherent in the BSP program structure. The distributed computing framework NEWT, which is based on the proposed model, is described and used to validate the approach. The framework retains most of the advantages that Map-Reduce provides, yet efficiently supports a larger assortment of algorithms, such as the aforementioned iterative ones.

Keywords:

BSP, fault tolerance, cloud computing, iterative algorithms, Hadoop YARN

Tõrketaluv Hajusarvutuste Raamistik Teadusarvutuse Algoritmidele

Lühikokkuvõte:

Arvuti riistvara füüsilised piirangud on lõpetanud protsessorite tuumade arvutusvõimsuse suurenemist, kuid arvutiarhitektuuride suurenev parallelsus säilitab Moore'i seaduse kehtivust. Samal ajal tõuseb arvutusvõimsuse nõudlus pidevalt, sundides inimesi kohandada algoritme paralleelsete arhitektuuride kasutamiseks. Üks paljudest paralleelsete arhitektuuride probleemidest on tõrkete tekkimise tõenäosuse suurenemine parallelsete komponentide arvu suurenemisega. Piinlikult paralleelsete ja andmemahukate algoritmidega seoses on MapReduce läbinud pika tee, et tagada kasutajatele suure hulga hajutatud arvutiressursside lihtsustatud kasutamine ilma töö kaotamise hirmuta. Sama ei sa öelda kommunikatsiooni intensiivsete algoritmide jaoks mis on levinud teadusarvutuse domeenis. Selles töös on pakutud uus BSP (*Bulk Synchronous Parallel*) inspireeritud parallelprogrammeerimise mudel, mille lähenemisviis on sarnane *continuation passing* programmeerimis stiiliga ja mis võimaldab rakendada BSP struktuuril baseeruvat loomulikku tõrkekindlust. Töös on kirjeldatud loodud hajusarvutuste raamistik NEWT, mis põhineb pakutud mudelil ja on kasutatud selle lähenemisviisi valideerimiseks. Raamistik säilitab enamik MapReduce eelisi ning efektiivsemalt toetab suuremat algoritmide hulka, nagu näiteks eelmainitud iteratiivsed algoritmid.

Võtmesõnad:

BSP, tõrketaluvus, pilvearvutused, iteratiivsed algoritmid, Hadoop YARN

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years cloud-based platforms have emerged as alternatives to supercomputers and grids for high performance computing needs. With the illusion of infinite resources, cloud computing allows one to loan computation time on demand with a flexible pay-as-you-use billing model. However, applications are placed in an environment associated with a high risk of hardware failure. This is further amplified in private cloud setups where use of commodity equipment lessens the infrastructure cost.

For these reasons, Hadoop MapReduce [Apa14c] framework has found widespread use in the cloud-based distributed computing field. It provides fault tolerance by replicating both data and computation in an attempt to guarantee that the started applications produce a result. Originally introduced by Google in 2004 [DG08], MapReduce excels at solving data-heavy embarrassingly parallel problems, however, it has trouble with more sophisticated algorithms [SJV12], as the model was simply not designed to support them. Furthermore, even MapReduce implementations that are aimed at iterative computation, such as Twister MapReduce [ELZ+10a], have trouble with most scientific computing problems, with one of the main reasons being that, by design, MapReduce processes are stateless. The stateless nature of a process implies that no state information is associated with the given process at any time, ensuring that any part of input data is eligible for any of the available processes without affecting the outcome. This concept ensures that failure of one of the nodes does not affect the sequential consistency of the program and is at the core of the MapReduce fault tolerance mechanism.

When MapReduce does not suffice, a common alternative is to use Message Passing Interface (MPI) - an established standard, which throughout the years has become the *de facto* way of writing parallel programs. While allowing for a large degree of flexibility in implementing synchronization between processes, MPI code tends to be error prone and difficult to debug and maintain. The ability to introduce various low-level optimizations comes with the danger of encountering

notorious deadlocks and race conditions. As of MPI version 3, fault tolerance is still not part of the MPI standard, so any developments in this direction are left to specific implementations. Most of the time these highly specialized solutions make the programmer do extra work to ensure his application is fault tolerant, or require additional cluster infrastructure, such as dedicated checkpoint servers. In practice, the effort to maintain these implementations is lacking, and they usually end up falling behind the most recent MPI standard. For example, OpenMPI has had a transparent checkpoint/restart system as part of its implementation of MPI for some time, however, since version 1.7, it's no longer maintained and thus is not part of the most current OpenMPI packages [Ind14, Ope14b].

This leads us to believe that the Bulk Synchronous Parallel (BSP) model can be an ideal basis for a parallel computing framework that would serve the needs of scientific computing in an on-demand environment, such as the cloud. It can have most of the advantages of MapReduce while providing a message passing paradigm similar to MPI without many of the issues involved. In this work a BSP-inspired programming model is proposed, enabling transparent stateful fault-tolerance for programs that follow it. The model is used to develop a distributed computing framework. A number of typical iterative scientific computing algorithms on are implemented on it and are used to validate the approach.

1.1 BSP Model

In 1990 [Val90] Valiant argued that a bridging model between software and hardware has to be introduced to properly utilize existing computing resources for parallel computation. BSP model was his proposed candidate for streamlining the move of sequential computation to the parallel infrastructure. A BSP based program consists of a series of supersteps as illustrated on figure 1.1, each divided into three stages:

- Concurrent computation (using only local data)
- Communication
- Barrier synchronization

One of its main advantages over MPI is the elimination of race conditions and deadlocks by avoiding circular data dependencies. The resulting program structure simplifies obtaining an overview of the implemented algorithm's granularity and estimating the expected parallel runtime and performance. While BSP may not have been widely adopted for its initial purpose, it has inspired new programming models and is the basis for several parallel programming libraries.

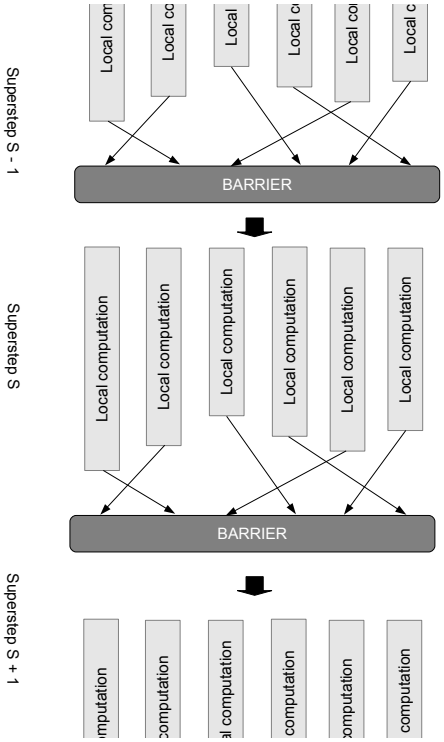

Figure 1.1: Illustration of the BSP model

1.2 Outline

Chapter 2 provides a description of current state of the art in distributed computing frameworks and outlines issues that are the motivation for this work. Chapter 3 describes the BSP-inspired programming model, that is used for the proposed distributed computing solution, as well as the implementation on top of Hadoop YARN, including the API (Application Programming Interface). Chapter 4 shows how one would adapt the *conjugate gradient method* and *partitioning around medoids* algorithms to the proposed model, followed by validating the proposal in Chapter 5 through the use of the aforementioned algorithms in a set of experiments that compare the created solution to a BSPlib implementation that uses MPI. Finally, Chapter 6 provides a summary and outlines the future work directions.

Chapter 2

State of the Art

This chapter provides the related work describing the existing distributed computing solutions in the context of fault tolerance and their suitability for long running computations. Additionally, table 2.1 provides an overview and comparison of these implementations. The table also includes the proposed solution, which will be described in more detail in the following chapters.

2.1 BSPlib

BSPlib is a specification for a programming library standard inspired by the BSP model [HMS+98]. Indeed, it was the first derivative of Valiant's efforts. Its goal was to define low-level communication and synchronization primitives that would allow the creation of parallel programs that conform to the ideas expressed in the BSP model, while keeping the API as simple and clear as possible.

Implementations which follow its specification such as PUB [BJOR99] or Oxford BSP toolset [Hil14] are not optimized for modern architectures, furthermore, Oxford BSP toolset will not even compile out-of-the-box on 64-bit systems, which are prevalent in today's high performance computing world (for example, the largest Amazon EC2 instance types are restricted to 64-bit). A more recent implementation exists in BSPonMPI [Sui14], however, as with MPI, the task of implementing fault-tolerance lies entirely on the shoulders of the user.

2.2 Pregel

Pregel [MAB+10] is a BSP-inspired parallel programming library that has since grown into a fully fledged parallel programming model. It was developed by Google engineers to address MapReduce's inability of handling iterative graph processing algorithms efficiently. In order to use Pregel for solving real-world

problems, one must first express them in terms of vertices, edges and operations to be performed on them. Google's proprietary implementation of the model is now used widely in-house to solve various problems that can be represented as a graph. Pregel has sparked tremendous interest from the parallel programming community and a number of Pregel-like solutions have been developed since its unveiling. Currently, two of the most promising open-source of projects of this kind are Apache Giraph [Apa14a] and Stanford GPS [SW13].

In Pregel, each vertex is associated with a value and a set of edges, which can change from superstep to superstep. These values are written to resilient storage at configurable intervals and can be used to restore the state of the whole computation when one or more nodes fail. Additionally, Pregel allows for messages to be transmitted between vertexes and thus all incoming messages of that superstep also need to be stored. This approach seems reasonable for many types of problems (granted it is assumed for Pregel that the state of the vector is relatively small), but the restriction of defining all supersteps of a program in a single function forces the programmer to implement algorithms as a state machine, and synchronize the current state with the master. This is a rather restrictive approach, which can get cumbersome for more complex iterative algorithms. One solution to this issue is the use of Domain Specific Languages, such as GreenMarl [HCSO12] proposed by authors of Stanford GPS. However, given the goal of Pregel, these languages are very specific to graph processing problems and thus not very suitable for general purpose programming.

2.3 Hama

HAMA [SYK+10] was originally envisioned as a framework that provides Hadoop-compatible interfaces to different computation engines, including MapReduce and Pregel. It has been one of the most actively developed (with occasional spikes of activity, as well as periods of stagnation) open-source frameworks based on the BSP model, and has gone through countless iterations, dropping support for MapReduce along the way. Currently, HAMA retains the Pregel part of the API (the 'HAMA Graph' package), but also provides a different approach to writing programs under the BSP model. This alternative interface is more akin to BSPlib than Pregel, and gives the programmer much more control over how the each superstep is defined by allowing the manual specification of synchronization points and not requiring the solved problem to be represented as a graph.

Despite being in development for years, no fault tolerance mechanisms have been implemented and any future developments in that direction are hard to assess due to a lack of available up-to-date information online.

2.4 MapReduce

In addition to the BSP implementations mentioned previously, one may argue that MapReduce also follows the Bulk Synchronous Parallel model (albeit in a very restricted manner), as it is defined by two supersteps: aptly named 'map' and 'reduce', with communication from the first superstep to the next one being followed by a global barrier. The distinguishing feature is that 'mappers' and 'reducers' are spawned separately, such that the number of active processes changes from one superstep to the other. This distinction is one of the main issues preventing MapReduce to accommodate most algorithms (most notably iterative ones). The core of the issue is that all intermediate data is written to the distributed file system, with no state being kept in memory between the two supersteps. The second issue is the stateless nature of MapReduce processes, and it surfaces when attempts are made to allow the model to accommodate iterative computations.

The fault-tolerance mechanism of MapReduce depends on rescheduling processes using the input of the failed process, since that is the only state restored during fault recovery, algorithms that hold a large amount of state information, while only small portions of it are needed to be transmitted to other processes, suffer a significant overhead from being implemented using MapReduce. Two examples of frameworks that achieve iterative fault-tolerant computation using the MapReduce model are Twister [ELZ+10b] and HaLoop [BHBE10]. The aforementioned issues apply to both of them, as they must use the underlying distributed file system to store the output of each iteration, which can be later used in the fault recovery procedure.

2.5 MPI

As of version MPI-1.3, the MPI standard does not include fault-tolerance mechanisms as part of it's specification. There exist extensions, such as FT-MPI [FD00], which propose an implementation that can recover up to $n - 1$ failed processes in a n-processes job, but still put the responsibility of recovering state on the user. Unfortunately these endeavors inevitably get abandoned when new versions of the standard get published (for instance, FT-MPI is written for MPI-1.2 and is no longer maintained). Some of the mainstream MPI implementations have had transparent fault-tolerance modules developed for them. For example, OpenMPI has had a transparent checkpoint/restart system as part of its implementation of MPI for some time, however, since version 1.7, it's no longer maintained and thus is not part of the most current OpenMPI packages [Ind14, Ope14b]. An additional issue is that typically these solutions require a distributed file system to be present on the cluster, such as IBM GPFS [IBM14] (a commercial product)

or Lustre [Ope14a], which requires additional deployment effort in an on-demand environment and not a stable cluster.

2.6 Summary

None of the mentioned frameworks provide transparent fault-tolerance and, at the same time, a convenient way to program iterative applications in addition to the ability to be easily deployed to on-demand infrastructure. As such, the proposed solution is designed with the first two goals in mind and the third is achieved by implementing it on top of Hadoop YARN, which should be satisfactory due to the relative ease of acquiring a Hadoop cluster on-demand in light of services such as Amazon Elastic MapReduce [Ama14], which now supports YARN.

	Progr. model	Applications	Fault tolerance	Data distribution	General comments
MPI	N/A	Any	Not part of MPI standard or its major implementations. Existing transparent solutions store entire task address space.	Explicit	Does not follow any specific parallel programming model, left up to user to implement.
BSPlib [HMS+98]	BSP	Iterative applications	Generally same as MPI.	Explicit	Messages are not received until global barrier has been completed. Difficult to get working on newer hardware.
Hadoop MapReduce [DG08]	MapReduce	Embarrassingly parallel, data processing	All input and output data is replicated. Restarts failed map and reduce tasks. Task state is not preserved.	Uses Hadoop Distributed File System (HDFS). Large data sets automatically split among mappers.	Programs consist of 'map' and 'reduce' functions. Speculatively executes slowest tasks on finished nodes. No real support for iterative execution. Long job configure time ($\sim$17 sec)
Twister [ELZ+10b]	MapReduce	Iterative applications	Only guarantees restoring input data that can be reloaded from the file system or static parameters inherited from the main program. Any transient information stored in Map and Reduce tasks will be lost.	No distributed file system, partitioning data is a manual process.	Supports iterative MapReduce applications. Assumes that the intermediate data produced after the map stage will fit in to the distributed memory.
Pregel [MAB+10]	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	Program consists of a single function, which is continuously executed on each vertex. Proprietary.
Apache Giraph [Apa14a]	BSP, Pregel	Graph processing	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	Program consists of a single function, which is continuously executed on each vertex. It extends Pregel by introducing master computations, shared aggregators and edge-oriented input.
Stanford GPS [SW13]	BSP, Pregel	Graph processing	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	Extends Pregel API with global computations instead of just vertex centric ones. Includes optimizations to repartitioning graphs and reducing network I/O.
Hama [SYK+10]	BSP	Any	N/A	Uses HDFS, data distribution similar to Hadoop MapReduce.	Manually placed barriers when synchronization is needed, similar to BSPlib
NEWT	BSP	Any	Checkpoints and recovers intermediary state/message queues and continues execution from the last checkpoint.	Uses HDFS. Creates a process for each input file. (Alternative approaches under consideration)	Programs consist of an arbitrary number of labeled functions with user-defined transitions

Table 2.1: Comparison between existing and the proposed approach

Chapter 3

Proposed Solution

This chapter describes the BSP-inspired programming model that will be the basis of the distributed computing framework in Section 3.1 and the implementation that uses this programming model on top of Hadoop YARN in Section 3.2.

3.1 Description

The goals of the proposed solution are the following:

- Provide automatic fault recovery.
- Retain the program state after fault recovery.
- Provide a convenient programming interface.
- Support (iterative) scientific computing applications.

The first thing to note is that in more complex iterative programs, each iteration may consist of more than one distinct BSP superstep. To accommodate the continuation of the recovered program at the correct stage of the iteration, without storing the entire address space, it makes sense to write it as a finite state machine (FSM). In the resulting FSM each such stage is equivalent to one of the states. This leads us to view programs under the BSP model as suitable for an abstract computer, consisting of:

- Memory, containing mutable state and message queues
- Mapping of labels to instructions, where each instruction corresponds to computation done at one of the supersteps
- Function pointer or state register, holding the label of the next instruction to be executed

- Communicator - allows messages to be sent and received

The given description is similar to a counter machine (apart from a communicator), and indeed we are simply applying the same principles to a higher level of abstraction. The instructions, in this case, are user-defined functions and the message queues hold incoming and outgoing messages, to adhere to principles outlined in the BSP model regarding communication and barrier synchronization. Writing a program under this model would then be similar to using continuation-passing style, known from functional programming.

Using high-level imperative programming concepts, the following pseudo code emulates the inner workings of the described abstract machine:

$state \leftarrow initialState$
$next \leftarrow initialLabel$
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 function defined by label $next$ and returns the label of the next function in the sequence. The mutation of state and sending/receiving of messages (through communicator $comm$) is achieved as a side-effect of these functions. There is a need for communication primitives that cover the semantics of sending and receiving messages. These primitives are made accessible through the communicator. The $barrier$ initiates communication and synchronizes all machines as per the BSP model.

The given generic program structure allows for the state, label of the next stage and incoming message queue to be stored into persistent storage (such as a distributed file system) between invocations of $execute$, for later recovery in case of machine failure. This recovery is then seamlessly achieved by using the stored data on the rescheduled task, instead of the initial one, and, in the same fashion, read from the checkpoint by the remaining processes, when they are notified of failure elsewhere in the network by a coordinator, which has to detect the failure state in the first place. It has to be noted, that the processes that did not fail do not need to be restarted and can complete the recovery by simply replacing the current state with an earlier one and then continue the execution.

A program under this model has to define the state and a mapping of labels to functions, which describe the program flow. The return value of each of these functions is explicitly the label of the next function to be executed. It is possible

to use this model to generalize any MapReduce program, since such programs can be represented with stages labeled 'map' and 'reduce' with a null state, sending messages at the end of 'map' to machines associated with specific keys, with these messages becoming available for processing at the 'reduce' stage. However, it is not restricted to problems that can be summarized in only two stages. For instance, one can model iterative programs with an arbitrary number of iterations, by having a function return its own label as long as more iterations need to be computed, and since the tasks remain in memory and are not rescheduled for each stage, the overhead is minimal. The frequency of checkpointing can then be configured by the user, based on the estimated running time and the stability of the underlying cluster hardware.

3.2 Implementation

Successful fault-tolerance of processes depends on a scheduling mechanism and a resilient storage environment, where checkpoints can be stored in a reliable manner and retrieved in case of machine or network failures. There are existing solutions that can be used for this purpose. To take advantage of the ongoing development of Apache Hadoop the following established and continuously supported software components were chosen:

- YARN (Yet Another Resource Negotiator) - separates the Hadoop framework components, allowing for models other than MapReduce to utilize cluster resources. [Apa14d]
- HDFS (Hadoop Distributed File System) - provides a highly fault-tolerant distributed file system. Designed to be run on commodity hardware. [Apa14b]
- Apache MINA - a network application framework which helps users develop high performance and high scalability network applications easily. [Apa14e]

The Hadoop kernel is written in Java and exposes mostly Java interfaces for developers, as such NEWT is implemented in Java.

Resource Management and YARN

Yet Another Resource Manager (YARN) or sometimes MapReduce v2 is the new generation of the Hadoop MapReduce (MR) architecture, which separates the resource management and job scheduling functionality from the JobTracker

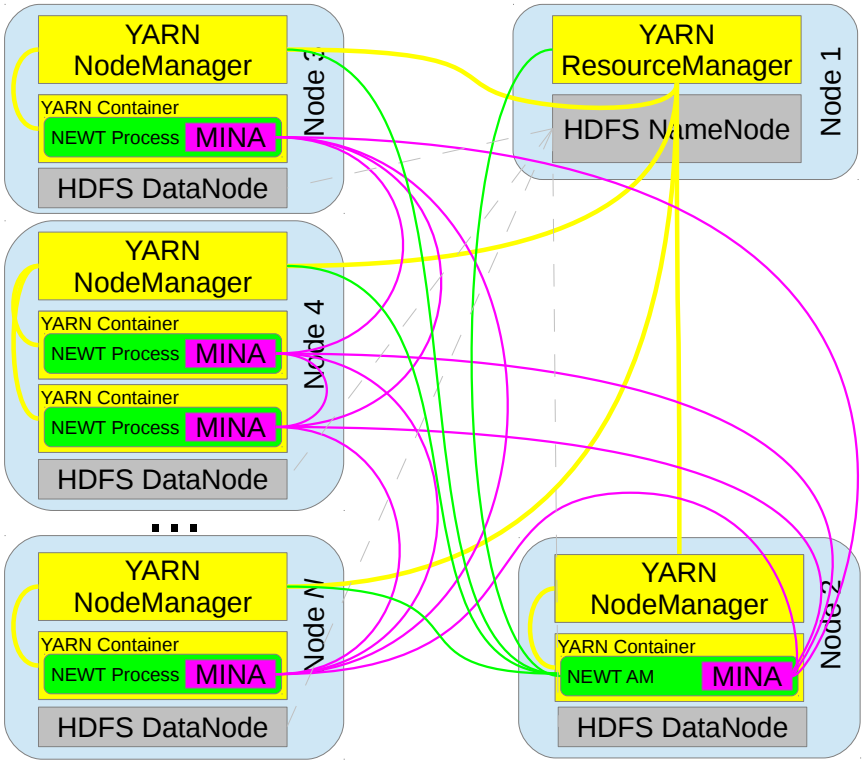

Figure 3.1: Overview of the architecture of NEWT and related technologies

used to manage MR jobs in the previous iterations of the architecture. This development is the result of the Hadoop developers recognizing the inadequacy of the MR paradigm for solving many computational problems that a Hadoop cluster resources could be utilized for. YARN opens up Hadoop clusters to alternative distributed computing and data processing frameworks.

The architecture of YARN is shown in figure 3.2. As can be seen from the figure, the two main components of a YARN user application are the ApplicationMaster (AM) and the Container. The AM is responsible for requesting resources (containers) and responding to their life-cycle events. The container has a number of computational (CPUs) and memory (RAM) resources attached to it, which are allocated by the YARN ResourceManager (RM). When the AM receives container handles from the RM, it is able to launch applications within those containers and periodically get notifications about the status of the launched processes.

When implementing NEWT on top of YARN, the NEWT AM serves as a

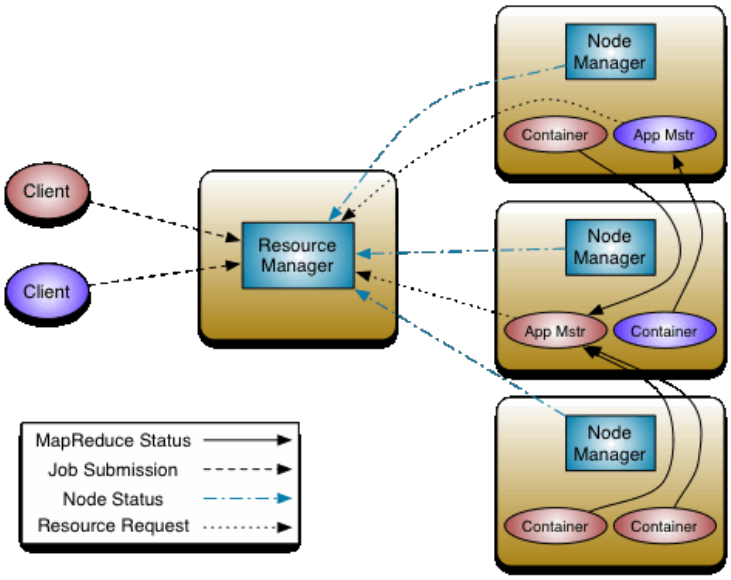

Figure 3.2: The architecture of YARN [Apa14d].

coordinator for the framework and the processes within the containers it receives from the RM run the main program loop as described in Section 3.1.

The *NEWTClient* is responsible for requesting a container from the RM for running the AM. If the RM is able to allocate the requested resources, the newly launched AM will take over the responsibility for starting processes as needed by the configuration of the task. The client may optionally monitor the status of the AM. The figure 3.1 shows the relationships between components of NEWT and related technologies. The green lines depict the connections the NEWT AM needs to establish to launch the processes: the connection to RM is used to request resources and monitor their use, and the connection with node managers is used to launch containers according to the container handles received from the RM.

Before the AM is launched, it needs to be provided with a configuration of the computational task that needs to be performed. This configuration consists of:

- input format of the task
- state object class of the task
- any number of functions that define the task (formatted as Java 1.7 closures)
- any custom message types that are used by the task

With this configuration the AM is able to set up the processes inside YARN containers to execute the required computational task.

Input Format

The framework provides the *Input* interface for defining the input format of the task. The interface defines a function *getForProcess(pid)*, which returns a filepath object that specifies the input for the given process ID, and a function *int initialize()* which returns the number of processes that need to be launched.

There are several input formats predefined by NEWT, including:

- NullInput - has a "number of processes" argument which the *initialize* function returns and *getForProcess* is *null* for all processes.
- HDFSFileInput - takes a path to a directory in HDFS as an argument and creates a process for each file in that directory, each process takes one of the files as input.
- CommandLineInput - takes the commandline arguments that are used to run the program and writes them to a file in HDFS, processes can parse the arguments using the *CommandLineInput.parseFromInputStream* method.

State Object

The state object class has to inherit from the *BSPState* abstract class, and has to define a constructor that accepts a *JobContext* instance. The *JobContext* class provides access to certain contextual information specific to the given process, such as it's process ID, the total number of processes, and the input stream for the given process. The constructor has to initialize all the state that is required for the task at hand. Additionally, *writeTo* and *readFrom* methods need to be defined, which detail which of the state should be written to/read from a checkpoint. Alternatively, the state object class may implement the Java *Serializable* interface, but this is not recommended.

Task Functions

When defining the description of the task, the user creates the AM instance and uses it's *addStage(functionLabel, functionClosure)* method to register functions that the task should run. The label of the first function added will be considered the initial function by default, it is possible to override this behavior via *setInitialStage(functionLabel)*. The current version is written using Java version 1.7 so the functions are defined by inheriting the *Stage<BSPState>* abstract class, which requires the user to implement the *execute(bspComm, state, superstep)* method. In future this functionality may be rewritten to use the much more concise Java 1.8 closure syntax.

The function accepts several parameters:

- Instance of the *BSPComm* class, which exposes communication related functionality to the function
- Instance of the subclass of *BSPState* which was declared as the generic parameter of *Stage*
- The current superstep number

Message Types

All messages sent by NEWT processes have to implement the *Writable* interface. Hadoop has a large number of writable types defined in package *org.apache.hadoop.io*, which are all usable in NEWT tasks by default. If any custom message types are to be used, they need to be registered with the AM using the *registerMessageType(messageClass)* method.

Fault Tolerance and HDFS

The Hadoop Distributed File System (HDFS) is known as the integral component of the Hadoop kernel and has been at the core of the Hadoop MapReduce framework, where it stored both input and output data, as well as intermediary key-value pairs that the mapper stage outputs. It mimics in functionality the Google File System and is designed with commodity hardware in mind. The file system stores data in blocks of configurable size and has a replication factor attribute, which defines on how many nodes each file block should be stored. The replication enables fault-tolerant storage, which in turn allows for fault-tolerant computing tasks to be executed on the Hadoop platform.

NEWT uses HDFS to store snapshots of the state of the processes running inside the containers that the RM allocates. These consist of:

- superstep number
- identifier of the next function
- state object of the executed task
- contents of message queues at the beginning of the superstep

In case of failure of any machine in the cluster, the AM is able to restart the failed process and provide it with a checkpoint ID, which will be loaded from HDFS and the containing state will be used instead of the initial one. The remaining processes will also be notified of the recovery and will replace their current state with the one loaded from HDFS.

One of the defining features of the MapReduce framework is that of data locality. It implies that the computation is moved to the data and not *vice versa*, which reduces overhead from data transfer over the network when starting tasks. Hadoop MapReduce used this property to great effect and it can also be achieved using YARN and HDFS. The HDFS API allows for the location of data to be retrieved and used in the YARN container request as the preferred target node, which allows NEWT to launch data-local processes.

Communication and MINA

The communication part of the framework is exposed through the *Communicator* interface, which defines common communication related operations performed by NEWT (Table 3.1).

The given interface defines communication operations for processes that run inside YARN containers, a similar interface also does the same for the AM.

At the time of creating NEWT, there has been ongoing work in integrating MPI with the YARN resource manager [ham14], but no publicly available stable implementation existed, as such it was decided to implement the *Communicator* using Java NIO, specifically - Apache MINA. Alternative communication modes can be added by implementing the *Communicator* using frameworks other than Apache MINA.

There are several tasks that the communication module has to perform, these include:

- initialization and establishing connections
- barrier synchronization
- message passing between processes

Initialization and Establishing Connections

As can be seen from figure 3.1, each NEWT process and the NEWT AM establish a socket connection with each other participant using the TCP/IP protocol. Initially the AM establishes connections with all processes and maintains a mapping of process IDs to addresses, which is forwarded to the newly created processes. The processes use this mapping to establish socket connections with each other. When any processes fail, the AM has to update the mapping after relaunching the failed processes and sends the updated mapping to all participants yet again, such that they can reestablish connections with both new and old members of the task.

function	description
init(context) reinit(context)	Initialize the communicator and setup the connections to other processes. The *context* argument is an instance of *JobContext* class, which provides access to certain contextual information specific to the given process.
sendBufferedMessagesAsync(context, pid) waitUntilBufferedMessagesSent(context, pid)	Sends the accumulated message buffer to the process identified by an integer *pid*. The operation is asynchronous, as such the *send* function is non-blocking and the *wait* function blocks until the last send completes.
requestFromMasterAsync(request) waitUntilAnswerFromMasterAsync()	Sends a *request* to the master, which is an enum signifying one of the requests from processes that the master answers to. The operation is asynchronous, as such the *send* function is non-blocking and the *wait* function blocks until the last send completes and then returns the response object.
requestFromMasterSync(request)	Synchronous version of the previous operation.
isConnectedToProcess(pid)	Returns true or false depending on the state of the connection with process signified by integer *pid*.
readFromProcessAsync(pid) waitUntilReadFromProcessAsync(pid)	Listens for a message from the process signified by *pid*. The operation is asynchronous, as such the *read* function is non-blocking and the *wait* function blocks until the last read completes and then returns the response object.
dispose()	Closes connections with other processes and disposes of the communicator.

Table 3.1: The *Communicator* interface

Barrier Synchronization

The barrier synchronization step serves both to ensure the guarantees of the BSP model in relation to race conditions and to coordinate fault-tolerance mechanisms.

The algorithm for performing barrier synchronization is as follows:

- On the process side:
  - sends a *BARRIER ENTERED* packet to the AM
  - does a blocking read from the AM
  - when the read completes the synchronization is done
- On the AM side:
  - listens to processes for *BARRIER ENTERED* packets until it receives one from each
  - sends a *barrier response* to each process, this can be either *BARRIER EXITED*, *CHECKPOINT NEEDED* or *RECOVERY NEEDED*, the latter of which is also followed by a checkpoint ID

Message Passing Between Processes

Message passing related primitives are exposed through the *BSPComm* class, an instance of which is passed to each execution of the functions that define the task. This allows the interaction with the user to happen independently of any used communication library. The communication methods are:

- send(msg,pid)/send(msg,pid,tag) - queues message *msg* for delivery to a process with ID *pid*, may be marked with the specified tag
- sendAll(msg)/sendAll(msg,tag) - queues message *msg* for delivery to all processes in the current task, may be marked with the specified tag
- move()/move(pid) - moves a message from the receive queue, or from the receive queue of the process with ID *pid*
- getReducedValue(tag,reduceOp) - syntactic sugar for the "reduce" collective operation, retrieves messages with the given tag and applies operation *reduceOp* to them, returning the result

The *reduceOp* is of the *ReduceOp<Writable>* type, which can be implemented by the user. A number of reduction operators are provided by default, these being: SUM, PROD, MAX and MIN (for int, float and double types).

These functions just write messages to a messages buffer, the actual transfer of messages is initiated just before barrier synchronization. A separate buffer is maintained for each other process in the current task. Since messages are written to the buffer as they are queued up in the function and the buffer is sent as a single TCP stream, the order of messages in the receive queue on the target process will be the same as the order they were sent in. This fits with the idea that the framework should relieve the user from worrying about race conditions.

Since messages need to be delivered to their destination before a process sends the *BARRIER ENTERED* packet, the process asynchronously sends the buffers then waits until all of them have been received via a blocking read from the destination process. Whenever a message buffer is received a confirmation packet is sent to the sending process. This is done to avoid a race condition where barrier synchronization has already completed, but the message buffer has not been received in it's entirety, which would allow processes to move on to the next superstep without having received messages that are expected to be there.

Chapter 4

Adapting Algorithms to NEWT

To illustrate the usability of this approach two scientific computing applications - conjugate gradient method (CG) [She94] and k-medoids clustering were implemented on NEWT as examples of how one would adapt algorithms to it's model. In previous work these particular algorithms were used to compare a number of BSP and MPI implementations [JSK13] to each other and this enables a direct comparison of NEWT's efficiency of to previous MPI results.

The algorithm descriptions detail the structural adaptations necessary to implement them with NEWT. For how these implementations might look in actual code refer to the simple pi estimator NEWT code in Figure 4.1. In addition, the source code for all algorithms used in this chapter is available online at [new13] in the *test* package.

4.1 Conjugate Gradient

CG is a rather complex iterative algorithm for solving sparse systems of linear equations and is outlined in figure 4.2. The parallel version of the algorithm can be implemented in the typical 'single program multiple data' (SPMD) fashion, with state of size N split among p processes as evenly as possible. The pseudocode in figure 4.3 presents CG (without preconditioning) implemented using MPI.

From algorithm 4.3 we can tell that if it were to be implemented in BSP style, each iteration would need to be split into several supersteps due to requirements of synchronizing a global dot product on two occasions and synchronizing overlapping portions of vectors between neighbors for matrix-vector multiplication. Taking these requirements into consideration, one possible segregation into stages is represented in figure 4.6 and is as follows:

- Init - initializes all needed state variables and checks how close the initial guess for x is to the actual solution

```
public static void main(String[] args) throws IOException {
    Configuration conf = NEWTConfiguration.createDefault();
    final int NUM_PROCS = 8;
    Input input = new NullInput(NUM_PROCS); //no input needed
    conf.set("newt.output.dir", "/out");
    NEWTJobMaster<NullState> am =  //no state to keep track of
            new NEWTJobMaster<NullState>(conf, NullState.class, input);
    am.addStage("map", new Stage<NullState>() {
        @Override public String execute(BSPComm bsp, NullState state) {
            double sum = 0.0; int samples = 1000;
            Random random = new Random();
            for (int i = 0; i < samples; i++)
                sum += Math.sqrt(1 - Math.pow(random.nextDouble(),2));
            bsp.send(0, new DoubleWritable(sum/samples), "pi/4");
            return "reduce";
        }
    });
    am.addStage("reduce", new Stage<NullState>() {
        @Override public String execute(BSPComm bsp, NullState state) {
            if (bsp.pid() == 0) {
                DoubleWritable result =
                        bsp.getReducedValue("pi/4", Reducer.DOUBLE_SUM);
                result.set(4*result.get()/NUM_PROCS);
                addToOutput("result", result, bsp); //write output to HDFS
            }
            return Stage.STAGE_END;
        }
    });
    am.run();
}
```

Figure 4.1: Simple pi estimator implemented with NEWT

- Start of Loop - defines the beginning of the loop (containing operations up until the first dot product), to serve as a reference point in flow control, at the end messages need to be sent to all processes, containing the partial dot product and error value
- Check Ending Condition - completes the computation of dot product and current error value, the latter being used to decide whether the algorithm should be finishing or the main loop should continue, returning the label of one of the two possible stages
- Continue Loop - prepares the vector p for the subsequent matrix-vector multiplication, sending the required overlapping portions of this vector to neighboring processes

Input: b, A and initial guess for x
Output: better approximation for x
$r \leftarrow b - Ax$
$err \leftarrow error(r)$
while $err > threshold$ and $iter < max$ **do**
  $z \leftarrow r$
  $\sigma_{old} \leftarrow \sigma$
  $\sigma \leftarrow z \cdot r$
  $p \leftarrow z + \frac{\sigma}{\sigma_{old}} p$
  $q \leftarrow Ap$
  $\gamma \leftarrow \frac{u}{p \cdot q}$
  $x \leftarrow x + \gamma p$
  $r \leftarrow r - \gamma q$
  $err \leftarrow error(r)$
end while
return x

Figure 4.2: Conjugate gradient method of approximating solution to $Ax = b$.

- Do MatVec - receives the overlapping vector pieces and completes matrix-vector multiplication, then computes another partial dot product, sending the result to all processes
- End of Loop - completes the computation of dot product from the received partial ones and computes the new error value, then returns the label of the 'Start of Loop' stage
- Stop - the stage that is executed when the error gets below the required margin or the maximum iteration count is exceeded in the 'Check Ending Condition' stage, completes the computation and writes the output to disk

When written as a NEWT program (as shown in algorithm 4.5) the structure is essentially the same as the MPI version, but with the distinction that the code is split into several functions, akin to a MapReduce program. Figure 4.4 shows an excerpt from the implementation of CG, showing only the synchronization of state between stages.

4.2 PAM k-medoids Clustering

Partitioning Around Medoids (PAM) is an iterative clustering method that divides a set of observations (2D points in our case) into k clusters based on the pairwise distances between them. The algorithm consists of the following steps:

```
//initializing MPI and state variables, omitted
PID = MPI_Comm_rank()
NPROCS = MPI_Comm_size()
r = b − Ax
err = norm(r)
while (error > TOLERANCE && it <= MAX_IT) {
        z = r
        σ_old = σ
        σ = dot(z, r)

        //find global value of sigma and the maximum error
        MPI_Allreduce(σ, 1, MPI_SUM)
        MPI_Allreduce(error, 1, MPI_MAX)

        if (it == 0)
          p = z
        else
          p = z + (σ/σ_old) p

        //synchronize required overlapping portions of
        //vector p for matrix−vector multiplication
        if (PID != 0)
          MPI_Send(p[0], post.size, PID−1)
        if (PID != NPROCS−1)
          MPI_Send(p[p.size−pre.size], pre.size, PID+1)

        if (PID != 0)
          MPI_Recv(pre[0], pre.size, PID−1)
        if (PID != NPROCS−1)
          MPI_Recv(post[0], post.size, PID+1)

        q = A · p using pre and post overlaps in the calculations
        γ = (σ/dot(p,q)) p

        //find global value of gamma
        MPI_Allreduce(γ, 1, MPI_SUM)

        x = x + γp
        r = r − γq

        it = it + 1
        error = norm(r)
}
//collecting results, omitted
```

Figure 4.3: Pseudocode of CG implemented using MPI

```java
public static class BeginLoop extends Stage<CGState> {
    @Override
    public String execute(BSPComm bsp, CGState state) {
        //calculations
        bsp.sendAll(new DoubleWritable(state.u), "sigma");
        bsp.sendAll(new DoubleWritable(state.error), "error");
        return "checkCondition";
    }
}

public static class CheckCondition extends Stage<CGState> {
    @Override
    public String execute(BSPComm bsp, CGState state) {
        state.u = bsp.getReducedValue("sigma", Reducer.DOUBLE_SUM).get();
        state.error = bsp.getReducedValue("error", Reducer.DOUBLE_MAX).get();
        if (state.error > TOLERANCE && state.it <= MAX_IT) {
            //calculations
            return "doMatVec";
        } else {
            //calculations
            return Stage.STAGE_END;
        }
    }
}
```

Figure 4.4: Synchronization of CG state in the NEWT program

- Randomly select k objects as the starting medoids
- Associate each object to the closest medoid, forming k different clusters
- For each cluster, recalculate its medoid m:
  - For each object o in the cluster with medoid m, swap m and o and compute the cost of the cluster
  - Select the object o with the lowest cost as the new medoid for this cluster
- Stop once medoids no longer change

This procedure of dividing objects between clusters and recalculating the medoids is repeated until the medoids no longer change positions between the clusters, at which point the clustering has become stable and is concluded. In the parallel implementation each process handles k/p clusters, where p is the number of parallel processes, this means the medoids need to be synchronized after they are recalculated and each process has to send points that are closer to medoids it does not handle to the corresponding process.

The segregation into stages of the algorithm is rather straightforward and follows the algorithm's description very closely, as is displayed in figure 4.7:

```
func init(state, comm)
        state.r = state.b − state.A · state.x
        return "beginLoop"

func beginLoop(state, comm)
        if (state.it > 0)
          state.γ = comm.getReducedValue("gamma", SUM)
          state.x = state.x + state.γstate.p
          state.r = state.r − state.γstate.q

        state.z = state.r
        state.σ_old = state.σ
        state.σ = dot(state.z, state.r)

        comm.sendAll(state.\sigma, "sigma")
        comm.sendAll(state.error, "error")
        return "checkCondition";

func checkCondition(state, comm)
        state.σ = comm.getReducedValue("sigma", SUM)
        state.error = comm.getReducedValue("error", MAX)

        if (state.error > TOLERANCE && state.it <= MAX_IT) {
          if (state.it == 0)
            state.p = state.z
          else
            state.p = state.z + (state.σ / state.σ_old) state.p

          if (comm.PID != 0)
            comm.send(p[0], post.size, comm.PID − 1)
          if (comm.PID != NPROCS − 1)
            comm.send(p[p.size − pre.size], pre, comm.PID + 1)
          return "doMatVec"
        else
          output results
          return STAGE_END

func doMatVec(state, comm)
        if (comm.PID != 0)
          pre = comm.move(comm.PID − 1)
        if (comm.PID != comm.NPROCS − 1)
          post = comm.move(comm.PID + 1)

        state.q = state.A · state.p  using pre and post overlaps
        state.γ = (state.σ / dot(state.p, state.q)) state.p

        comm.sendAll(state.γ, "gamma")
        return "beginLoop"
```

Figure 4.5: Pseudocode of CG implemented using NEWT

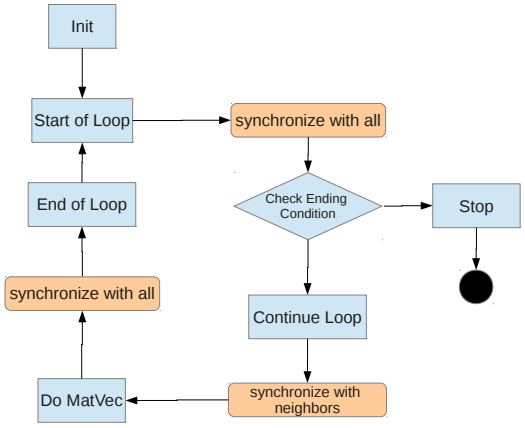

Figure 4.6: Structure of CG under the proposed model

- Init - initializes all needed state variables and randomly selects the initial medoids in each cluster
- Check Ending Condition - calculates the difference between the previous iteration's medoids and the current one, if the medoids did not change then move to 'Stop', otherwise proceed to 'Divide Points'
- Divide Points - finds to which medoid each point is closest to, and transmits points that belong to clusters handled by other processes
- Recalculate Medoids - finds new medoids in each cluster and sends the new medoids to each other process
- Stop - the stage that is executed when medoids are determined to no longer change in the 'Check Ending Condition' stage, completes the computation and writes the output to disk

These examples demonstrate when the serial programs need to be split into multiple functions. The most obvious one is the synchronization requirement, such as the computation of the global dot product after the 'Do MatVec' stage for CG. The second case stems from branching statements, such as the 'Check Ending Condition' stage, which can continue execution in two different directions. The last case in the given example is seen in the CG 'Start of Loop' stage, which serves as a reference point for code, which should be executed repeatedly, serving as a base for a loop structure in this case.

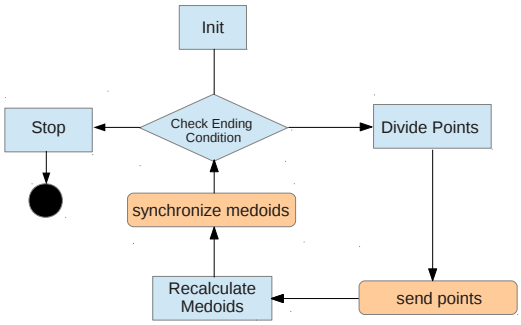

Figure 4.7: Structure of PAM under the proposed model

One simple optimization is to combine stages that are part of a loop, but do not require any communication between them, as the repeated intermediary barriers would cause unnecessary overhead. For example, the PAM 'Check Ending Condition' and 'Divide Points' would be merged in this fashion.

Chapter 5

Validation

Using NEWT implementations of algorithms described in the previous chapter a series of trials were conducted to validate the approach. The NEWT versions of these algorithms are compared to ones that use BSPonMPI - a BSPlib implementation that uses MPI for communication, which in previous work was determined to perform as good as pure MPI for the given algorithms [JSK13]. The test cluster composed of 16 Amazon's m3.xlarge (Standard Extra Large) instances, each with 15 GB of memory and 8 EC2 Compute Units (4 virtual cores), running Ubuntu Server 12.04 with Hadoop YARN 2.2.0 installed. Two types of trials were conducted: one to measure scalability and the other to assess overhead from creating state checkpoints and recovery in case of failure.

5.1 Measuring scalability

In the scalability trials, each of the algorithms was given input of size that was kept constant (a sparse system of 125000000 linear equations for CG and 250000 points across 64 clusters for PAM). Only the number of processes p was increased for consecutive trials.

The results in table 5.1 for NEWT include a $\sim$14 second overhead that is induced by YARN for initialization and allocation of process containers.

The scaling of the coarse-grained parallel algorithm (PAM) is slightly better than the BSPonMPI implementation when looking at the 4-core and higher results. It suggest that structuring the algorithm according to the model does not impose a significant overhead.

In the case of the fine-grained CG algorithm the scaling of NEWT is also better initially (2-16 cores) but starts to decline afterwards (32 and 64 cores). This is because the communication part of the runtime starts to significantly outweigh the computation part, resulting in a slightly worse result than BSPonMPI, which

	conjugate gradient	
p	NEWT	BSPonMPI
1	4476	4616
2	2225	2415
4	1245	1221
8	697	689
16	350	346
32	227	219
64	240	207

	k-medoids clustering	
p	NEWT	BSPonMPI
1	1889	1873
2	1248	1172
4	646	601
8	339	330
16	203	185
32	150	153
64	122	151

Table 5.1: Runtime comparison between NEWT and BSPonMPI (seconds).

leaves room for optimizing the implementation of the barrier synchronization.

The sequential version (on 1 node) of CG structured according to the NEWT's model consistently outperformed the sequential MPI implementations, the concrete cause of this is under investigation. A possible explanation is that this is related to the JVM being able to optimize the code better when certain algorithms are structured according to NEWT's model.

5.2 Measuring overhead

To measure the overhead imposed by storing checkpoints in HDFS enabled periodic checkpointing were enabled. The figures 5.1 and 5.2 show timelines for three kinds of executions of the implemented algorithms:

- without checkpointing
- with checkpointing enabled
- with checkpointing and recovery after a failure

To simulate failure in the cluster, one of the Amazon nodes was shutdown around halfway of the program's runtime. An additional node was added to the cluster, such that recovered processes may be started on it when one of the nodes goes down. The algorithms were executed on 16 Amazon m1.medium instances, each with 3.75 GB of memory and 1 virtual core.

From figure 5.1 it can be seen that the addition of checkpoints (every minute) for PAM had a negligible effect on performance, since the state kept by the algorithm is very small in relation to the amount of computation. The checkpoints consisted of floating point coordinates for the points belonging to clusters that were handled by any given process and any incoming messages, totaling under a

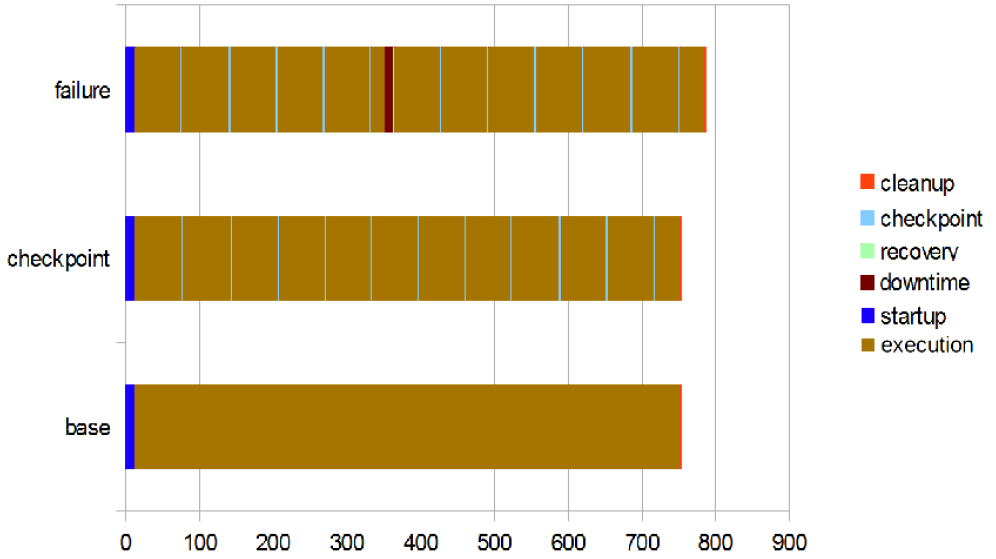

Figure 5.1: PAM performance after enabling checkpointing and on node failure

megabyte on average (much less than the entire address space of the program). The time it took to write such checkpoints to HDFS was approximately 30 milliseconds. When node failure occurred in the cluster, there was a period of downtime of around 10 seconds, which included the time it took for YARN to recognize the failure and allocate a replacement container. Following the period of downtime, there was a brief session of recovery, which consisted of reading the state of every process from the checkpoints and the reestablishment of socket connections between them, this also took under a second.

When the CG algorithm was distributed among 16 processes (figure 5.2), it required to store over 400 megabytes of data to HDFS during checkpointing by each process. Due to the size of checkpoints, the frequency of their creation was reduced to three and a half minutes. Storing these checkpoints took over 30 seconds and reading them from HDFS in case of failure took approximately half the time. Table 5.2 shows average times for the different types of overhead the framework imposes for the two tested algorithms.

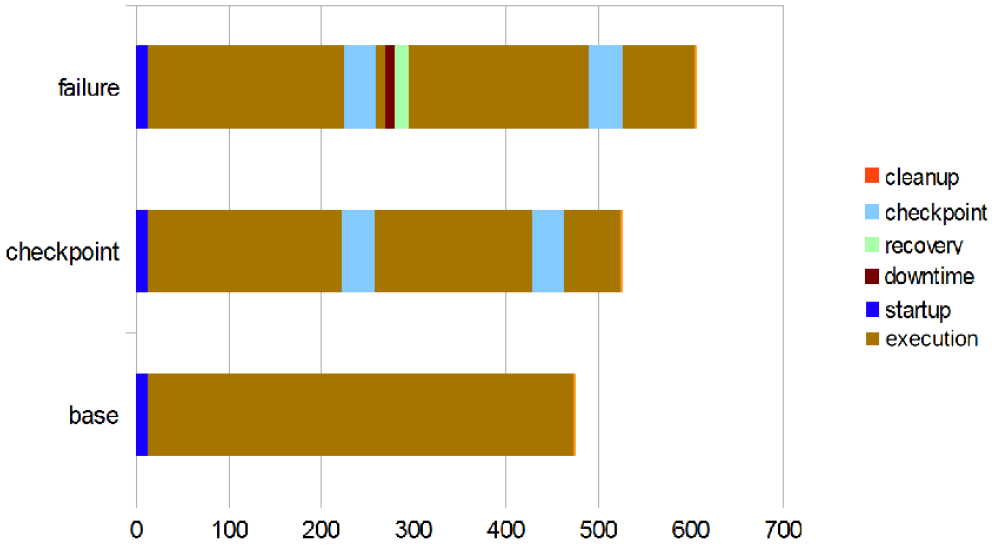

Figure 5.2: CG performance after enabling checkpointing and on node failure

5.3 Comparison to OpenMPI Transparent Checkpoint/Restart

This section presents comparison of size and creation time of NEWT checkpoints with ones created by applying the OpenMPI 1.5 transparent checkpoint/restart (CR) mechanism to the BSPonMPI program. The memory snapshots of the conjugate gradient algorithm done by BLCR 0.8.5 [Ber14] were around 470 MB in size, due to all of the program's memory pages being checkpointed, including those containing state which was not required to restart the NEWT processes, such as the immutable sparse matrix (generated on initialization). The average time to write these checkpoints to local storage was the same ($\sim$35 seconds) as time required for creating NEWT checkpoints in HDFS. When checkpoints were written directly to a mounted network file system (NFS) share (located on an additional m1.large Amazon instance) the time to finalize checkpoint creation exceeded 450 seconds. The PAM checkpoints created by BLCR were around 20 MB in size - significantly larger than NEWT checkpoints (for the same reason as the CG ones). The average time to write these to local storage was 3.3 seconds and directly to the mounted NFS share around 14.5 seconds, both significantly longer than then 0.1 second interruption to create NEWT checkpoints in HDFS.

These results tell us that in comparison to setting up a Hadoop cluster, significant time and resource investment is needed for creating a high throughput redundant storage system that makes the classic checkpoint/restart utilities work efficiently, such as setting up a IBM GPFS [IBM14] cluster instead of a single NFS server . This also shows that using Hadoop allows NEWT to efficiently make use of commodity infrastructure without much overhead. Due to the unstable nature of OpenMPI CR on the test infrastructure and the fact that continuation of processes needs to be performed manually, the restart functionality was not compared.

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

Table 5.2: Average overhead times (sec) imposed by NEWT

Chapter 6

Conclusion and Future Work

The work provides a summary of the current BSP-based distributed computing solutions and identifies several problems with their approaches. To counter these drawbacks a BSP-inspired parallel programming model that enables transparent stateful fault tolerance through checkpointing is presented. To validate the usefulness of the proposed model, the distributed computing framework NEWT is described and used to implement two wildly different computation kernels, which are used in validation experiments.

NEWT supports a larger range of applications than the current BSP implementations and utilizes Hadoop YARN to perform automatic checkpoint/restart of programs. The implementations of the computation kernels on the framework showed that it performs adequately for coarse-grained algorithms. However, results also show that the current barrier synchronization implementation could still be optimized for better support of very fine-grained algorithms. The current NEWT implementation's checkpointing time requirements were compared to BLCR's and determined that writing NEWT checkpoints to HDFS is as fast as writing BLCR checkpoints to local storage.

Future work also includes identifying common communication patterns and providing directives for executing them through the API, as well as providing an implicit way of handling input and output data. More advanced topics include implementing optional intelligent checkpointing strategies, such as hierarchical checkpoint/restart, and making collective operations more efficient by using communication patterns. Additionally, the framework could be used as the basis for a scripting language for parallel iterative algorithms, where a program in the envisioned language would be compiled into a NEWT program, transparently deriving the communication patterns and synchronization points.

The framework is open-source and available online [new13]. A scientific paper covering this work was accepted to "The 2014 International Conference on High Performance Computing & Simulation" [KS14].

Bibliography

[Ama14] Amazon Web Services. Amazon elastic mapreduce, May 2014. URL: https://aws.amazon.com/elasticmapreduce/.

[Apa14a] Apache Software Foundation. Apache giraph, May 2014. URL: http://incubator.apache.org/giraph/.

[Apa14b] Apache Software Foundation. Apache hadoop hdfs, May 2014. URL: http://hadoop.apache.org/docs/r2.0.5-alpha/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[Apa14c] Apache Software Foundation. Apache hadoop mapreduce, May 2014. URL: http://hadoop.apache.org/docs/current/.

[Apa14d] Apache Software Foundation. Apache hadoop yarn, May 2014. URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[Apa14e] Apache Software Foundation. Apache mina, May 2014. URL: http://mina.apache.org/.

[Ber14] Berkeley Future Technologies Group. Berkeley lab checkpoint/restart (blcr), May 2014. URL: http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR/.

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[BJOR99] Olaf Bonorden, Ben Juurlink, Ingo Von Otte, and Ingo Rieping. The paderborn university bsp (pub) library - design, implementation and performance. In *In Proc. of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP*, pages 99–104. Society Press, 1999.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[ELZ+10a] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, 2010.

[ELZ+10b] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, 2010.

[FD00] Graham E. Fagg and Jack 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*, pages 346–353, 2000.

[ham14] Hamster: Hadoop and mpi on the same cluster, May 2014. URL: https://issues.apache.org/jira/browse/MAPREDUCE-2911.

[HCSO12] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[Hil14] Jonathan Hill. The oxford bsp toolset, May 2014. URL: http://www.bsp-worldwide.org/implmnts/oxtool/.

[HMS+98] Jonathan M. D. Hill, Bill Mccoll, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library, 1998.

[IBM14] IBM Corporation. General parallel file system, May 2014. URL: http://www-03.ibm.com/systems/software/gpfs/.

[Ind14] Indiana University. Openmpi fault tolerance, May 2014. URL: http://www.open-mpi.org/faq/?category=ft.

[JSK13] P. Jakovits, S.N. Srirama, and I. Kromonov. Viability of the bulk synchronous parallel model for science on cloud. In *High Performance Computing & Simulation, 2013. HPCS '13. International Conference on*, 2013. (In print).

[KS14] P. Kromonov, I. Jakovits and S.N. Srirama. Newt - a resilient bsp framework for iterative algorithms on hadoop yarn. In *High Performance Computing and Simulation (HPCS), 2014 International Conference on*, July 2014. (Accepted for publication).

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, 2010.

[new13] NEWT bitbucket repository, November 2013. URL: https://bitbucket.org/mobilecloudlab/newt.

[Ope14a] Open Scalable File Systems. Lustre file system, May 2014. URL: http://lustre.opensfs.org/.

[Ope14b] Open Systems Laboratory. Ompi transparent checkpoint/restart, May 2014. URL: http://osl.iu.edu/research/ft/ompi-cr/.

[She94] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, 1994.

[SJV12] Satish Narayana Srirama, Pelle Jakovits, and Eero Vainikko. Adapting scientific computing problems to clouds using mapreduce. *Future Gener. Comput. Syst.*, 28(1):184–192, January 2012.

[Sui14] Wijnand J. Suijlen. Bsponmpi, May 2014. URL: http://bsponmpi.sourceforge.net/.

[SW13] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013. URL: http://ilpubs.stanford.edu:8090/1039/.

[SYK+10] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726, 2010.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

License

Non-exclusive license to reproduce thesis and make thesis public

I, Ilja Kromonov (date of birth: 11.11.1987),

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

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

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

   Fault Tolerant Distributed Computing Framework for Scientific Algorithms

   supervised by Pelle Jakovits and Satish Srirama,

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

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

Tartu, 26.05.2014