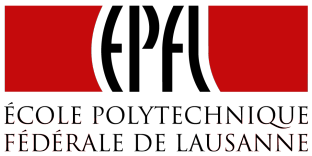
A Menthor for Graph Processing

Improving Parallel Graph Processing through introduction of Parallel Collections

Semester Project
II



by
Florian S. Gysin
Fall Semester 2011/12

Supervised by
Prof. Dr. Martin Odersky
Dr. Philipp Haller
Heather Miller

For further information about this work and the tools used or for an electronic version of this document feel free to contact the author.

Florian S. Gysin
florian.gysin@epfl.ch
www.barebulb.net

Ecole Polytéchnique Féderal de Lausanne
IC Faculty - Computer Science
Building INN
Station 14
1015 Lausanne

Abstract

Parallelization and distribution of algorithms have seen a lot of attention in these last years and a lot of effort was put into more efficient parallelized algorithms. A different aspect however, is how a user can be enabled to implement these kinds of algorithms in an easy and generic way. We worked with and improved MENTHOR, a SCALA framework for parallel graph processing. The focus was put on the different aspects of local parallelization over several processors with some thoughts on distributing MENTHOR and making it run on a cluster of networked machines. We improved the performance of MENTHOR significantly by introducing parallel collections and making several changes to certain parts of the MENTHOR code.

Contents

Acknowledgements

I would like to express my gratitude to everyone who supported me during the time I was working on this project. Only due to your support I managed to successfully complete his work!

First of all I want to thank my supervisors Heather Miller and Philipp Haller—this work would not have been possible without them. They supported me by providing interesting problems to solve, by giving input and motivation and by sacrificing a lot of time for discussing and working with me.

Aleksandar Prokopec for his experience and help with benchmarking Scala in general and parallel collections specifically.

Prof. Martin Odersky for giving me the opportunity to write this project at the LAMP Programming Methods Laboratory and for his inspirational lectures that contributed a lot to my ongoing interest in computer science.

All the students that accompanied me, not only during the time of this work but during my whole studies at EPFL, for the great time we had; be it over lunch, while learning or working in the students pool.

Chapter 1

Introduction

Parallelization and distribution of algorithms have seen a lot of attention in these last years. Especially in areas working with huge data sets and complex problems sequential processing will just no longer suffice. One of the fields affected is machine learning: machine learning algorithms have opened up new avenues of research and entirely new approaches to certain kind of problems. A limiting factor is however again the size of many meaningful problems, and the difficulty in parallelizing them.

Besides the task of devising and designing efficient parallel algorithms for specific problems, the task of implementing these respective algorithms can not be ignored. Implementing concurrent programs can be tricky and brings with it a lot of pitfalls which a researcher/developer needs to navigate. Although the benefits would be considerable, there seems to be certain reluctance in the machine learning community when it comes to full-out parallelization of their field of work [14].

This emphasizes the need and the importance of easy-to-use and high-level frameworks which researchers can use to implement their algorithms. The Menthor project aims to be such a framework.

1.1 Menthor in a Nutshell

Menthor is a framework written in Scala designed for parallel graph processing. The focus lies on a synchronous computational model which is generic and easily understandable in order to make the actual implementation of algorithms as simple as possible.

To quote the authors of Menthor:

> "Our goal in designing this framework is to enable researchers and practitioners to quickly implement and experiment with their algorithms in a parallel or distributed setting. We believe that a synchronous model which transparently distributes functional computations across cores (and eventually, machines in a cluster) is a first step towards this goal, by simplifying reasoning about program semantics."[7]

This section aims to give a short overview of Menthor and its functionality. For a more detailed description of the Menthor framework and its internals please refer to the work by P. Haller and H. Miller [7].

1.1.1 Computational model

Menthor operates using an hierarchical actor model. At the base of the computation lies the actual graph which is to be processed by whatever algorithm is implemented. The graph—an actor itself—consists of a collection of vertices, which are generic entities containing some generic data value (*e.g.* a floating point value, an integer value, *etc.*).

Vertices know about their neighbours, *i.e.* the other vertices they share a directional edge with, thus defining the graph.

The graph actor can be seen as the master actor (called 'master' from now on). It is responsible for creating worker actors (called 'workers' from now on) and for distributing the data, *i.e.* giving each worker a partition of the original graph. The master also synchronizes the 'supersteps' between the workers (see below) that are reminiscent of the Bulk Synchronous Parallel model introduced by Valiant et al. [16]. Furthermore, the master takes responsibility for collecting the result of the computation either directly by accessing the individual vertices, or through the use of 'crunch steps'.

One of the aims of MENTHOR is to hide the computational model of the parallelization and distribution from the user. Thus, from the perspective of a user of MENTHOR the described computational reasoning is somewhat different: although the computation follows the above model, a user of MENTHOR only needs to think in terms of a graph and the vertices therein. A user can thus focus fully on the actual graph algorithm he or she would like to implement. This is done via the `substep` function by defining how each vertices value will change over time.

Supersteps and substeps

Supersteps are synchronized by the master actor. Each worker actor will start superstep number i at the same time. Within any superstep a worker will perform a number of substeps. Substeps operate on graph vertices—possibly changing the state of the vertex—and produce outgoing messages to other graph vertices, which are collected by the worker. At the end of each superstep the messages produced by all vertices are sent to the worker containing the corresponding destination vertex. Before the next superstep begins, these messages are put in the 'inbox' of the destination vertex. Therefore, at the beginning of each new superstep, every vertex is assured to have all the messages produced by other vertices *during the last superstep.*

Crunch steps

Crunch steps can be seen as MENTHOR's counterpart to the 'reduce' phase of the Map Reduce paradigm. As the name—and the reference to the reduce phase—imply, crunch steps basically perform a 'reduce' operation as it is known in functional programming: a crunch step aggregates a single result over all vertices. Furthermore this result is sent as a message to all vertices and is thus available to each vertex at the beginning of the next superstep.

1.2 Parallelization vs. Distribution

There exist two ways of increasing performance through concurrent processing. The first is to locally parallelize the computation over several processors/cores[1], the second is to distribute the computation over multiple interconnected machines in a network or cluster.

These two ways of concurrently attacking the same problem are not fundamentally different from each other *in theory.* In practice however, there is a huge difference between local and remote concurrent algorithms, *e.g.* the issue of shared versus distributed memory. This leads to the fact that the same implementation mechanics can be very efficient in one case, but not in the other.

Our long-term objective with MENTHOR is to have a framework which provides both these features and combines them in a useful way: the problem can be distributed over

[1] In this work we will not distinguish between single-core processors and hyper-threaded cores on (physical) multi-core processors.

nodes in a cluster, and on each node the available sub-problem will be locally parallelized over the available cores.

1.2.1 Terminology

In this work we make a clear distinction between the terms 'parallelization' and 'distribution'. When we talk of *parallelization* we mean processing a problem in a way, such that multiple processors/cores on some local machine are involved, this generally means that these processors have access to shared memory and—in our case—that the code runs within the same JVM[2]. When we talk about *distribution* we mean processing a problem on different nodes or machines, which are part of a cluster or otherwise connected over a network.

1.3 Contributions

Contributions of this report are as follows:

- We argue in favour of two-level concurrency of graph processing algorithms in order to make the most of current hardware systems.
- We start out with proof-of-concept implementations of the Page Rank algorithm which are parallelized; we improve the performance through the introduction of parallel collections as introduced in Scala 2.9.
- Benchmarking results are presented which show that the proof-of-concept implementation can profit hugely from efficient local parallelization.
- We explore the use of different data types in the node-locally parallelized code and discuss their impact on runtimes.
- We discuss problems and solutions for a future distributed implementation of the Menthor framework.
- A final parallelized version of Menthor is presented which displays a *considerable* speed-up over the existing version.

1.4 Structure of this Report

The remainder of this report is structured as follows: Chapter 2 discusses background and related work. Chapter 3 describes the efforts undertaken to improve the local parallelization of Menthor and increase its local efficiency. parallel collections are introduced and we also investigate implementation details as data types and their impact on runtimes. Chapter 4 contains an evaluation of the benchmarking results that compare the new computation modes and the improved parallelization to the existing solutions. Also several special aspects of the parallel implementation are tested with more benchmarks to identify optimal solutions. Chapter 5 explores the topic of distributing the Menthor framework over multiple nodes in a network, we discuss issues that arise and suggest how to resolve them. Eventually, we conclude in Chapter 6 with remarks on future work.

[2] Java Virtual Machine

Chapter 2

Background and Related Work

2.1 Previous Work

The MENTHOR graph processing framework was created by Philipp Haller and Heather Miller [7]. It provides the user with an API to parallelize graph processing task in an easy way, without relying on inversion-of-control style programming.

MENTHOR's programming model uses event-driven actors. These are an efficient and light-weight abstraction for parallel programming models in virtual machines that do not require an explicit means to manage the execution state of a program [8].

The SCALA Actors library unifies event-based and thread-based actors to provide a full high-level framework for actor based programming models. SCALA actors are light weight and provide strong integration with existing threading models of mainstream VM platforms [9].

There have been previous efforts to distribute the MENTHOR framework, foremost by Georges Discry et al. [5]. He produced a distributed version of MENTHOR based on AKKA actors 1.1 [10], that provided good solutions for issues which arise during distribution of a project like MENTHOR. There were however issues with the generality of this solution when it came to merging the innovations into the main MENTHOR branch.

Given the difficulties encountered in the approach of Discry, SCACS [13]—Scala Cluster Service—has the potential of making it significantly easier to distribute MENTHOR since it provides the user with an intuitive and easy to use API to distribute tasks over a cluster of computers. We used SCACS while investigating the distribution of MENTHOR and while identifying issues which a distributed version has to address.

2.2 Related Work

Map Reduce is a framework introduced by Google in 2004 to provide a simple way to design large scale distributed programming tasks [4]. The idea of Map Reduce is based on the *map* and *reduce* functions known from functional programming and provides a new functional abstraction for distributed algorithms.

The Map Reduce paradigm was also adopted in some parts by the machine learning community and has been used on a small subset of machine learning problems with considerable effort [3, 15]. Several parties have identified multiple drawbacks which make Map Reduce difficult to impossible to use in a machine learning setup or make computation inefficient [11].

HADOOP is a Java framework by Apache which provides Map Reduce functionality. It serves as a good example that it is possible and feasible to run large scale parallel and distributed computations using the Java Virtual Machine [1].

As a reference algorithm to benchmark the MENTHOR framework we chose the Page Rank algorithm which was introduced by Page et al. [2]. The reason we chose this algorithm is that it is widely known and actually used (in some variant) in very large scale graph processing tasks: Google employs it to rank search results using a system called PREGEL [12]. Further the actual algorithm is rather simple, making it easy to understand and reason about computations and message flows.

In an early sketch of this work we also studied the possibility of providing a native way in MENTHOR to perform matrix and other linear algebra calculations. For this we studied SCALALA, a SCALA library for linear algebra functionality [6].

Chapter 3

Improving Parallelization

MENTHOR implements a hierarchical actor based model to parallelize computation locally. The first framework implementation relied on SCALA actors, a later implementation makes use of the AKKA actor library. Both versions already provide a parallel computation model, after all this is the whole point of the MENTHOR framework.

It was however our opinion that the performance of MENTHOR could further be improved by making sure that the local parallelization is as efficient as possible. As a main step we introduce parallel collections into wide parts of MENTHOR. Parallel collections are a part of the SCALA standard library[1] and provide the user with an easy-to-use and highly efficient set of parallel operations on local collections.

3.1 Previous Parallelization

The first version of the MENTHOR framework uses SCALA actors. Using this version the Page Rank algorithm was implemented which calculates the Page Rank values over a set of homepages (in our case Wikipedia pages downloaded at `www.wikipedia.org`). This algorithm served as a usage example and reference implementation for a big part of the past MENTHOR development. The implemenation of the algorithm is thereby independent of the underlying version of MENTHOR as we try to stick with a fixed API. Our work also uses Page Rank as a reference algorithm implementation to benchmark and test our efforts to further improve the MENTHOR framework.

3.1.1 Scala actors

The previously existing implementation of MENTHOR relying on SCALA actors contains a bug which impacts its performances *in some cases* quite considerably. If the number of vertices in the graph is divisible by the number of cores on the current machine, the master creates as many workers as there are cores and distributes the graph data into equally sized partitions. However, if this is not the case, *i.e.* the number of vertices is not divisible by the number of cores, the master switches into a second mode where it creates as many workers as vertices; thus giving each worker a single graph vertex to handle.

This scheme presents two distinct problems:

- Firstly, the Page Rank algorithm which was chosen as a reference implementation is computationally relatively cheap. On each iteration each vertex does some very simple arithmetic operation on a hand full of floating point values. Creating and setting up a separate worker instance for each of these vertices constitutes a huge

[1] Since version 2.9.0.

overhead which is not at all justified by the computational (un-)complexity of the underlying algorithm.

- Secondly, it is not clear for a user which of the above modes is selected when he starts the computation. As choosing the mode of operation is decided internally by the master and depends on the number of vertices, the operation mode varies with the input size of the problem. This means that running two computations with input sizes A and B where A and B are very close to each other can still result in vastly varying run times if size A is divisible by the number of cores and size B is not.[2] This curiosity is reflected by the benchmarking results of the reference implementation which can be found in Section 4.2; a bigger input size does not always end up taking longer to compute than a smaller input size.

A first task of this work was to fix the above described problems to get clear and predictable performance results from MENTHOR. We will further refer to the 'old' version of MENTHOR still suffering from the described issue as SA-SEQ (Scala Actors using Sequential Collections).

3.1.2 Akka actors

A second version of MENTHOR that existed before the start of our work makes use of the AKKA actor library. AKKA actors provide basically the same API as SCALA actors, but have some advantages in configuration. Further, AKKA actors version 2.0 and further seem to promise an good way to distribute a computation over nodes in a network—please refer to Chapter 5 for more a more detailed discussion. Our work is based on this version of MENTHOR, *i.e.* relies on AKKA actors and not SCALA actors.

3.2 Introducing Parallel Collections

3.2.1 New operation modes

To resolve the above mentioned drawbacks in the way SA-SEQ version handles worker creation we decided it is best to trust the user with the control over how many workers are created in each setup. To achieve this we introduced a new API to MENTHOR, *Operation Modes*. For each graph to process with MENTHOR an operation mode can be set when the graph is initialized. According to this operation mode the master actor will then create a specific amount of workers and set them up with the correspondingly sized partitions of the graph data.

The new operation modes are the following:

Single Worker Mode In Single Worker Mode (SWM) the master actor creates only a single local worker containing the entire graph.

Multi Worker Mode In Multi Worker Mode (MWM) the master actor tries to make an educated decision about the optimal number of workers for the computation at hand. In the current version of MENTHOR this means creating as many workers as there are available processing cores on the local machine. The vertices are split evenly over all the created workers.

I Am Legion Mode In 'I Am Legion' Mode (IAL) the master actor creates one worker per vertex, *i.e.* each worker handles the computation and communication of a single vertex. This is only feasible for algorithms with require very expensive

[2] For example on a machine with 4 cores an input of A=1000 vertices will lead to four workers containing 250 vertices each, where input B=1001 vertices will lead to 1001 workers(!) each containing a single vertex. This is obviously quite a different situation, hence the run times are not comparable.

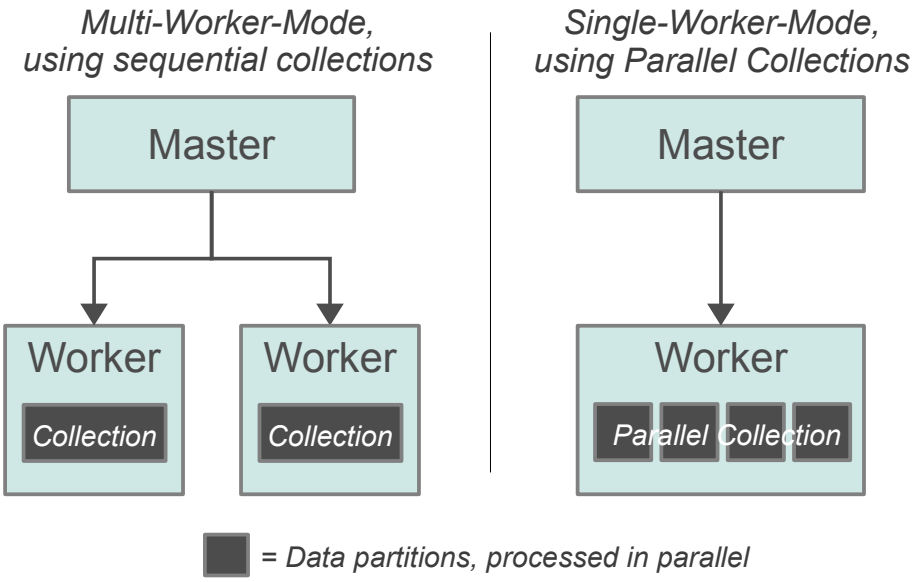


Figure 3.1: Multi-Worker-Mode using sequential collections on the left and Single-Worker-Mode using parallel collections on the right (both using Akka actors). The graph is partitioned in both cases and the partitions are processed in parallel.

computations on single graph vertices. (This is also the mode used in some of the SA-SEQ computations, see Subsection 3.1.1.)

Fixed Worker Mode In Fixed Worker Mode (FWM) the user specifies exactly how many workers are to be created by the master. This can be useful in specific situations (*e.g.* for testing/benchmarking purposes) but in general we suggest the user trust Menthor to figure out the correct number of workers for the most efficient computation.

3.2.2 Analysis

We performed benchmark measurements of the different operation modes and the different versions, *i.e.* with and without parallel collections. The cases benchmarked are the following:

SA-SEQ (as described in Subsection 3.1.1),

AA-SEQ-SWM (Akka Actors using Sequential Collections and Single Worker Mode)

AA-SEQ-MWM (Akka Actors using Sequential Collections and Multi Worker Mode)

AA-PC-SWM (Akka Actors using Parallel Collections and Single Worker Mode)

AA-PC-MWM (Akka Actors using Parallel Collections and Multi Worker Mode)

After initial tests and a preliminary analysis we decided not to further pursue the setups which employ the IAL mode. For reasons described above (see Subsection 3.1.1) this operation mode is highly inefficient in the case of the Page Rank algorithm. The run times are therefore not comparable to those of other implementations.

For the above mentioned settings and operation modes we ran benchmarks to evaluate their respective efficiency. The most interesting modes are AA-SEQ-MWM, representing the prior way Menthor parallelized its local computation and the new mode

AA-PC-SWM, which puts the task of local parallelization with the parallel collections framework. These two modes of operation give us the most direct comparison between parallelization using actors and parallelization using parallel collections. For a detailed evaluation and analysis please refer to Chapter 4.

3.3 Adaptations for Parallel Collections

As we have shown with the preliminary analysis of the impact of parallel collections it is well worth it in terms of speedup to introduce this feature into MENTHOR. While working on rewriting part of the algorithm to enable the use of parallel collections we noticed a strong effect of certain specific parts of the source code on the overall runtime. We decided to delve into these issues more deeply to see if it was not possible to further enhance MENTHOR's power by tweaking the code which is run in parallel by the parallel collections API.

3.3.1 Substep Parallelization

The thing which stood out most in terms of impacting the runtime was the functionality to collect outgoing messages of all vertices on a worker in each superstep.

Each vertex goes through a number of substeps in each superstep. During these substeps it generates a number of messages to other vertices. These message need only be available at their destination vertex at the beginning of the next superstep. To maximize efficiency the worker responsible for a certain vertex collects all messages generated during a superstep and handles dispatching them to their destination vertex.[3]

Now, as the processing of the substeps over all vertices is exactly the computation which MENTHOR tries to parallelize in the first place, we need a way of concurrently collecting all vertices' outgoing messages. A schematic implementation of this code can be seen in Listing 3.1.[4]

Listing 3.1: Schematic design of the substep parallelization.

```
1
2  var allOutgoingMessages List[Message] = ...
3  var outgoingMessagesPerVertex : Array[List[Message]] = ...
4
5  allVertices.foreach{ eachVertex =>
6          ...
7
8          // Computation of substep of the vertex
9          val outgoing = eachVertex.substep.stepfun()
10
11         // Add outgoing messages of this vertex
12         outgoingPerVertex(i) = outgoing
13
14         ...
15 }
16 // Collect all outgoing messages into one collection
17 allOutgoingMessages = outgoingPerVertex.flatMap(x =>
       x).toList
```

[3] In particular this includes checking if the destination is remote, *i.e.* on a different worker (read: Akka actor), or local in which case the messages are directly passed to the vertex.

[4] Note, that this is not the actual SCALA code, but a simplified 'pseudo-code style' version.

A Note on the collection of messages

The code shown in Listing 3.1 contains a `foreach` which is run in parallel by the parallel collections framework. It is therefore important to pay attention when concurrently accessing objects which are defined outside of this foreach. To avoid race conditions (and thus lost messages) we decided to use a set of message collections, one for each vertex (line 3 in the source code of Listing 3.1). During the parallel `foreach` the messages of each vertex are added to 'its' message collection. Only after the parallel computation part is done—after the `foreach` that is—are all vertices' messages collected into one set of outoing messages of this superstep (line 17).

Impact of collection data types

We noticed that the data type of the collection of messages which is accessed from the parallelized code (*i.e.* the code within the `foreach`) has a big impact on the computation runtime. To get our hands on some real data we conducted benchmarks to find out which collection data types are best suited for the task at hand.

3.3.2 Analysis

Both the *inner* and the *outer* data type were varied during the benchmarking.[5] This allowed us to see an impact of

- adding/changing items in the *outer* collection,
- and collecting all items in *the inner* collection.

Times for data I/O, computation time and clean up were measured for different combinations of data types.

For a detailed analysis including benchmarks and graphs please refer to Section 4.3 in Chapter 4.

[5] In the source code in Listing 3.1 (line 3) the *inner* collection corresponds to the List of Message objects, whereas the *outer* collection refers to the Array containing said lists.

Chapter 4

Benchmarking & Evaluation

We performed benchmarks on different versions of MENTHOR, using different operation modes and different implementations of the local parallelization. As a reference algorithm to benchmark we chose the Page Rank algorithm[2] which was already implemented as a MENTHOR usage example. In this chapter we evaluate the new parallelization strategies introduced in Chapter 3.

4.1 How do we benchmark?

The actual timing information originates from within the SCALA code. We wrote a short SCALA Trait called `TicToc` which can be used by any class that needs timing information. `TicToc` provides a very simple API: the `tic` method starts a time measurement, the `toc` method terminates the last time measurement (the measurements thus behave like they're being pushed to a stack). Finally `TicToc` also lets the user print all the timing information or save it to a log file. To start our benchmarking runs we wrote short Python scripts. The Python scripts take care of setting the correct Java Virtual Machine settings (*cf.* Appendix, Subsection A.2.3) and start the actual SCALA run. The runs are repeated by the Python scripts several times over and all the timing information is saved to a log file, allowing us to calculate average runtimes over the different runs.[1]

4.1.1 Benchmarking environment

All benchmarks were run on a server machine called MTCQUAD in the EPFL network. The machine was chosen because it is fitted with 8 cores, making it thus very suitable as a candidate for local parallelization. Whenever possible benchmarks were run in the evenings or at night, care was taken to only benchmark when the machine is not under heavy use by other users. The exact specifications of MTCQUAD can be found in Appendix A, Section A.2.

A note on our input data

The data which was used to benchmark the Page Rank algorithm implementation originates from Wikipedia.org. As it consists of actual real-life web pages linking to each other, the complexity of the graph is distributed unevenly over the vertices, as certain vertices (*i.e.* pages) will tend to link to more neighbours (other pages) than others. This has the effect that we can not exactly predict how an increase in the problem input size will affect the runtime of the Page Rank algorithm: if we process twice the number of pages there is no guarantee that this second half of the graph will be of the same

[1] For our benchmarks we averaged the runtimes over 5 or 10 runs.

complexity (*i.e.* interconnectedness) as the first half, hence it might be easier or harder to process.

A note on input sizes

Due to the non-uniform distribution of complexity in the graph (*cf.* last paragraph) it was not deemed useful to display actual vertex numbers. The input sizes given in our benchmarks hence do not represent the number of vertices in the processed graph, but the number of lines read in from our input data. The number of lines roughly correlates with the number of vertices in a linear fashion. To still give the reader an idea of the numbers of vertices processed, Section A.3 lists the precise number of vertices for different input sizes.

4.1.2 Measurement of different processing phases

It is important to know what benefits we can expect by working with parallelization models and what will not be possible. Reading data from disk, for example, will not be improved by any kind of process changes.[2] Therefore, we measured three phases of the total Page Rank computation for all of our benchmarks:

I/O phase The I/O phase contains the code which reads the graph data from disk, initializes the data structures, and builds up the network of workers (if needed).

Computation phase The computation phase is concerned with the actual time spent on the algorithms computation. It starts at the time where the graph processing is started and ends when the algorithm terminates.

Clean-up phase The clean-up phase measures the rest of the time spent after the main computation is done. This includes collecting the results, creating human-readable output and displaying this output to the user[3].

In retrospect it proved very valuable to have applied this separation of measurements. Several issues were identified by comparing an implementations impact on computation with its impact in data I/O for example; this would not have been possible without detailed time measurements. For this reason we often show several different graphs for a certain benchmarking run, *e.g.* pure computation, pure data I/O and the total runtime. The clean-up phase is however comparably small and stays constant over any kind of input size or computational model, we do no longer bother with it (but it is part of the 'total runtime' of course).

4.2 Comparing Operation Modes

In Subsection 3.2.1 we introduced different operation modes to represent different models of local parallelization. As the main goal of this work was to improve the efficiency of local parallelization it is of course inevitable that we perform solid benchmarks to assure the efficiency of the different variants and compare them among each others.

Figure 4.1 shows the results of our first series of benchmarks. In this figure we see the computation time of the Page Rank algorithm for different input sizes and different operation modes[4].
We note several important things:

[2] Without also changing the storage hardware, *e.g.* by replacing hard disks it with a RAID system.
[3] In other words printing it to standard out.
[4] For an explanation of the different operation modes please refer to Subsection 3.2.1.

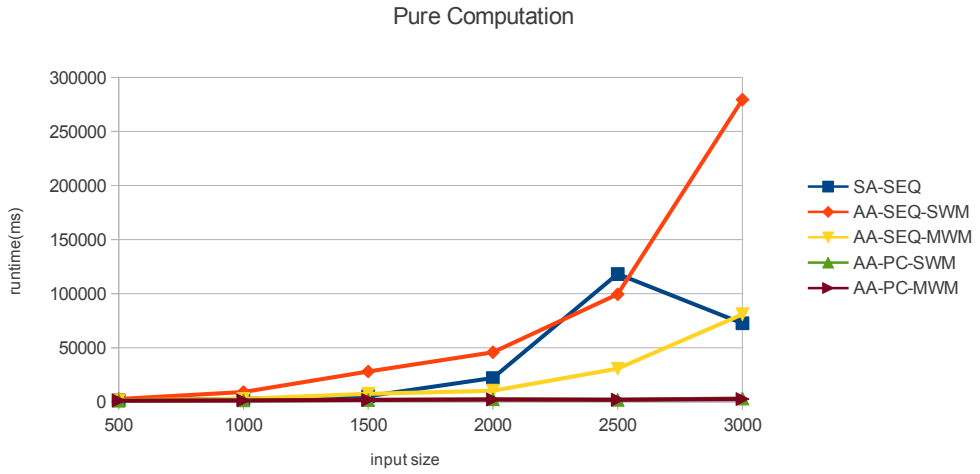


Figure 4.1: The computation time of different operation modes for different input sizes.

- SA-SEQ operation mode behaves in a strange way. An input size of 3000 actually takes *less* time to process than a smaller input size of 2500. This anomaly can be explained by internals of this older MENTHOR version, for a detailed explanation please refer to Subsection 3.1.1.

- As is to be expected AA-SEQ-SWM has the worst runtime. The explanation for this fact is obvious, as this operation mode does not employ any kind of parallelization using neither different workers nor parallel collections.

- The newly implemented computation modes AA-PC-SWM and AA-PC-MWM are both *considerably* faster than the existing implementations which rely purely on workers for parallelization. It seems that the introduction of parallel collections into MENTHOR was a sound decision that should lead to a remarkable overall speedup.

Figure 4.2 shows the data I/O benchmarks for the same operation modes as shown above. We see that I/O times of all previous MENTHOR versions are about the same. However we also see a strong increase in time spent on data I/O for the new operation modes employing parallel collections (AA-PC-SWM and AA-PC-MWM). For small input sizes this effect is negligible—it enlarges however with increasing inputs, up to a point where more time is actually spent on I/O than on the computation itself (not shown on graph). The issue of this data I/O inefficiency is investigated in Subsection 4.4.1 and the problem is solved.

Figure 4.3 shows the total runtime of the Page Rank algorithm over the different operation modes. Despite the introduced inefficiency in data I/O the new operation modes still feature a considerable speedup when compared with the existing modes.

4.3 Comparing Collection Data Types

In Section 3.3 we explained the changes which were made to the MENTHOR code in order to successfully integrate parallel collections. One of the changes involved the concurrent collection of messages which are sent from vertex to vertex. As explained in more detail in Subsection 3.3.1 messages are buffered in a *collection of collections*. The *outer collection* serves as a way to separate access to the *inner collections*, allowing us

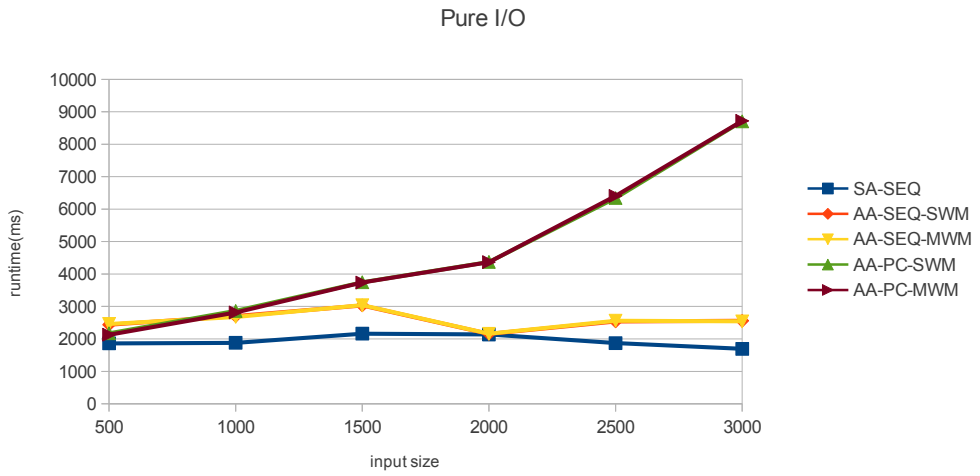


Figure 4.2: The data I/O time of different operation modes for different input sizes.

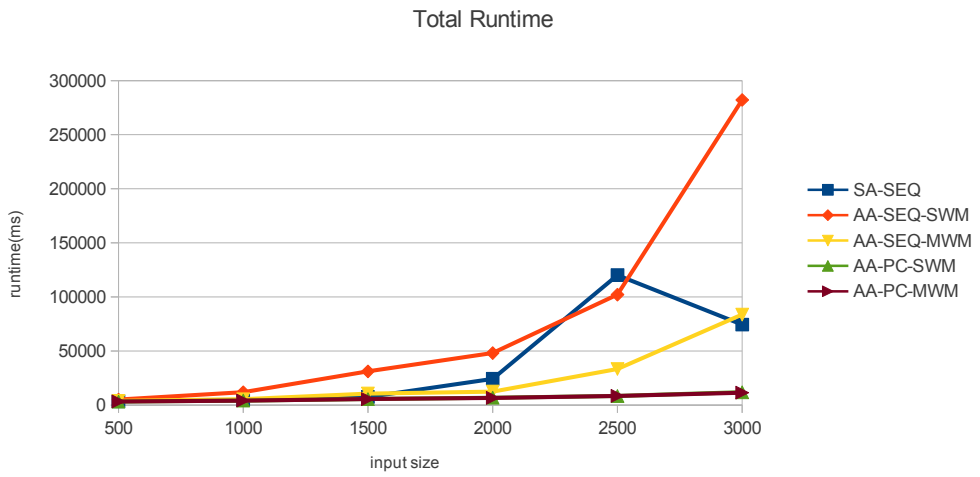


Figure 4.3: The total runtime of different operation modes for different input sizes.

to access different inner collections in parallel without worrying about race conditions or other concurrency problems.

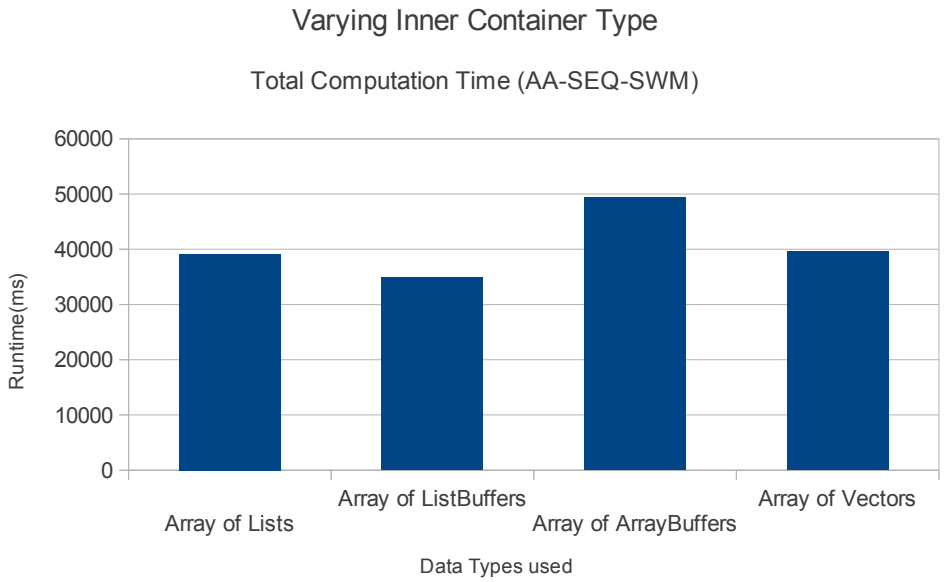


Figure 4.4: The computation time of the algorithm for different versions using different *inner* collection types to collect messages in parallel. (Input size = 2000)

As this new feature is woven through a big part of the worker computation code we thought it a good idea to perform benchmarks to evaluate which data structures, *i.e.* collection types, are best suited for the use case at hand. Both the *inner* and the *outer* collection data type were varied in these benchmarks, the results of which can be seen in Figure 4.4 and Figure 4.5.

As we can see in Figure 4.4, varying the inner collection type shows relatively little difference in the overall runtime. Still, the results differ enough to say that it is reasonable to use `ListBuffer` as the type of the inner collection.

Figure 4.5 shows the benchmarking results for the variation of the outer collection type. While `Array` and `ArrayBuffer` perform in a comparable fashion, `ListBuffer` gives us a horrible runtime. This also corresponds to our intuition, as random access on a large collection is a lot more efficient on arrays than it is on linked lists.

After taking into consideration the results shown in these graphs, we decided to use *Arrays of Lists* and *Arrays of ListBuffers* in the current versions of the Menthor framework.

4.4 A second iteration of refinement

The changes introduced in Section 3.2 and Section 3.3 already provide a significant speedup when compared with the original versions (classic SA-SEQ and AA-SEQ-MWM). While benchmarking AA-PC-SWM on different numbers of computing cores we discovered however that there are still further improvements to be made, some of which were directly implemented in the `parcol` branch of Menthor. This section covers some of these changes and tries to explain why the respective changes affect the computation time in the way they do.

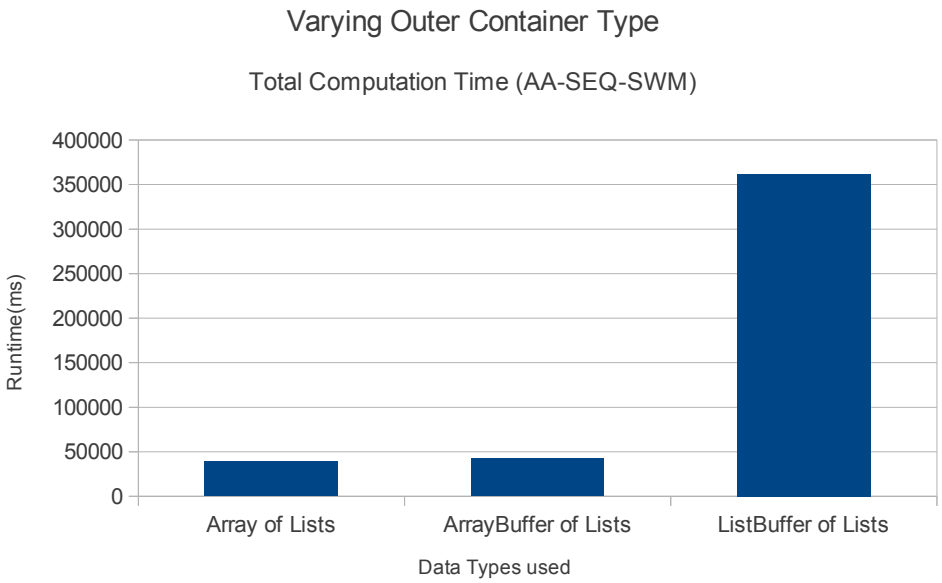


Figure 4.5: The computation time of the algorithm for different versions using different *outer* collection types to collect messages in parallel. (Input size = 2000)

4.4.1 Data I/O – building the graph

Adapting the source code for the introduction of parallel collections brought with it some changes, one of which was picking `GenSeq` as the type of the vertex collections in the graph and the workers. This is useful, as `GenSeq` is the lowest common supertype of parallel and sequential collections. This allows the developer to switch between different collection types without actually changing much else in the source code.

However, although the API provided by `GenSeq` allows to work with parallel and sequential collections, specifics about the efficiency of certain features are very different. For example it is pretty efficient to create a collection of vertices of type `List[Vertex]` by prepending vertices one after the other. If the same is done with a `ParArray[Vertex]` on the other hand, prepending elements one at a time basically means copying the whole array on each addition of a vertex—resulting in a horrible time complexity.

This effect was responsible for the big increase in data I/O time which is visible on the first series of benchmarks in Figure 4.2. We fixed this problem by introducing a `ListBuffer[Vertex]` in the master which is used during the I/O process to build up the graph of vertices. It is to this buffer collection that vertices are added while their respective data is read in from disk. Once all vertex objects are created and the graph is thus complete, the collection is transformed into a `ParArray` and is thereby parallelized. The introduction of this temporary buffer collection fixed the problem which was introduced by switching to `ParArray` as the main type to hold vertex collections.

The effects described can be seen in Figure 4.6. It shows the difference between data I/O time of the inefficient version versus the fixed data I/O version. The curve for the unfixed version shows the beginning of an exponential development, this corresponds to the (horrible) time complexity of $O(n^2)$ for building up an Array one-element at a time.[5] As was to be expected, the fixed part is considerably faster, not even showing a noticeable increase in data I/O time for inputs of used sizes.

[5]Building an array of length n means allocating a size 1 array and writing the first element; then allocating an array of size 2, copying the first entry, writing the second, *etc.* Adding elements $1..n$ calls for $\sum_{i=1}^{n}(i-1)+1$ or $\frac{n(n+1)}{2}$ write operations—n^2 in O-notation.

Figure 4.6: Comparison between data I/O time of the inefficient versus the fixed data I/O part. As was to be expected, the fixed part is considerably faster.

Figure 4.7 shows a comparison between computation time of the inefficient versus the fixed I/O part. We can see from this that the runtime is not affected by the introduced solution. This comes from the fact that the buffer collection is never referred to in the actual computation (and is most likely garbage-collected before it starts).

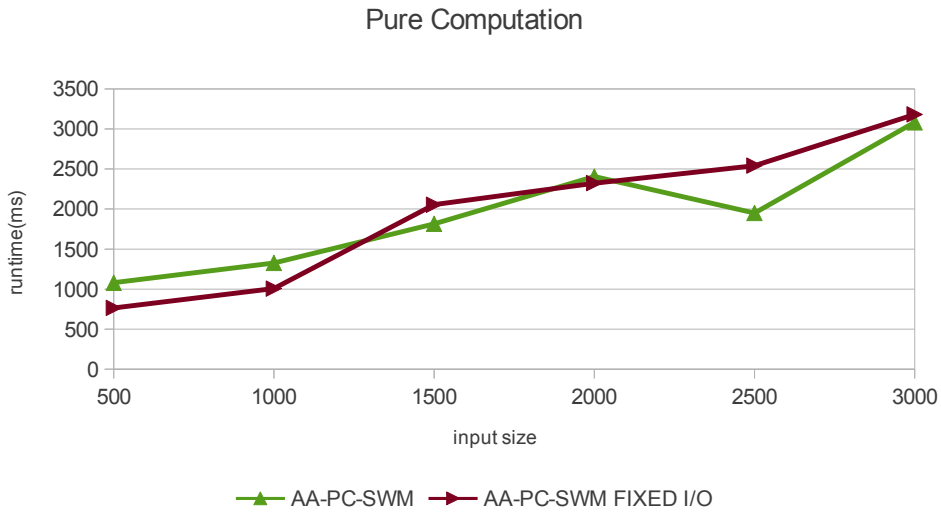


Figure 4.7: Comparison between computation time of the inefficient versus the fixed data I/O part.

To give an idea of the overall impact of the I/O-fix Figure 4.8 shows the total runtime of fixed versus unfixed version. Although the AA-PC-SWM version with 'broken' I/O is still faster than the previously existing MENTHOR modes like AA-SEQ-MWM, we can see that the impact of the introduced inefficiency *on the affected version* was indeed considerable. This emphasizes again the care which must be taken when choosing how to operate on different kinds of data structures.

The changes on the graph build-up described in this section belong to the most

important findings we achieved after a large series of benchmarks which set out to test different parts of MENTHOR.

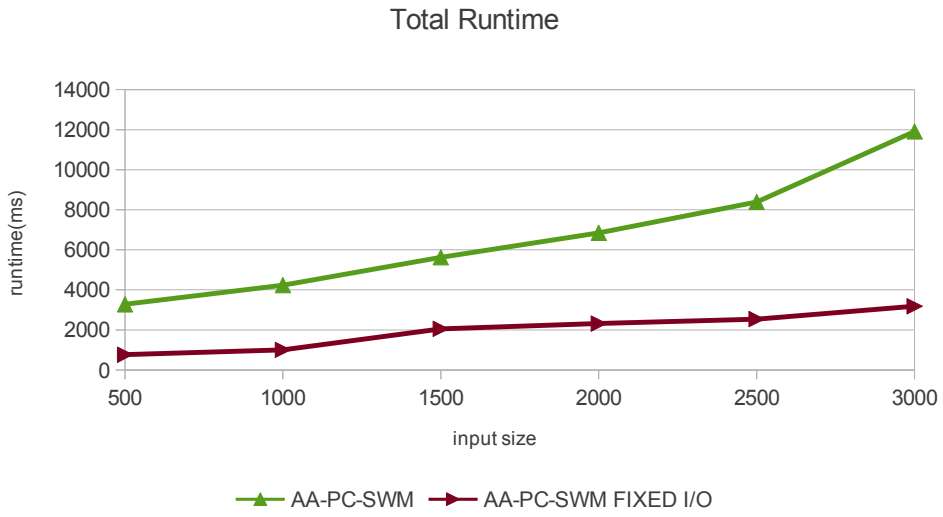


Figure 4.8: Comparison between total runtime of the inefficient versus the fixed data I/O part. As can be seen the performance is enhanced considerably by the latter.

4.4.2 Hashmap typing

Each vertex sends out messages during the run of an algorithm, in the case of Page Rank it updates his neighbours with its latest Page Rank value. Now, generally each worker is responsible for a rather large set of graph vertices. Therefore it is quite probable, that a good amount of messages sent from vertex to vertex will actually end up on the same worker. Sending messages using the AKKA framework is not as efficient as accessing objects locally—thus a special handling of local messages was introduced.

At the beginning of a new superstep the messages which were sent to a worker and ended up in its message queue are sorted into a hash map (*cf.* Listing 4.1, lines 8–11). Correspondingly, at the end of each superstep, when the messages are sent, each messages destination is checked. If it turns out that the destination of a message is actually the same worker the message is not passed via the AKKA framework but is put into a local hash map (*cf.* Listing 4.1, lines 15–21). In between these two events lies the actual computation where vertices access their incoming messages from the hash map.

In the previous versions of MENTHOR the hash map had the type `HashMap[Vertex, List[Message]]`. As can be seen from the linked source code in Listing 4.1 the build-up of each vertex' message list is done through creating new List objects which consist of the old incoming list plus the prepended message. This follows from the usage of the `List` type. Now, creating as many objects only to throw them away shortly after can be a waste of resources.

Listing 4.1: Schematic design of the message handling for local messages.

```
// Definition of the HashMap for worker-local messages
var incoming = new HashMap[Vertex, List[Message]]() {
    override def default(v: Vertex[Data]) =
        collection.mutable.ListBuffer()
}
...
def superstep() {
    // Sort incoming messages from the queue into the HashMap
    while (!queue.isEmpty) {
        val msg = queue.dequeue()
        incoming(msg.dest) = msg :: incoming(msg.dest)
    }
    // Actual computation using the messages
    ...
    // Sort the outgoing messages
    for (out <- allOutgoing) {
        if (out.dest.worker == self) {
            incoming(out.dest) = out :: incoming(out.dest)
        } else {
            out.dest.worker ! out
        }
    }
}
```

We tried therefore to improve the performance of this part through the introduction of a new type for the hash map: `HashMap[Vertex, ListBuffer[Message]]`. Through the usage of the `ListBuffer` type we hope to cut away the unneeded creation of objects as each vertex's incoming list will only have to be created once—future messages can be appended to the `ListBuffer`, as it is a mutable collection.

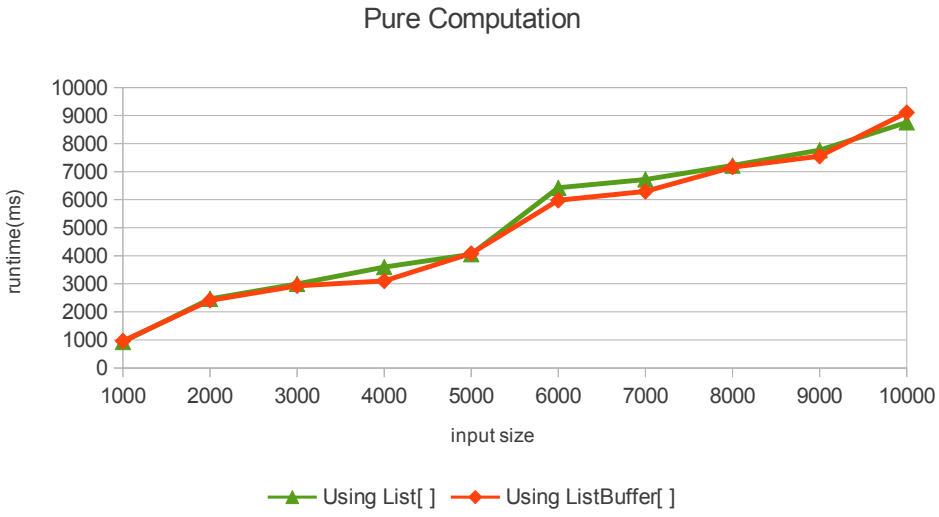


Figure 4.9: The impact of the HashMap typing on computation time.

In Figure 4.9 we display the results of benchmarks which compare the computation time using two different versions of hash map types, `HashMap[Vertex, List[Message]]` and `HashMap[Vertex, ListBuffer[Message]]`. As we can see we achieve no clear improvement of the computation time by choosing one map type over the other.

This does not correspond to our intuition which told us that the version using `ListBuffer` types should be more efficient than the version using `List` types. We try to make an educated guess about why this is case.

Reasons as for why we were not able to achieve our goal in this regards could be the following:

- Most Wikipedia webpages are linked to by a rather small number of other pages. Thus, generally the number of messages per vertex and therefore the size of its incoming list are not that large, which limits the effect of building up the list.

- The overhead through creating and garbage-collecting `List` objects is not as big as initially assumed.

- The MTCQuad machine used for benchmarks and the applied JVM settings provide us with a lot of memory for our computation. Although we tried to use bigger input sizes than in other benchmarks, we may still not have exhausted the available memory. As garbage collection gets more important if the available memory is *not* sufficient, the effect we expected to see could become visible for certain cases which exhaust the memory available—*i.e.* for computations using small amount of memory or very large input sizes.

4.4.3 Limiting the number of threads

The parallel collection framework generally takes control of setting up the number of threads used for parallel computations on a parallelized collection. There is however an API—one might call it a 'dirty hack'—that also allows the user to manually limit the 'parallelism' of a parallel collection scenario, *i.e.* the number of threads the operating system or rather the JVM assigns to the task. This can be done through a command which sets the *default parallelism* value for fork join tasks, as seen in Listing 4.2.

Listing 4.2: How set the number of threads for parallel collections. The parameter `i` would be the integer giving the number of threads.

```
1 scala.collection.parallel.ForkJoinTasks
2 .defaultForkJoinPool.setParallelism(i)
```

We used the above option to benchmark different configurations, the results thereof can be seen in Figure 4.10. The range tested was from one to seven threads, as a comparison we also included the results for 'unmodified'—*i.e.* not manually altered—parallelism. The latter is shown under *eight* threads in the graph, which is what the parallel collection framework would devise by itself.

The benchmark was run for the same versions also used in the benchmarking of the hash map typing (*cf.* Subsection 4.4.2). At the time we had the suspicion that, the garbage-collection would play a bigger role in the hash map case employing `List` types and would thus impact performance. The results tell us otherwise, the hashmap type still shows no significant influence on the computation time.

Nota bene It is worth noting that the decrease in performance, *i.e.* the increase in runtime is over-all very small when we decrease the number of parallel threads using the parallelism setting. One possible explanation could be that computation bottlenecks exist in the sequential code that are not parallelized and are thus also unaffected by the changed number of threads. This is a topic that would be interesting for future investigation; could such sequential code parts be parallelized or optimized in some way, Menthor would benefit considerably.

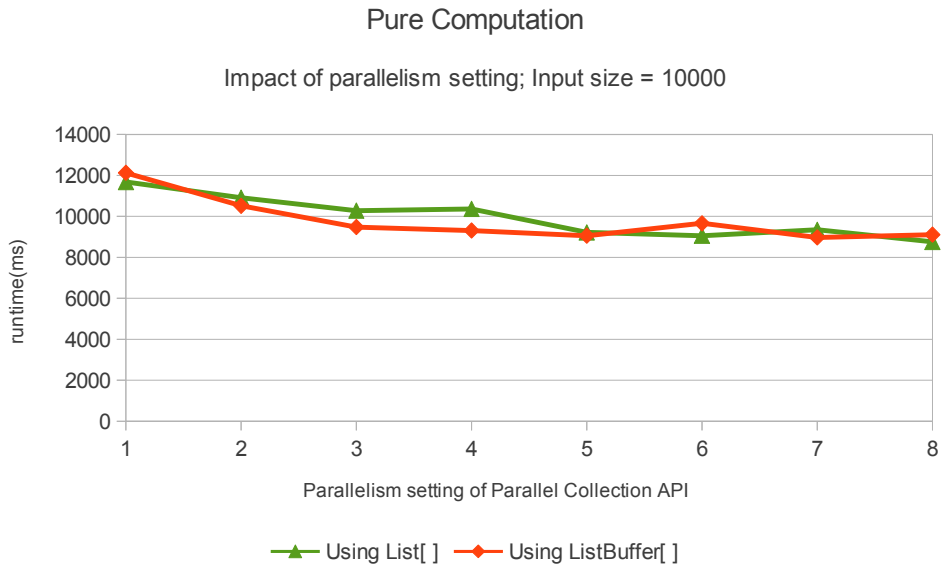


Figure 4.10: The impact of limiting parallelism via parallel collection API. The benchmark for '8' threads actually corresponds to an *unmodified* parallelism setting.

4.4.4 Hashmap to AtomicReferenceArray

In Subsection 4.4.2 we described the process of sorting incoming—and to some degree also outgoing—messages into a hash map. This sorting procedure which happens in two loops (*cf.* Listing 4.1, lines 8–11 & 15–21) constitutes the bulk computation on each worker *which is not yet parallelized*.

We assume that a lot of time could be further saved by optimizing this functionality. One approach which we contemplated—but due to time limitations were not able to fully implement—is the following:

Using an `AtomicReferenceArray`[6] one could forgo the hashmap altogether. In the current Menthor version it is possible to use an array because the number of vertices is fixed and known at the beginning of the computation without ever changing thereafter. Further an AtomicReferenceArray guarantees atomicity in accessing its stored values: this would allow us to do the sorting of the outgoing messages in each vertex' substep, *i.e.* within the parallelized part of the worker code. Also the code construct which was introduced to collect outgoing messages (described in Section 3.3) would become superfluous and could thus be removed.

We believe that a rewrite of the Menthor worker code according to this mentioned idea could result in a good improvement of performance—especially so as it tackles one of the portions of code that is not yet parallelized but runs sequentially. However, future plans for the Menthor framework include having dynamic graphs—allowing the user to add or remove vertices during computation—as many applications require this. This would prohibit the use of an array data structure[7], possibly making the above described changes infeasible.

[6] java.util.concurrent.atomic.AtomicReferenceArray
[7] Adding elements to an array is very expensive.

4.5 Final Version

We set out to create a new version of the MENTHOR framework which employs parallel collections and, more importantly, is faster than the existing ones. After taking into consideration the findings of the benchmarks we presented in this chapter we can say that we achieved this goal.

A final version of the latest MENTHOR branch was created using the AA-PC-SWM operation mode, *i.e.* AKKA actors, parallel collections and a single worker. In this version we also incorporated the various changes introduced through this chapter, *e.g.* the correction fixing the data I/O inefficiency. In the cases where we benchmarked different variations of the same code (*e.g.* for different data structure types), we naturally chose the variation giving us the best runtime.

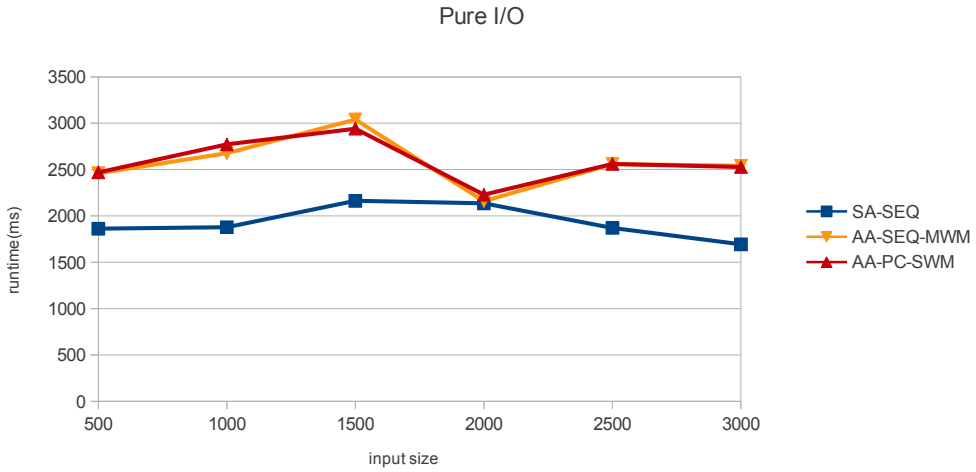


Figure 4.11: The data I/O time of the final MENTHOR version compared to previous implementations.

Figure 4.11 gives us the data I/O part benchmarks for the three main versions of MENTHOR, SA-SEQ being the first implementation, AA-SEQ-MWM the classic AKKA Actor version and AA-PC-SWM the latest final version using only one worker and parallel collections. We see that this part of the total runtime was not affected much with the changes introduced during the course of this work: the timings stay more or less the same. Furthermore, the data I/O part makes for only a small part of the total runtime, it is dwarfed by the computation part especially for larger input sizes.

In Figure 4.12 we see the speedup of the computation part achieved through the introduction of parallel collections (along with the accompanying adaptations and tweaks). The newly introduced version using parallel collections is considerably faster for any input size, more so the bigger the input is chosen. A similar picture can be seen in Figure 4.13 where we display the total runtime of the Page Rank algorithm for all three versions.[8]

[8] As the computation part makes for the lion's share of the total runtime, the graph looks very similar to the one showing the pure computation benchmarks.

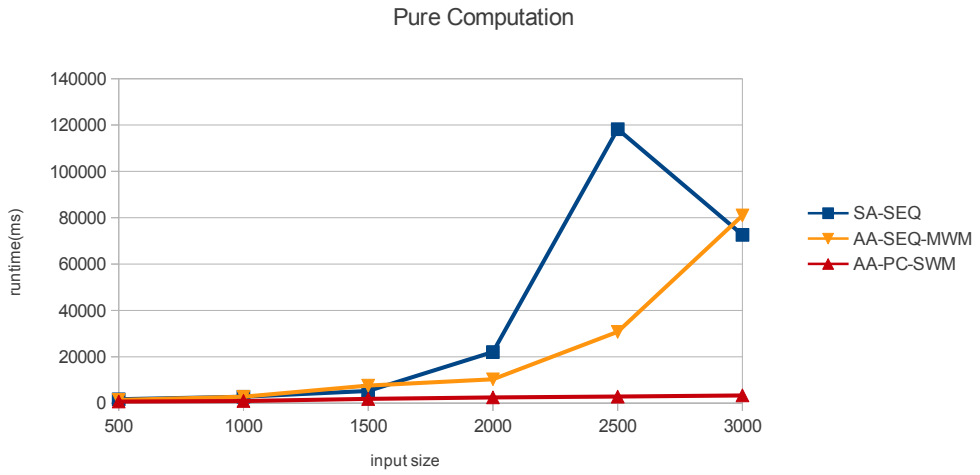


Figure 4.12: The computation time of the final MENTHOR version compared to previous implementations.

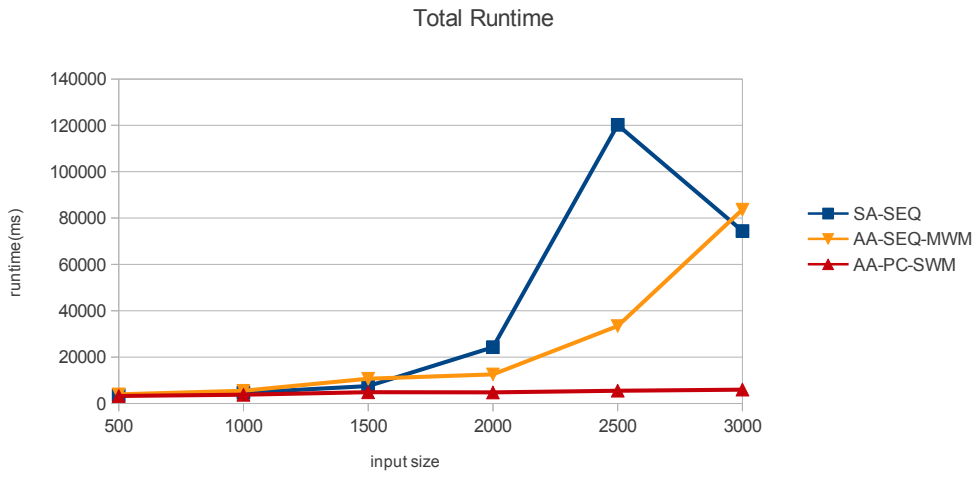


Figure 4.13: The total runtime of the final MENTHOR version compared to previous implementations.

Chapter 5

Distribution of Menthor

Besides node-local parallelization distributing the MENTHOR framework is an important step towards making most efficient use of modern hardware to gain the best performance possible. Distributing MENTHOR is however not a trivial task, this work makes strides in that direction.

5.1 Previous Distribution

There have been previous efforts to distribute the MENTHOR framework, foremost by Georges Discry et al. [5]. Their work provided a solid proof-of-concept implementation demonstrating the possibility and feasibility of distributing the MENTHOR framework. It is here we would like to continue in identifying and addressing key issues which arise while distributing MENTHOR. We hope that we can thus lay the ground work for a future distributed version of MENTHOR which is based on AKKA actors 2.0 (and beyond) and which also incorporates efficient local parallelization.

A new version of the AKKA actor library (version 2.0) is currently under development. This new version promises to bring a lot of improvements compared with the existing version 1.1 which is used in the current local and distributed versions of MENTHOR. One particular issue which is addressed is the use of AKKA actors in the setup of distributed computing. As AKKA 2.0 is not published at the time of writing, we thought it not reasonable to try to distribute MENTHOR using the old AKKA version.

5.2 Distribution Issues

While working towards a MENTHOR version that should one day be distributed we identified several core issues. These are issues that generally arise when distributing a program over a network, they are inherent to distributed computing. It is therefore important to recognize these problems as soon as they arise and to deal with them in an efficient manner.

5.2.1 Data serialization

One of the main issues when it comes to distributing any kind of program is serialization. Messages which were passed around in the JVM so far, and were stored in memory shared by the different local parallel threads, now have to be passed through a network. The process of serialization, *i.e.* transforming an instance of a class into a data stream to be propagated by network protocols brings with it some limitations upon the kind of objects which can be passed around. In our context, which is the context of SCALA code

eventually running in a JAVA Virtual Machine this means using classes that are 'serializable'. Similarly to the JAVA interface *Serializable* which is implemented by the classes who are to be serialized, SCALA provides a trait *Serializable*—again to be extended in custom classes. Implementing/extending above mentioned interfaces/traits will make the referencing class serializable—as long as the members of the class are themselves serializable.

Messages MENTHOR uses Message objects to send information around between different concurrent computing instances. In a distributed setup these are obviously exactly the objects which have to be serialized. Now, message objects are very simple wrappers that contain a generic data item of type T. JAVA and SCALA check for serialization during runtime, this means that in this case we can pass the responsibility of ensuring serialization to the user of the framework: as long as the type T of this data item is serializable, the message object will be serializable as well.

Vertices An other kind of data which is passed through the network are graph vertices, namely during the initialization of the distributed computation where the graph is set up and distributed over all computing nodes. Vertex objects are a bit more complicated to serialize than messages. Vertices also have a data member of generic type T that puts the burden of assuring serializability on the user. Aside from that they also have a list of *neighbours*, *i.e.* the graph vertices to which the current vertex has a directed edge. In the non-distributed case, where serialization is not an issue, these neighbours were are as a list of references to other vertex objects. The serialized version of the vertex class needs to ensure two things:

1. The collection class used to store the neighbour references is itself serializable—which is for example not the case with the common `List`—and

2. the references themselves can be resolved after deserialization.

The latter of these two points is addressed in the following subsection.

5.2.2 From references to lookups

Vertices do not only contain data items which are to be serialized, but also references to a variety of different actors in the graph computation. Namely these are a reference to the master actor, a reference to the actor holding the vertex and a set of references to the neighbours of the vertex in the graph. These references work of course fine to pass messages and method calls between objects. In a distributed environment however, special care needs to be taken in order to preserve these references in their functionality. The references can be categorized into two sets:

Akka Actor References The references to the master actor and the worker holding the current vertex are of this type. Luckily AKKA already provides a built in mechanism to handle remote actor references, the so called called actor registry. The actor registry allows to switch between `LocalActorRef` and `RemoteActorRef` through lookups using a unique identifier (UUID).

Menthor Vertex References The second kind of references concern the edge relations in the graph which is to be processed. Each vertex contains a set of references to the vertices to which there exist a directed edge in the graph.[1]

For the vertex references a distributed version of MENTHOR would need to provide some sort of lookup or directory functionality. A way to implement this would for example

[1] In the (local) parallelized implementations this is handled as a list of `Vertex` objects.

be a static singleton instance of a directory translating between local object references and global vertex identifiers. In a distributed setup of MENTHOR each remote node instance will be running in its own JVM, thus allowing it to have its own singleton directory. During the creation of the graph the master actor would have to take care of also building up the global directory master which is then distributed, or rather copied redundantly, to all remote node instances.

Chapter 6

Conclusion

We set out to create a new version of the Menthor framework which employs parallel collections and, more importantly, is faster than the existing versions. We can say with certainty that we have reached this goal.

In this work we showed that the Menthor graph processing framework benefits hugely from the use of parallel collections. When compared with reference implementations of the same algorithm in different versions/modes of Menthor which do not use parallel collections, speedups of up to a factor of *20* have been achieved. This suggests that parallel collections will play a definite role in future versions of Menthor and will take over most responsibility of node-local parallelization.

The Scacs framework was used to explore distribution of Menthor over multiple nodes. Instead of aiming for a fully distributed version, which would have exceeded the extent of this work, we identified key issues of distribution and presented possible solutions. As these findings are all based on the Menthor master branch, they should be of help for future projects concerned with the distribution of Menthor.

Further different parts of the Menthor source code were benchmarked to study the impact of small changes in the computational model on the runtime. We also compared the performance of different data structures in different use cases to choose the most efficient ones for the latest version of Menthor.

6.1 What is missing?

Menthor was benchmarked using an implementation of the Page Rank algorithm. It is important to keep in mind that different kinds of graph algorithms have very different requirements in terms of computation and communication in the processed graph. By choosing efficient and well-performing variants of Menthor implementations in the Page Rank case we can not automatically guarantee that the same choices will also be the most efficient for other algorithms using Menthor. For future benchmarking it would be interesting to implement more graph algorithms with Menthor and compare the benchmark results of these algorithms against each other.

All the benchmarks were run on a single machine in the EPFL network, MTCQuad. It would be interesting to see how changing the hardware impacts different variants of the Menthor computation, for example by varying the amount of available memory or switching to machines with a very high number of processors.

6.2 Future Work

6.2.1 Moving Menthor to Akka 2.0

The Menthor framework is at the moment based on Akka version 1.0 and 1.1 (both are compatible actually). During the last stages of this work the Akka project published their latest version, milestone one of version 2.0. Although preliminary tests in incorporating Akka 2.0 were made, Menthor was not yet updated fully to use the new available features of Akka 2.0. In the future Menthor could however certainly profit of the latest additions and improvements of the Akka project.

6.2.2 Distribution

The main branch of Menthor still remains a node-locally parallelized version and does not yet provide distribution over multiple nodes in a network. To really capitalize on modern cluster and network hardware it is however inevitable to eventually come up with a distributed version, as any shared-memory parallel machine will only allow for that many cores/processors and that much available memory on a single machine.

Appendix A

Additional Information

A.1 Operation Modes

SA-SEQ (as described in Subsection 3.1.1),

AA-SEQ-SWM (Akka Actors using Sequential Collections and Single Worker Mode)

AA-SEQ-MWM (Akka Actors using Sequential Collections and Multi Worker Mode)

AA-SEQ-FWM (Akka Actors using Sequential Collections and Fixed Worker Mode)

AA-SEQ-IAL (Akka Actors using Sequential Collections and I Am Legion Mode)

AA-PC-SWM (Akka Actors using Parallel Collections and Single Worker Mode)

AA-PC-MWM (Akka Actors using Parallel Collections and Multi Worker Mode)

AA-PC-FWM (Akka Actors using Parallel Collections and Fixed Worker Mode)

AA-PC-IAL (Akka Actors using Parallel Collections and I Am Legion Mode)

A.2 MTCQuad Specifics

A.2.1 Hardware

The following hardware assets were available to us on the machine MTCQuad which we used for benchmarking.

CPU 4 x Dual-Core AMD Opteron(tm) Processor 8220 SE

CPU Speed 2800 Mhz

CPU Cache 1024 KB

Memory 16 GB

Swap 16 GB

A.2.2 Software

The following software was installed and used on the machine MTCQuad which on which we run our benchmarks.

Operating System Ubuntu 6.06 'Dapper Drake' LTS

Scala Version Scala code runner version 2.9.1.final – Copyright 2002-2011, LAMP/EPFL

Scacs Version Development build from August 17th, 2011

Akka Version Akka Actors 1.1[1]

A.2.3 JVM Settings

The following JVM settings were used to benchmark the different Menthor verions on MTCQuad:

```
-Xmx4G -Xmx8G
```

These settings set the heap space to be between minimum of 4 and a maximum of 8 Gigabytes.

[1] We also experimented with Akka 1.0 and 2.0, these were, however, not used for the benchmarking process.

A.3 Number of Vertices per Input Size

Input Size	Number of Vertices
1000	6821
2000	11467
3000	18696
4000	23637
5000	27556
6000	30731
7000	34597
8000	39368
9000	49557
10000	59067

Table A.1: The number of graph vertices which are processed for certain input sizes.

Appendix B

Benchmarking Results

All run times in the following tables are listed in *milliseconds*.

B.1 Comparing Operation Modes

These benchmarks use the following collection data type for vertex substep message collection: `Array[List[Message[Data]]]` (*cf.* Section 3.3). The algorithm performed 30 iterations. All runtimes are given in milliseconds and were averaged over 5 independent runs.

Input Size	I/O	Computation	Cleanup	Total
500	1862.2	1694.6	27.2	3584
1000	1878.4	2774.6	49.4	4702.4
1500	2162.2	5291.4	66.8	7520.4
2000	2135.4	22080.6	74.2	24290.2
2500	1870.4	118219.2	96.4	120186
3000	1693.8	72624.8	120.4	74439

Table B.1: Runtimes for SA-SEQ mode with different input sizes.

Input Size	I/O	Computation	Cleanup	Total
500	2438.8	2717.4	23.2	5179.4
1000	2706.4	9135.8	46	11888.2
1500	3032	28098	66.8	31196.8
2000	2155.8	45931	71.8	48158.6
2500	2537	99499.8	95.8	102132.6
3000	2558.8	279437.8	119.4	282116

Table B.2: Runtimes for AA-SEQ-SWM mode with different input sizes.

Input Size	I/O	Computation	Cleanup	Total
500	2460.6	1427.4	26	3914
1000	2676	2776	47	5499
1500	3040.2	7562.8	68.2	10671.2
2000	2159.6	10318	73	12550.6
2500	2559	30750.6	97.6	33407.2
3000	2540.2	80987.8	118	83646

Table B.3: Runtimes for AA-SEQ-MWM mode with different input sizes.

Input Size	I/O	Computation	Cleanup	Total
500	2172.8	1080	29.2	3282
1000	2862.8	1326.8	47	4236.6
1500	3745.4	1815.4	66	5626.8
2000	4371.6	2404.8	74.6	6851
2500	6342.2	1950.8	97.4	8390.4
3000	8706.4	3082.2	120.4	11909

Table B.4: Runtimes for AA-PC-SWM mode with different input sizes.

Input Size	I/O	Computation	Cleanup	Total
500	2129.8	1217	26	3372.8
1000	2815.4	1332.6	50.4	4198.4
1500	3737.6	1818.2	66	5621.8
2000	4365	2256.4	74.6	6696
2500	6405.2	2037.4	96.6	8539.2
3000	8717.8	2677.2	122.2	11517.2

Table B.5: Runtimes for AA-PC-MWM mode with different input sizes.

B.2 Comparing Collection Data Types

The input size for the following benchmarks is fixed at 2000, the algorithm performed 30 iterations. All runtimes are given in milliseconds and were averaged over 10 independent runs.

Outer Container Type	I/O	Computation	Cleanup	Total
Array of Lists	2198.6	36868.2	71.8	39138.6
ArrayBuffer of Lists	2352.1	40189.3	71.3	42612.7
ListBuffer of Lists	2200.5	359267.5	72	361540

Table B.6: Runtimes for AA-SEQ-SWM with varying collection types for the outer vertex message collection.

Inner Container Type	I/O	Computation	Cleanup	Total
Array of Lists	2198.6	36868.2	71.8	39138.6
Array of ListBuffers	2160.6	32638.2	70.4	34869.2
Array of ArrayBuffers	2345	46968	71.5	49384.5
Array of Vectors	2204.1	37307.1	72.6	39583.8

Table B.7: Runtimes for AA-SEQ-SWM with varying collection types for the inner vertex message collection.

B.3 Final Version

Input Size	I/O	Computation	Cleanup	Total
500	2469.8	736.2	25.2	3231.2
1000	2771.4	955.6	48.8	3775.8
1500	2941.6	1850.4	70.6	4862.6
2000	2229	2500.6	78.6	4808.2
2500	2561.2	2850.4	106.4	5518
3000	2527.6	3358.2	124.2	6010

Table B.8: Runtimes for the latest MENTHOR branch. It uses the operation mode AA-SEQ-SWM and benefits from the various benchmark findings presented in Chapter 4.

List of Figures

List of Tables

Listings

Bibliography

[1] Apache. Hadoop MapReduce. http://hadoop.apache.org/mapreduce/. 5

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998. 6, 13

[3] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006. 5

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004. 5

[5] Georges Discry. Extending the menthor framework for parallel graph processing to distributed computing. Semester Project Report, Ecole Politéchnique Féderal de Lausanne. 5, 27

[6] GNU Lesser GPL. ScaLaLa – Scala Linear Algebra. https://github.com/scalala/Scalala. 6

[7] Philipp Haller and Heather Miller. Parallelizing machine learning- functionally: A framework and abstractions for parallel graph processing, 2011. 1, 5

[8] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In David E. Lightfoot and Clemens A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006. 5

[9] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009. 5

[10] Typesafe Inc. Akka project. http://akka.io/. 5

[11] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. June 25 2010. 5

[12] 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 Friedhelm Meyer auf der Heide and Michael A. Bender, editors, *SPAA*, page 48. ACM, 2009. 6

[13] Heather Miller. Scacs – Scala Cluster Service. https://github.com/heathermiller/scacs. 5

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. *SIGMOD '08*, pages ?–?, June 2008. 1

[15] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009. 5

[16] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. 2