An Empirical Study of Large-scale Graph Processing with Pregel

Nefeli Papapetrou Lampraki
An Empirical Study of Large-scale Graph Processing with Pregel

Master’s Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Nefeli Papapetrou Lampraki

7th June 2013
Author
Nefeli Papapetrou Lampraki

Title
An Empirical Study of Large-scale Graph Processing with Pregel

MSc presentation
June 11th, 2013

Graduation Committee
Prof. Dr. Ir. H. J. Sips (Chair) Faculty EEMCS, PDS, TU Delft
Dr. Ir. Ana Lucia Varbanescu (Supervisor) Faculty EEMCS, PDS, TU Delft
Dr. Ir. Alexandru Iosup Faculty EEMCS, PDS, TU Delft
Dr. Ir. A. J. van Genderen Faculty EEMCS, CE, TU Delft
Abstract

In this age of information, data gathering has become a new growing trend. Social networking sites, Internet banking, online communities, all gather and store data concerning their users preferences, interactions or activities. This data is strongly relational and can be represented in the form of graphs where a Vertex represents the subject of the information and Edges represent its interaction or connections with other entities. The purpose of using resources to store these large amounts of information is later processing that will lead to useful result derivation. Hence, we are dealing with graph processing. The large amount of information is leading to distributed graph processing for result calculation within a finite time.

Distributed large-scale graph processing, is a computational intensive task that entails many challenges such as distributed programming and knowledge of platform specific architectural characteristics. One approach that attempts to make the graph processing easy, platform independent and efficient, is Google’s Pregel model. Pregel’s computational model is based on an iterative program representation with two distinct entities (Master, Worker), that abstracts distribution related details behind a flexible, user friendly API.

In this project, using C++ and MPI we implement a Pregel clone, called MPI^2, and we evaluate its performance in the DAS4 cluster of TU Delft using three well studied graph processing algorithms (i.e., Page Rank, BFS, APSP). We conclude that the Pregel system successfully abstracts the graph processing task to a level where platform specific information is transparent. Furthermore, we observe that the system scales with the increasing number of computing nodes without introducing extra communication overhead. Finally, we notice that our system is sensitive to platform limitations like memory usage and we propose methods for its future development.
Preface

Everything started when I played my first computer game at the age of 15. Then suddenly I was finishing my MSc in Greece, programming in a Graphics Card (GPU) with its many architectural benefits and its many challenges and limitations. This spiked my curiosity about programming in unconventional architectures to the point where in TU Delft I was given the chance to program in a cluster of computers with an enormous computational power at my disposal. And all this to contribute in the optimization of the graph processing problem, that I believe to be one of the most challenging/interesting problems of the future. I could not imagine of a more suitable research path for me.

I would first like to thank my supervisor, Dr. Ana Lucia Varbanescu for her patience, understanding and guidance. She was my mentor during the last year, giving me inspiration and setting a great example for me. But most importantly she did not give up on me when frustration and work overload would overwhelm me. Also, I would like to thank all the members of the Graphitti project, who challenged me with their questions and helped me with their feedback in learning how to defend my work in front of a research group. Additionally, I would like to express my gratitude to Kees Verstoep and Munire van der Kruyk who were there to help me with immediate support on the many technical problems that I encountered both on the DAS4 and in the local desktop environment. Last but not least, I would like to thank my family and my friends, that are now spread all over the world, for their support that will always be priceless.

Nefeli Papapetrou Lampraki

Utrecht, The Netherlands
June 11th, 2013
To Angelos and Margarita, the greatest parents a child could hope for.
# Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Context</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement, Research Questions, and Contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 The Graphitti project</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Thesis Outline</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Background Knowledge</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Graphs</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Graph representation</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Graph Processing</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Distributed Computing</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Message Passing Interface (MPI)</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 Distributed ASCI Supercomputer (DAS-4)</td>
<td>14</td>
</tr>
<tr>
<td><strong>3 MPI(^2): Design &amp; Implementation</strong></td>
<td>17</td>
</tr>
<tr>
<td>3.1 Google’s Pregel Model</td>
<td>17</td>
</tr>
<tr>
<td>3.2 MPI(^2)</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Master - Worker communication schema</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2 Using MPI(^2)</td>
<td>25</td>
</tr>
<tr>
<td><strong>4 MPI(^2) Case studies</strong></td>
<td>27</td>
</tr>
<tr>
<td>4.1 Page Rank Implementation</td>
<td>27</td>
</tr>
<tr>
<td>4.1.1 Algorithm description</td>
<td>27</td>
</tr>
<tr>
<td>4.1.2 Porting Page Rank to Pregel</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Breadth First Search (BFS)</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1 Algorithm description</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2 Porting BFS to Pregel</td>
<td>36</td>
</tr>
<tr>
<td>4.3 All Pairs Shortest Paths (APSP)</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1 Algorithm description</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2 Porting APSP to Pregel</td>
<td>40</td>
</tr>
</tbody>
</table>
5 Experiments and Results

5.1 Datasets ......................................................... 43

5.2 Execution Time and Scalability .................................. 45
  5.2.1 Page Rank .................................................. 45
  5.2.2 BFS ......................................................... 48
  5.2.3 APSP ........................................................ 50

5.3 Time distribution .................................................. 52
  5.3.1 Page Rank .................................................. 52
  5.3.2 BFS ......................................................... 54

5.4 Discussion ......................................................... 57

6 Conclusions and Future Work ........................................ 59

6.1 Summary and Conclusions ........................................ 59

6.2 Future Work ....................................................... 61
## List of Figures

2.1 Large graph example ........................................ 8
2.2 Weakly Connected graph ................................. 9
2.3 Fully Connected graph .................................. 9
2.4 Directed graph ............................................. 9
2.5 Unconnected graph ........................................ 9
2.6 List graph .................................................. 10
2.7 Loop graph ................................................ 10
2.8 Star graph .................................................. 10
2.9 Sink graph ................................................ 10
3.1 MPI*2 organization ....................................... 19
3.2 Adjacency list ............................................. 21
3.3 Master - Worker Communication ....................... 24
4.1 BFS traversal tree of graph .............................. 35
5.1 Page Rank: Computation vs Workers number for small data sets . 46
5.2 Page Rank: Computation vs Workers number for large data sets . 47
5.3 BFS: Computation vs Workers number for small data sets ........ 48
5.4 BFS: Computation vs Workers number for large data sets ........ 49
5.5 APSP: Computation vs Workers number for Wiki-Vote dataset ... 51
5.6 Page Rank: Time distribution for small graphs ............. 53
5.7 Page Rank: Time distribution for large graphs .............. 54
5.8 BFS: Time distribution for large graphs .................... 55
5.9 BFS: Time distribution for small graphs ................. 56
Chapter 1

Introduction

1.1 Context

As a society we seem to have developed an increased desire for storing information [28]. Information concerning almost all our daily activities is stored: browsing preferences, bank transactions, social site connections and interactions, and much more. All these data collections, that can grow to fill exabytes of storage [28], contain items that are interconnected and, eventually, have the potential to reveal very interesting information about our social habits, interaction, and preferences. The relational nature of the data allows it to be represented in the form of graphs, where each vertex depicts an entry and each edge the relation between entries. Other sources of graph data are bioinformatics (e.g., genome analysis), text mining, traffic analysis in networks or road networks, and gaming.

Representing the data in the form of graphs, allows us to use existing graph processing techniques to analyze and mine this data. When dealing with small sized graphs, many techniques have been proposed that solve the processing problem using a single computing node (further referred to as node) in finite time. These sequential graph processing techniques are well comprehended, implemented, and tuned [16]. However, nowadays we are dealing with graphs that can scale up to billions of vertexes and edges, and this scale is much more challenging. The time that a single node would demand in order to complete any computation that requires, for example, a full graph traversal, is not acceptable. As a result, we need to shift from single node solutions to multiple node ones. In other words, to deal with the scale and complexity of the graph processing applications today, parallelism is mandatory.

When addressing graph processing with custom parallel solutions, one has to tackle two different classes of challenges: those related to parallel and distributed computing, and those related to the specifics of graphs datasets and algorithms. In
terms of computing, data and work partitioning, correct communication between
nodes, and results gathering are all challenges that need to be addressed. Furthermore, as the size of the graphs increases, so is the parallel processing system. For such large scale systems, hardware failures cannot be ignored and should be acknowledged and/or dealt with in the software. Finally, for every different computing infrastructure (eg: Cluster, Supercomputer, multi-/many-core processor) that will be used, we need to tune the node communication, the data distribution and decide on the fault tolerance technique.

On the other hand, given the intrinsic properties of graph processing applications datadriven computation, poor locality and irregular memory accesses, and low compute-to-communicate ratio [20], parallel processing of graphs can lead to unexpectedly poor performance [25]. Thus, different kinds of processing will require different approaches for efficient parallelization. For example, a page rank algorithm needs different handling than an information retrieval algorithm. And in many cases, for every different graph that we want to analyze, we need to take into account its specific characteristics and adapt the data partitioning accordingly.

To handle all these issues, one should have in depth knowledge of both graph processing and computer science, which limits potential beneficiaries to a small group of specialists. Thus, we recognize the need for a high-level graph processing system that will abstract away the specific details of both parallel graph processing and machine architecture and organization, and can easily be used by anyone familiar with the data analysis concept.

1.2 Problem Statement, Research Questions, and Contributions

As graph processing problems grow both in size and complexity, parallel execution of these workloads becomes mandatory. The diversity of hardware platforms brings along a lot of opportunities, as well as a large number of challenges. In reality, it is unlikely that common users of graph analytics will be able (or interested) in parallelizing their analysis, while avoiding this step will lead to unfeasible execution time.

Problem statement Parallelizing large scale graph processing is difficult, because both the complexity of the hardware and the complexity of the analysis are increasing. Therefore, the complex nature of parallelizing graph processing algorithms limits the complexity, the scale, and eventually the number of datasets that can be currently processed. While high-level graph processing systems are proposed to address this problem, they are not properly analyzed and understood, and therefore not sufficiently used.
**Research Questions** To address these limitations, we propose an empirical investigation of the feasibility and performance of a high-level, distributed graph processing system.

Specifically, we aim to answer the following research questions:

1. How difficult is it to implement a high-level graph processing system on a real cluster, using message passing?

2. What is the performance such a system can achieve on a real cluster? How does the performance differ for different algorithms and different datasets?

3. How effective (in terms of effort-to-performance ratio) is such a system for implementing graph processing algorithms in parallel?

**Approach** In this thesis, our investigation spans from the design to the prototype, and from usability to performance, all in the context of a real, large cluster of homogeneous nodes. Called MPI$^2$, our solution is based on Pregel, a model proposed by Google in 2010 [21], which provides the user with a high-level front-end for designing and implementing the graph processing task, and supports this front-end with a complex parallel system that maps (easily, in theory) on parallel hardware. Specifically, in this work, we discuss the design and implementation of a “clone” of the Pregel backend using message passing over a cluster of homogeneous nodes. We further study the feasibility and usability of the model for different graph processing algorithms. Finally, we conduct an empirical study to evaluate MPI$^2$’s performance in a real cluster. Our results show, among other things, that the model (1) does allow for simple implementation of graph processing algorithms, (2) scales well on a cluster of homogeneous, powerful nodes, and (3) it is vulnerable, for very large graphs, to performance drops due to message congestion.

While the idea of using high-level graph processing systems to address both productivity and performance is not new (see Neo4J [13], Stratosphere [9], Giraph [1], and Stanford’s GPS [29]), the combined study of design and prototyping with performance and usability, as well as the range of experiments we perform on real hardware, for multiple algorithms and multiple data-sets provides new insights into the capabilities and limitations of such a system.

**Contributions** The main contributions of this work are the following:

- We provide insights into the design and implementation of MPI$^2$, a first Pregel implementation using MPI in C++. We further discuss the challenges that the deployment of MPI$^2$ brings in a real cluster.

- We implement three different algorithms in Pregel, and we discuss the usability of the model.
• We conduct an empirical performance study using these three algorithms, and we show how our MPI^2 performs on a large number of datasets.

• We trace back our performance and usability findings from MPI^2 to the original Pregel model, and discuss the advantages and limitations of the model itself.

1.3 The Graphitti project

This work is a part of the Graphitti project, which proposes a system for parallelization and distribution of large-scale graph processing workloads.

Our system is built to address the increased complexity and diversity of both data sets and analysis procedures in graph processing applications, while being able to cope with a large variety of target architectures. Specifically, we propose a modular solution that will tackle both the productivity and performance concerns.

Graph processing is becoming both complex and diverse: graphs grow in size and in complexity (size- and information-wise), and analysis turns into difficult data-mining. While parallel processing for graph analysis is already mandatory for any results, the much more difficult “best-fit” problem arises: matching the right platform (infrastructure) with the right graph processing workload such that performance is good, resources are not wasted, and users are able to cope with the increased complexity. To address these challenges, we provide a way to design portable applications using a single, machine-independent, application-centric API, and a set of machine-specific, optimized backends that take care of the efficient execution of the applications on the target architecture.

The features of our system are as follows:

• A standard data format, that will allow sharing data-sets and allow data modeling for optimized distribution. This further enables our single API approach, which in turn addresses productivity concerns when changing target architectures.

• A single, common API, allowing application portability for different types of architectures. Together with the common data format, this API enables ease-of-use of the same application for different scales of machines and/or data-sets.

• Multiple back-ends, which are the core of our system and address several classes of architectures: multi- and many-core processors, accelerators, normal desktops, clusters, and IaaS clouds. Specific optimizations and/or approaches have been taken to insure good performance at the back-end level. Each backend takes care of data-distribution, parallel execution, synchronization, and result gathering for a class of target platforms.
The validation of our approach happens in two stages: (1) run the same work-
load on different platforms and be able to compare the results (ongoing work), and
(2) solve the “best-fit” problem using performance prediction based on data- and
application-models (future work). Once available, the performance predictor will
be integrated in Graphitti in a performance feedback loop, for matching of work-
loads and platforms.

1.4 Thesis Outline

This thesis intends to study the impact of the Pregel model in Large-scale Graph
Processing, and thus knowledge of graph processing and distributed programming
may be required. In Chapter 2 we provide some background knowledge on graph
representation and processing. We explain the concept of distributed computing
and why it is required in large-scale graph processing, and we give an overview
of the Message Passing Interface and of the DAS-4 cluster. In Chapter 3 we de-
scribe Google’s Pregel model and we present MPI\(^2\), our approach to the graph-
processing problem. Then in Chapter 4 we present the Datasets that were used
in our experiments and the graph analysis that we performed with MPI\(^2\) using
known graph processing algorithms. In Chapter 5 we present the performance of
our MPI\(^2\) applications and finally in Chapter 6 we discuss our conclusions and we
propose future directions of our study.
Chapter 2

Background Knowledge

2.1 Graphs

The term graph is familiar from mathematics where it represents a collection of discrete elements (vertexes) which are connected with links (edges) [14]. By extension, in computer science, the same term refers to an abstract data structure that is used to store data items and the relations between them.

Most of the data elements that are stored and collected are relational. The term relational data, refers to a number of entities, that represent in their collective, a group of information. This data, was previously stored in databases. Every database entry contained information about its own fields and it was possible for an entry to contain a shared key, that was pointing to another entry.

Now, instead of databases, entries and shared keys, we have graphs, vertexes and edges. A graph vertex is representing a standalone part of information that usually is characterized by a unique ID and possibly contains other values. Graph edges represent possible connections between vertexes. For example if a graph represents a country’s roadmap, then each city would be represented as a graph vertex, and the roads between cities that have a direct connection with each other would be the graph edges.

An example of a real graph is presented in Figure 2.1. It represents the Linked-In network of a user (Nefeli, in this case): the vertexes represent contacts in the Linked-In list of the user, and the edges represent interconnections between all these users.
Graphs can be divided into several types, and can be categorized by several characteristics, most of which refer to the nature of the graphs edges. A graph is defined as directed when each edge is uniquely described by its source and its destination vertex. In this case and edge initiating from vertex A and ending in vertex B [A-B] is different from edge [B-A]. For example, if you think of a graph that represents the distribution of votes for elections inside a community, each vertex can represent a citizen, and an edge [A-B] would represent the vote from citizen A to citizen B. Of course, edge [B-A], if existing, refers to a separate vote. Instead, in an undirected graph, if there is an edge from A to B, there will also be an edge from B to A. Taking the roadmap example from above, a road from city A to B can be used for traffic from both A to B and B to A (we assume here there are no one-way roads).

Both directed and undirected graphs can be weighted - i.e., each edge can contain additional information for the connections between vertexes. For example, in the road network mentioned above, the distance between the connected cities is a typical edge weight.

Another graph characteristic is connectivity. A graph is connected when all the vertexes in the graph are reachable from all other vertexes. If any vertex in the graph is directly reachable from any other vertex in the graph, the graph is called fully connected.

There is the possibility to have a graph that contains vertexes that share no edges with other vertexes. In this case the graph is considered unconnected. In the former example of the election graph, the citizens that did not place a vote and were not voted will be isolated, and the community graph would be unconnected since there
would be some vertexes with no connecting edges. Note that for directed graphs, we define two more types of connectivity: weak connectivity - there is an undirected path (i.e., assume all edges are bidirectional) between any two vertexes in the graph, and strong connectivity - there is a directed path between any two vertexes in the graph.

Some examples of graphs with these characteristics can be seen in figures 2.2 - 2.5.

Figure 2.2: Weakly Connected graph

Figure 2.3: Fully Connected graph

Figure 2.4: Directed graph

Figure 2.5: Unconnected graph

Density is another graph property of interest. The density of a graph is defined as a ratio between the number of edges in the graph and the number of possible edges in the graph (i.e., the number of edges in the fully connected graph with the same number of edges). If this number is close to 1 - i.e., if the vertexes of a graph are highly connected with each other, then the graph is dense. If the number is closer to 0, i.e., the vertexes are less connected to each other, the graph is considered sparse.

Finally, depending on the number and types of connections between vertexes, certain graph topologies can be defined. Figures 2.6 - 2.9 present four extreme cases of graphs that represent a certain topology. The Star graph, has only one source vertex that is pointing to all others. The Sink graph is the exact opposite, having one destination vertex that all other vertexes are pointing to. The List and the Loop graphs are representing a sequence of vertexes that are always pointing to the next vertex.
2.1.1 Graph representation

When it comes to storing a graph $G=(V,E)$, one can imagine multiple solutions. Traditionally, the most common solutions are separated in two different classes: adjacency matrices or adjacency lists. An adjacency matrix is a two-dimensional array, $|V| \times |V|$, and each of its cells represents the status of the direct connection of a pair of vertexes - i.e., existing or not, or a weight. Note that for dense graphs, this is an efficient solution to quickly search for the connection between any two nodes. However, for sparse graphs, the adjacency matrix is too sparse to be efficient.

An alternative solution, typically used for sparse graphs, are adjacency lists, which present the information with the vertexes as main entities. The graph is stored as a collection of vertexes, where each vertex contains a list of neighbours or adjacent edges. In this case, the size of the data structure will be directly dependent on both the number of edges and vertexes, making sparse graph storage more efficient. Furthermore, the vertex information is clustered, making algorithms that target to collect graph information (average connectivity, strongly connected components, neighborhoods) more efficient. However, this representation is difficult to maintain when edges and vertexes are added and removed frequently from the graph.

With the growth of graphs and data collections, the use edge lists has emerged as a new way to represent graphs. In this case, the graph is a collection of the edges present in the graph, together with any values that could be associated with each vertex and edge. The dynamic removal or addition of an edge is very efficient. Similarly, adding a new node is implemented efficiently by adding its edges to the already existing vertexes. However, in order to gather information about the graph,
several traversals of the full edge list can be required. Furthermore, removing a
node and/or adding isolated nodes to a graph require special attention.

We note that choosing a representation depends on the application, the type of
graph, and the available resources. In this work, we are using the edge based graph
representation for external storage, and a vertex-based adjacency-list for the in-
ternal (in-memory) representation. The reasons for this choices are discussed in
Chapters 3 and 4.

2.1.2 Graph Processing

As we are living in a digital world, all the actions that we perform are somehow
electronically recorded and later analyzed in order to build our electronic profile
based on our preferences and choices. Thus, a huge amount of information is the
result of actions such as phone calls, Internet browsing, social networking, Inter-
et shopping or even bank card usage and other actions that we perform everyday.
There is a lot of interest for all this data, from the government, third-parties[18],
and research institutions. Therefore, everything is stored regardless the enormous
size. In fact, it is estimated that the amount of information stored into the world’s
databases, doubles every 20 months[34]. All these rapidly growing digital data
collections contain a lot of important data items and their relations and are suitable
to be represented in the form of graphs.

The main reason to save information is to be able to access it and process it
when needed. In computer science this is called Data Mining and it is described as
the process of discovering patterns from large data sets in order to extract know-
ledge in a human-understandable structure [2]. Some use-cases for data mining
are: creating summaries, extracting subsets with specific characteristics, creating
statistics and analyzing emerging patterns. Specifically for graphs, there is a lot
of interesting analysis to be made on the nodes and their relations, with different
interpretations depending on the provenance of the data (social networks, phone
calls, airline flight schedule).

The task of processing graph information is not only important, but also very
challenging. Because of the constant data storage, the graphs to be processed are
constantly increasing in size, level of detail and complexity of links. Thus, we are
not dealing with simple graphs, but with very large complex graphs that can hardly
be represented in the traditional node-edge fashion[12]. Therefore, it is expected
that the processing of these graphs requires great amounts of computing power and
it needs to use the system resources in the best possible way. Therefore, a very
important research challenge is how to process such large graphs in reasonable
amount of time with limited resources.
Graph problems have some inherent characteristics that make them difficult to be optimized with current computational problem-solving approaches [20]. First of all, the complexity of performing an algorithmic task on a certain graph, does not depend only on the algorithm complexity, but also on the structure of the graph. The same number of nodes if being interconnected in a certain way, can cause different performance drops in different algorithms, and speed-up others. Moreover, if the structure is irregular, exploiting parallelism is troublesome since the graph cannot easily be partitioned and distributed.

Furthermore, in graph processing data access overrules locality [20]. Usual graph processing tasks are traversals that collect or propagate information in a certain way. While running, the traversal algorithm will need to continuously fetch the data in memory or store them there throughout the computation. The computation that will take place, will be relatively short comparing to the time of all the needed accesses. Finally, it is not always possible to exploit data locality for optimising memory accesses, since the data that graphs represent contain interconnections that are distributed among the dataset.

2.2 Distributed Computing

Since the early 1980s the term Distributed Computing has become a separate and quite popular branch in Computer Science. The term refers to multiple compute nodes that are physically distributed (in a small or even large geographical area), yet collaborate to solve the same problem.

In terms of architectures, distributed systems are typically built from autonomous computational entities, each with each own local memory, and interconnected by an external network. These systems have no common memory. In the case the problem to be solved needs these nodes to communicate with each other, they will do so by message passing protocols. This communication, which happens whenever non-local data is needed, is based on a messaging protocol between data requester and data owner. This can be explicitly implemented by the programmer, when using programming models like MPI [4], or implicitly solved by the programming model, when using virtual global address spaces (like in UPC [10].

The structure and topology of distributed systems can vary a lot. The system may be homogeneous, containing multiple nodes of the same kind and the same interconnections between them. It can also consist of different kinds of nodes and network links, case in which it would be called heterogeneous. Depending on the size of the system, as well as on its access policies, types of applications it executed, or eventual costs, distributed systems have evolved from clusters (the smallest, lease convenient, yet most controllable distributed systems) to grids, clouds,
and federated clouds. For the purpose of this work, we aim to design a system able to function efficiently for clusters.

We note that there are several supercomputers that also belong to the class of distributed computers, despite being hosted in a room or even in a single rack. Most of them are programmable using message passing - MPI or vendor-specific protocols. Therefore, our solution should be portable, at least at the functionality level, for execution on machines such as IBM BlueGene [17].

There are several reasons for which distributed computing may be preferable over single node computations. First, it may be required by the application: if data needs to be produced in one location and be available for use in another. Example of such scenarios are distributed information processing systems such as banking systems, distributed databases, or even massively multiplayer online games.

Second, it is beneficial in terms of performance: if a compute-intensive task can be split in smaller concurrent tasks, then the workload can be distributed among several nodes, which, by computing in parallel, will solve the problem faster. In most cases, it is more cost efficient to obtain the desired performance using several low-end computers, rather than a single high-end supercomputer. Recently, due the BigData trend, the same approach is being taken for data-intensive applications as well: it might be beneficial to split the data in the memory of multiple nodes in a distributed system than to store it in the disk drive of a single one. In general, the performance gain in such cases is due much less to the performance of the computation, and much more to the improved I/O access.

Finally, distributed computing is more fault tolerant than single node systems. There is no single point of failure, and several system recovery mechanisms may be easily implemented to guarantee system recovery to a safe state after certain types of failures occur. Most these fault-tolerance mechanisms are based on Checkpointing and Replication[11].

In this work, we are using the DAS-4 system (Section 2.2.2), a multi-cluster system dedicated to education and research in The Netherlands.

2.2.1 Message Passing Interface (MPI)

Programming distributed systems is typically a combination of data partitioning, data distribution, intra-node computation and inter-node communication. When implementing all this functionality, programmers typically use dedicated programming models.

One of the most popular programming models for distributed systems - most probably, still the most popular one - is MPI. It was proposed in 1994 with the purpose of becoming the standard for message passing interface, and thus to be
used in a wide variety of distributed systems and networks [5]. MPI is a message-passing library interface specification, and is based on the consensus of over 40 organizations that form the MPI forum [15]. MPI is a standardized and portable message-passing system designed to function on a wide variety of parallel computers.

The quick adoption of MPI is due to its ease of use. MPI abstracts the message exchange to a higher level, allowing users that do not possess low level knowledge of a platform’s communication environment, to use message exchange. Another option for communication between distributed processes is via a transport protocol (TCP, UDP) which is an unnecessary complicated process, requires specific knowledge and handling, and is not preferred for mainstream software development. Furthermore, MPI offers standardization. It provides clearly defined and easily implemented routines. MPI specifications are following the language bindings of C, C++, Fortran-77 and Fortran-95. Moreover, MPI is offering portability. Programs that are developed in one platform but use MPI for message exchange, can be executed in other platforms that support MPI, with no need of modifications. Also MPI offers over 115 routines and that makes it quite flexible and available for programmers use.

MPI can virtually run on any high-performance computing (HPC) platform and thus code written with MPI will be very portable among parallel and distributed systems - although originally designed for distributed memory, nowadays it has been ported for most distributed and shared memory architectures. These targets include IBM BlueGene, CRAY T3E. Several MPI implementations are available, such as Open MPI, MPICH, or MVAPICH.

As it is probably obvious by now, we chose MPI as our programming model. Besides MPI’s popularity, there were two main reasons behind this choice: (1) Pregel is essentially a message passing framework, thus implementing it in a message passing environment seems the most feasible option, and (2) MPI is both portable enough and explicit in its parallelism, which allows us to test the same solution on different (non-cluster) architectures, thus having a starting point for alternative implementations.

2.2.2 Distributed ASCI Supercomputer (DAS-4)

DAS-4 (The Distributed ASCI Supercomputer 4)[8] is a distributed system designed by the Advanced School for Computing and Imaging (ASCI), that consists of six clusters connected by a very efficient Wide Area Network (WAN). In the realization and running of DAS-4 several universities including the Technical University of Delft are directly involved. DAS-4 started operation on 2012. Its predecessors (DAS 1, 2 and 3) date back to 1997 and were used widely for research.
on communication software, parallel languages and programming systems, schedulers, parallel applications, and distributed applications.

DAS-4 consists of 200 dual-qual-core computer nodes, within 6 clusters located in 5 different physical locations. DAS-4 contains also HPC accelerators, currently being programmable Graphic Processing Units (GPUs). Each cluster contains a head node and many compute nodes. The head nodes are intended for program development (editing, compiling) and the program execution is always done in the compute nodes. The job scheduling is handled via the DAS-4 cluster scheduling system Grid Engine. [7]

All the nodes within the Supercomputer, are running instances of the CentOS Linux operating system and a synchronous shell-like interface, is enabling the user to interact with the head nodes. Furthermore, additional software such as optimizing compilers, visual packages, MPI implementations (OnenMPI, MPICH) and performance analysis tools, is installed in the nodes of DAS-4. The user can develop programs in a variety of languages some of which are OpenCL, Cuda, C++, C.

The connectivity in DAS-4 is highly innovative and it consists of an internal and an external network. Within each cluster, the nodes are connected locally with both an internal 1 Gbit/s Ethernet and a QDR InfiniBand network. The Ethernet link is used mainly for management issues, and the QDR which offers low latency and high throughput, is used by applications and is accessed via MPI, or via the regular IP layer. To arrange external connectivity, the head nodes act as routers to connect to the backbone network of each university. Additionally, compute nodes can communicate over a very efficient Wide-Area network called StarPlane, which is based on dedicated 10 Gbit/s light paths provided by SURFnet.

The experimental part of this thesis, was completed with the use of DAS-4. The majority of the experiments were done in the nodes of the Cluster located in TU Delft(fs3.das4.tudelft.nl). In cases of high demand, additional experiments were run in the Cluster of Leiden (fs1.das4.liacs.nl) and Amsterdam (fs0.das4.cs.vu.nl).
Chapter 3

MPI^2: Design & Implementation

3.1 Google’s Pregel Model

Pregel\cite{21} is a graph processing system created by Google in 2010, to address distributed processing of large scale graphs. The Pregel team gathered all the problems that were previously known in the field of large scale graph processing, and attempted to tackle most of them. Pregel is designed to be flexible enough to express a broad set of algorithms, as well as scalable and fault tolerant.

Pregel has a high-level organisation that is inspired by Valliant’s Bulk Synchronous Parallel model\cite{31} and abstracts the graph processing task. The processing of the graph is divided iterations, called Supersteps. In every Superstep the user defined function Compute() is called, if possible in parallel, for every graph Vertex that is part of the computation. Within the execution of Compute() previous messages regarding the corresponding Vertex can be collected, its value can change according to the given algorithm and new messages can be sent along its outgoing edges. To achieve synchronicity, the messages that are sent in the Superstep N-1, are buffered and available to be collected only after Superstep N is instructed.

Pregel is designed for Google cluster architecture, and thus it is intended to run with several "compute" nodes. The computational model is consisted of a node assuming the role of Master, and the rest of the nodes assuming the role of the Workers. The role of the Master node is mainly to manage and coordinate the activities of the Workers, yet it can also collect statistics of the computation and perform any auxiliary tasks that may be needed. Each Worker, is assigned a part of the initial graph, and will run the Compute() method on the Vertexes contained in it, when instructed by the Master. All Workers are able to communicate directly with each other, through the communication interface that is offered in Googles cluster, and they can be always discovered through usage of the systems name ser-

\footnote{The name MPI^2 is an abbreviation of the full name of our system: My Pregel Implementation in MPI.}
There are some basic stages that a program follows when executed in Google's Pregel. Initially, from the many program copies that begin execution, one is assuming the role of the Master. The Master will listen to registration messages from the Workers who will discover its location from the clusters name service. Then, the Master will partition the graph according to the user's configuration, trying to exploit parallelism in the best possible way. There may be more partitions than the number of available workers. Next, the Master will split the graph in sets of records and will assign portions of the input to the registered workers.

After all workers have finished loading their tasks, the Master will instruct the Workers to perform a Superstep. The workers will loop through the active Vertices in their set, processing the messages that were sent/received in the previous Superstep, executing Compute(), and sending any messages needed for the next Superstep. When all the Workers are through with the computation, they will report back to the Master who will instruct the next Superstep, or end the computation if all the vertices are inactive.

A vertex that has no more work to do will deactivate itself - i.e., will vote to halt, and become inactive. An inactive vertex can only become active again if it receives a message. The computation continues as long as at least one worker is still active. If that is no longer the case, all nodes have finished their work and the application can be closed. In other words, when all workers vote to halt, and there are no more in-flight messages, the application has terminated.

Besides the main computation model, Pregel defines mechanisms to optimize the runtime system. For example, to reduce the communication overhead that can occur, the system can combine several messages intended for a certain vertex in the same Superstep. In order to do so, the user must define a Combiner that will work as an independent entity in the system, supervise the messages being sent, and combining them in batches of messages for different nodes. Furthermore, for gathering statistics of the global communication and monitoring the data in each Superstep, every vertex can provide a value to be aggregated. These values will be combined, using a reduction operator like min, max, or sum. The module responsible for these actions in Pregel is called the aggregator.

Finally, for fault tolerance, Pregel is using checkpointing. When a Worker node fails (i.e., does not respond in one or more Supersteps), the Master will rollback to a safe state, recorded at the most recent checkpoint, and read the graph state at that time. Using this safe state as a reference, it will reassign the partitions of the failing node to the remaining active workers. Confined recovery with message logging is also proposed, to save computer resources in case of failure and recomputing only the lost partitions.
3.2 MPI$^2$

Inspired by Google’s Pregel description, in this MSc thesis, we created a Pregel-like implementation in C++, using MPI. We call our system My Pregel Implementation, or MPI$^2$. MPI$^2$ shares many similarities with Google’s Pregel. The entities of the Master, the Worker, the Supersteps and the Compute() method, are functioning as described in chapter 3.1. However, fault tolerance is not yet implemented and the graph partitioning is handled with small deviations from the original paper.

![Figure 3.1: MPI$^2$ organization. Note that all the entities in this figure are able to send and receive messages via MPI, however several arrows are not depicted for aesthetic purposes.](image)

The execution of MPI$^2$ is done in the environment of DAS-4, using MPI. In order to start the code, the user has to execute an MPIRUN command where the number of desired compute nodes and the input file name need to be specified. The input file should contain node names with positive numeric IDs, in the range of 0 to 2,147,483,647 presented in the following edge based format:

```
{vertex_name \t edge_name \n}
```

To aid the user, a graph validator and converter tool has been implemented. In the case that the graph files need to be checked for compatibility the user can use this tool providing the input file name and enabling the tool’s first mode. The tool will go through the graph and verify that this format is indeed supported by MPI$^2$. Additionally, this tool can convert certain Vertex based representations to edge based. To do so, the user must provide to the tool an input graph in the following
format, and enable the tool’s second mode. Furthermore, the graph converter is
designed in a way that allows simple modifications that will result in the support
other graph representations.

\{vertex_name \t @ inEdge1 inEdge2 # outEdge1 outEdge2 \n \}

After MPI\(^2\) starts, each instance of the program will run in different nodes of
the cluster, with its local variables. The instance with process ID 1, will act as the
Master, and the rest will assume the role of the workers.

First the Master will listen for registration messages from the workers. When
running in MPI, the Master can take advantage of the environments name service
to get the IDs of all the processes (Workers) that started executing. However, in
this implementation we chose message based registration.

The reason we chose this approach is threefold. First, in this manner, the pro-
gram can be easily ported with a few modifications in another cluster that does not
use MPI and offers its own message parsing service. Second, in this way, we can
support dynamic change in the Workers formation. In case a worker malfunction is
detected, the system can run the registration process again to detect the left active
workers. Finally, in this manner, we are not deviating too much from the original
Pregel instruction.

Afterwards, the Master will read the input Graph file, and will translate it to an
Adjacency list, where all the nodes of the graph are depicted as Vertexes and all
the connections between nodes are depicted as edges, as you can see in figure 3.2
on page 21.

In this manner, the program can function with both Vertex oriented and Edge
oriented graph representations. The current version of MPI\(^2\) contains an input
function that is transforming Edge oriented Input files to an Adjacency list. In case
the user, does not wish to use this graph representation and does not wish to trans-
form the input file with the prementioned tool, he can adapt that function in an
appropriate manner in order to read any text based graph representation.
While storing the input information, the Master will also split the graph in partitions\textsuperscript{1} depending on the number of registered workers. In this implementation of MPI\textsuperscript{2}, the number of partitions will be always equal to the number of registered workers. The partitioning is done using a simple hash(NodeID) mod N function, where N is the number of partitions.

Next, all the partitions are set to the corresponding Workers, via MPI. This send is not necessary since in DAS-4 all the nodes that are initiated by an MPIRUN, have access to a shared disk space. However, we chose to implement it, to make our program portable and able to run in other clusters as well.

After the Workers receive their partitions, and map them in their local Adjacency list format, they will wait for the Master to instruct to them, the first Superstep. During every Superstep, the executions steps are identical. First the Workers will try to receive any messages that was send in the previous Superstep. For every message they will update their local information, and after having handled all messages, they will execute the Compute () function.

The Compute () function is algorithm specific, and the user will need to overwrite it, in case he wishes to run an algorithm that has not yet been implemented in MPI\textsuperscript{2}. During the Compute () function the graph values will update accordingly and messages will be sent via MPI to be received from the recipients in the beginning of the next superstep.

When each Worker finishes the execution of each Superstep, will send a message to the Master, indicating also the number of messages that he send to each other worker. The Master collects all the information and only after all Workers

\textsuperscript{1}The term does not bare any similarity with the term ”partitions” that was used in section 3.1 but refers to the term ”set of records” that was used in the same section.
have completed the current Superstep, he will instruct the next one. Along with the message that instructs the Superstep, the Master will also inform each Workers, how many messages were sent for them to collect. In this way, possible message loss can be detected.

The Superstep loop will continue until the end condition of the algorithm is reached. This differs between algorithms. In some cases the computation will stop after a predetermined number of Supersteps, in others when the computed values reach convergence, and it is also possible that the Vertexes will vote to halt as described in section 3.1.

After the computation ends, the Master will receive the graph values from all the Workers and will confirm their termination. Finally, the Master will print the Vertexes with all their values, and will terminate as well.

### 3.2.1 Master - Worker communication schema

In figure 3.3 on page 24, you can see the basic message exchange between the Workers and the Master node. In this section we explain in detail the message exchange between the Master and the Workers.

As we can see, the Master starts with waiting for all the Workers to register. The registration lasts for 2 seconds or, until we received a registration message from all the other nodes. An advantage of using MPI for the communication is, that we can extract the total number of the processes that started the computation for the same program instance, and thus we can calculate the number of the remaining nodes.

Next, the Master will read the input file and store it in the local memory as an Adjacency list. Then, the Master will send a series of messages to each Worker that was successfully registered, with information about the Vertexes number in the input Graph, the length of the partition that is assigned to it, and the Adjacency list that contains all the Vertexes and edges of the partition. Furthermore, the Master shares the information about all the registered Workers, so the message exchange between Workers can become possible. In some cases, when the input Graph is too small, there will exist some Workers that have no partition assigned to. The IDs of these Workers are known to the Master that does the partition assignment and also to the other Workers from the information that is provided to them by the Master.

Another information that is sent by the Master, is the existing partitions number. In this implementation of MPI², when a Vertex needs to communicate with another Vertex, the ID of the Worker than contains the target Vertex, is discovered with use of a common hash() function, that takes as a parameter the total number of the partitions as indicated by the user. This behavior is time efficient since there
is no need for a creation of a hashing table that would have to be shared with all the Workers.

Then, the superstep loop follows. When the Master enters this loop, it will send a message that commands all Workers to start executing the superstep. Each Worker may receive this message at a different time and thus start the computation earlier than another Worker. However, there will be no Worker starting a new superstep before everyone has completed the previous superstep.

When a Worker receives a command to start the next Superstep, it will first collect all the messages that was send for it in the previous Superstep. Then, it will execute the Compute() function which is Algorithm specific, and it will remember the number of messages that will be sent to other Workers. When the Compute() function completes, the Worker will inform the Master of all the sent messages and the Master will include this number in the next Superstep command. In this manner, each Worker is aware of the number of the messages that will need to be collected in the beginning of the next Superstep and thus, possible message loss can be detected. The end of the Superstep loop is depending on the algorithm ans the user configuration. In can end either after a certain number of repetitions has been reached, or if all the Workers vote to halt.

The final step after the end of the Superstep loop, is to collect the results. In the current MPI^2 implementation, the Workers send to the Master their Adjacency lists and the Master is responsible of collecting and printing the results to the user. Depending on the input Algorithm, this may not be necessary. The user is encouraged to change the result functions accordingly, to avoid any extra information sending.
Figure 3.3: Master - Worker Communication

24
3.2.2 Using MPI\textsuperscript{2}

To use MPI\textsuperscript{2} to run your own algorithm in a graph dataset, the programmer needs to have knowledge about [certain things] and follow some recommended steps.

- **Know your algorithm** As simple as it sounds, having a clear knowledge of the algorithms functionality, is essential in order to use MPI\textsuperscript{2}. Thus, the first step that the user needs to take, is to divide the algorithm in clear Supersteps. This means that the user needs to detect a certain computation that can be repeated in an iterative way through the Vertexes of the input graph. One of the advantages of the Pregel model lies in the various Supersteps that divide the calculation into chunks of computation that can be done parallely by the Workers to the partitions of the Graph, while being coordinated by the Master.

- **Initial setup** The next step that the user needs to take is write and Init function that sets the initial values for all the Graph Vertexes. This may not be needed in some algorithms as the values may be overwritten in the first superstep, but in some other algorithms may be essential.

- **Program Master and Worker behaviors**
  
  - **Master Behavior**
    
    The Masters behavior indicates what that the node acting as a Master, will do in every superstep. According to the Pregel model [21] the Masters main responsibility is Worker coordination. This entails monitoring the Worker behavior by placing certain barriers that force the Workers to proceed through the Supersteps in a predictable manner. Moreover, the Master can keep statistics of the input graph, information about the messages transferred during the Supersteps, timing information, and remember aggregator values.

    The functionality of the Master follows those guidelines with every input algorithm and thus the main part of the code is intended to be present in all implementations. However, in order to take advantage of the full potentials of MPI\textsuperscript{2}, the user needs to write some parts of the Master behavior that may be algorithm specific.

    In order to make this possible, the user needs to have a clear understanding of the functionality of the Master node and its expected behavior in the certain algorithm. This way he will be able to program the Master function according to the algorithms needs.

  
  - **Worker behavior**
The Workers behavior, indicates the algorithm specific actions that each node acting as a Worker, should perform in every superstep. According to the Pregel paper [21] the Workers need to keep information about the Vertexes of the input graph and perform the Compute() function in every superstep. The Compute() function iterates through all the incoming messages, updates the Vertexes values, and checks wherever a Vote to halt the computation is in order.

The main part of the functionality of the Compute() function is algorithm specific. In different algorithms the way the messages are decoded can be different, the Vertexes in which messages should be send in the next superstep is a subject to change, and most importantly the condition when a Vertex should vote to halt can vary. Thus the user needs to program the above function accordingly. For example, in our Page Rank implementation, the Compute function does no check and terminates after 30 loops, and in the BFS it checks whether all the graph nodes have been coloured.

- **Message schema** Next, the user will need to draw a clear message schema that describes the message exchange that will take place between the Master node and the Worker nodes. In this way, the user will have a clear understanding of the message exchanging that is expected of the program. Then he will be able to make a comparison with the standard message exchanging as provided with MPI² and evaluate if further messages are needed, or existing messages are obsolete.

During our case studies, we concluded that following the above steps was really useful, gave our programs a consistent structure, and maximized the amount of code reuse between implementations. Code reuse is expected between Pregel implementations since the user should only be required to modify MPI² only in the algorithm specific parts, or when he wishes to add or modify a functionality.
Chapter 4

MPI’2 Case studies

4.1 Page Rank Implementation

The first algorithm that we implemented in MPI’2 is the Page Rank algorithm [23]. This algorithm dates back to 1996, and is designed to assign weights to any collection of entities measuring the importance of each entity in the group. The Page Rank algorithm gained popularity when on 1998 Google obtained the licence rights to use it in their search engine. Page Rank ranking is a solution to the growing need of having a global ”importance” ranking among web pages with no context similarities.

4.1.1 Algorithm description

The Page Rank algorithm is discussed in the original Pregel paper as well[21]. Implementing this algorithm gave us the opportunity to closely follow the instructions of Google and thus make sure that our model deviates as little as possible from the original. In this way we can make sure that our research is providing results that closely resemble the Pregel systems results and thus our analysis can be used to provide insights into this popular model.

Another advantage of implementing Page Rank in MPI’2, concerns the algorithm itself. We believe that Page Rank is a useful and widely used algorithm that is implemented and analysed several times, in many systems and languages. Hence, by implementing it in MPI’2, we can make use of other implementations to compare and have a better understanding of our performance results.

The Page Rank algorithm has the functionality of a page ranking algorithm and it represents the likelihood that a person who randomly navigates on web links, will arrive at any particular page[23]. A page with a high Page Rank score is more likely to be visited. In order to calculate the Page Rank of a certain page A, we need information from all the web pages that point to A. The links to other pages,
count as a votes of support. This approach is based on the assumption that pages which are highly linked, have more importance than pages with a few links. The same rule applies in scientific articles. Articles that are most cited are more important than articles which have few citations. Furthermore, a citation from an article that is highly rated is more valuable.

An intuitive definition of page rank is:

*A page has a high rank if the sum of the ranks of its backlinks is high[23].*

The Page Rank score is calculated using a simple iterative algorithm. Equation [4.1] is evaluated iteratively until the values reach convergence. With convergence we mean the tendency of the Page Rank values to stabilize. Due to the way Page Rank is calculated, the values will keep on changing and in finite amount of time this change will become negligible.

The Page Rank algorithm executes the following formula, several times until the values reach convergence.

$$PR(A) = (1 - d) + d \times \left[ \frac{PR(N1)}{S(N1)} + \ldots + \frac{PR(Ny)}{S(Ny)} \right]$$  \hspace{1cm} (4.1)

In equation [4.1] $PR(A)$ is referring to the Page Rank score of page $A$, $y$ is the number of pages that link to page $A$, $S(x)$ is the total number of outgoing links initiating from page $x$, and $d$ is a damping factor that can range from 0 to 1. This damping factor is preventing pages to have too much influence in the estimation of the final page rank. If $d$ is too high then it will take a lot of time for the PR score to settle. On the other hand, if $d$ is too low we will have a repeated overshoot above and below average, causing the Page Rank numbers to never settle.

Another thing that we notice in equation [4.1] is the fraction $\frac{PR(Ny)}{S(Ny)}$. This fraction shows that Page $Ny$ will divide its "support" equally among all outgoing links. Thus, a highly ranked page, with an equally high number of outgoing links, will not contribute much in its neighbors Page Rank score. On the other hand a page that is linked to many pages with high Page Rank, will receive a high rank itself accumulating the considerably high votes of its neighbors.

As we discussed, to calculate the Page Rank of a page, we need to know the Page Rank scores of its neighbors to include them in the calculation. However, when we are investigating a certain page, trying to evaluate its score, we cannot know all the backlinks that point to that certain page. Thus, an iterative algorithm needs to crawl all the pages in the web and try to detect all the links towards the page under question. As a result, when the number of pages is large, this process will require an immense amount of time and computational resources, in order to calculate all
the needed information.

Furthermore, the Page Rank of pages in a graph are directly dependant on each other. If we have a collection of pages with a calculated Page Rank score, and we add a single new page in the collection, we need to calculate the rank for the new page, and we also need to repeat the calculation for all the pages. The reason of this domino effect is the fact that the pages distribute their vote equally among their neighbors. Thus, the Page Rank value of the page that will link to the newly added page will decrease, and as a consequence the Page Ranks of all the other pages that it was pointing to, will have to be adapted as well. This will cause a propagated change that will affect all the pages of the collective that were having backlinks to the affected pages. Thus a Page Rank calculation requires an iterative algorithm that will repeat formula 4.1 several times, until all the Page Rank values reach convergence.

Note that in order for the calculation to begin, the same initial value, larger than 0, needs to be given to all the pages. This value will not affect the final page ranking, since it is going to be shared equally with the other pages of the collective.

4.1.2 Porting Page Rank to Pregel

Page Rank was the algorithm with which we gave a form and structure to MPI*2. During our implementation, we followed closely the instructions that were given in the original Pregel paper in order to achieve a comparable user interface and a compliant behavior for our system.

As previously discussed, webpages and weblinks are ideal to be represented in the form of graphs. Each webpage will be represented as a graph vertex and each link between webpages will be represented as a graph edge. Thus, we define the structures as seen in listing 4.1, which are used to create our Adjacency list as described in section 3.2 in chapter 3. We are using two basic data structures in MPI*2. The Vertex structure that contains information relative to the Vertex and the Edge structure. Each Vertex stores the address of the first node of its edge list and the next Vertex in the Adjacency list, in addition to its ID, its Page Rank value and other implementation specific variables. Each edge stores the address of the next edge node that concerns the same source Vertex in addition to its ID and value. The Edge value is not useful in the Page Rank calculation, and it is only kept for consistency purposes.

```c
struct edge{
  int id;
  double value;
  //Points to the next Edge node
};
```
struct vertex{
    int id;
    double value;
    //vertex connectivity : Total number of edges
    int conn;
    int active;
    //Partition id for this vertex
    int p_id;
    //Points to the next Vertex node
    vertex * next;
    //Points to the list of edges
    edge * out_edges;
    //Used to store the tentative Page Rank value before applying
    //the damping factor
    double sum;
};

Listing 4.1: Vertex and Edge structures

In order to validate the correctness of our implementation, we compared our results with a sequential implementation of Page Rank, as described in listing 4.2. To keep consistency we did not focus on optimising the sequential algorithm. Instead we mapped the steps described in the original Pregel paper [21] in a sequential algorithm.

The first step of the algorithm is to initialize all the vertexes as seen in line 3 of Listing 4.2. Then, the Main() function executes the main algorithm loop for 30 iterations, as indicated in the Pregel paper [21]. In every iteration, the algorithm traverses every Vertex in the list and calculates its propagated PR by dividing its own Page Rank value to the number of outgoing edges, as seen in line 10. This is the point where the vertex divides equally its Page Rank vote among its neighbors. The algorithm performs the same for every Vertex in the edge list of the current Vertex, and adds the propagated value of the source Vertex to the temporary sum of the destination Vertex (line 16). When all the Vertexes and all the Edge lists have been traversed, a last loop of the Vertex list finalizes the tentative Page Rank values as indicated in the equation 4.1. When the main loop finishes all the iterations the Values of the Page Rank will have been stabilized.

INITIALISATION
For all Vertices in list :
    Vertex_Value = 1/NumOfVertices();

MAIN
For i= 0 ... 30 {
    //Propagating Page Rank value to all connected Vertices
    Do{
        Current_Vertex = list.getNextVertex();
    }
Propagated_PR = Current_Vertex.Value / Current_Vertex.conn;

Do{
    Current_Edge = Current_Vertex.getNextEdge();
    Pointed_Vertex = getVertex(Current_Edge.ID);
    Pointed_Vertex.Sum = Pointed_Vertex.Sum + Propagated_PR;
}While (Current_Vertex.checkForNextEdge() != NULL)

}While (list.checkForNextVertex() != NULL)

list.resetPointer();

//Finalizing Page Rank values
Do{
    Current_Vertex = list.getNextVertex();
    Current_Vertex.Value =
                0.15/NumOfVertices() + 0.85 * Current_Vertex.Sum;
}While (list.checkForNextVertex() != NULL);
16 Votes++;
17 ) While ( Votes == maxWorkers );
18 }
19
20 results = 0;
21 Do{
22     result = master.collectResults();
23     master.saveInformation(result);
24     results++;
25 }
26 ) While ( results == maxWorkers );
27
28 master.printResults();
29
PageRank_Worker_Behavior
30
31 // Superstep
32 For i= 0 ... 30 {
33     toCollect = worker.listenForSuperstepCommand();
34     worker.Compute(PageRank, toCollect);
35 }
36
37 worker.SendResults();
38
PageRank_Compute
39 worker.CollectMessages(toCollect);
40
41 Do{
42     Current_Vertex = list.getNextVertex();
43     Propagated_PR = Current_Vertex.Value/Current_Vertex.conn;
44 
45     Do{
46         Current_Edge = Current_Vertex.getNextEdge();
47         DestinationWorker =
48             hash( Current_Edge.ID , TotalPartitions);
49         worker.sendMessage( DestinationWorker, Current_Edge.
50             ID, Propagated_PR);
51         } While ( Current_Vertex.checkForNextEdge() != NULL )
52 }
53
54 ) While ( list.checkForNextVertex () != NULL )
55
PageRank_Collect
56 for i=0 ... toCollect{
57     worker.getMessage(&Vertex_ID , &PR_Value);
58     Pointed_Vertex = getVertex( Vertex_ID );
59     Pointed_Vertex.Sum = Pointed_Vertex.Sum + PR_Value;
60 }
61
32
Listing 4.3: Pregel like algorithm for Page Rank calculation

In Listing 4.3, we describe the structure of the Page Rank algorithm that runs with MPI$^2$. The program is split in 4 essential parts. The Master\_behavior(), the Worker\_behavior(), the Compute() and the Collect() functions. In the Master\_behavior() we can see the basic functionality of the node acting as the Master: it instructs the beginning of each Superstep for all the Workers, and notifies them of the amount of messages that were sent for them in the previous Superstep. Then it waits until all the Workers finish with their computation, before instructing the following Superstep. The number of Supersteps instructed by the Master is the same as the number of iterations in the sequential algorithm, i.e. 30. After the final Superstep, the Master collects the results from all the Workers and terminates the computation.

The Worker nodes also execute 30 Supersteps. In every such iteration, they wait for the "Superstep start" message from the Master, and execute the Compute() function. Compute() first calls the Collect() function to gather all the messages that were sent in the previous Superstep, and performs the necessary updates in the adjacency list. Similarly to the sequential algorithm, Collect() will iterate once through all the messages and update the tentative Page Rank value of the Vertexes. A second traversal through all the Vertexes with an updated Page Rank will finalize their values. This second loop could be avoided with the application of a combiner. If the Master gathers all the messages for a certain Vertex and sends them as a single message to the responsible Worker, then the calculation time is expected to increase. However, this will make the Master functionality more complex and the introduced workload in the Master may slow down the overall performance of this implementation. Although we do not implement a combiner in this work, we believe that their functionality is promising and certainly worth researching in the future development of MPI$^2$.

After the Collect() function returns, the Compute() will continue with the calculation of the propagated Page Rank value (line 47 listing [4.3]) and it will sent it via MPI messaging to all its neighbors. Collect() iterates through the Edge list of every Vertex, and then sends it to the Worker than contains the target Vertex. The discovery of the target Worker is done by hashing the Vertex ID. The hashing
function used, is the same that divided the graph into partitions in the beginning of the computation, as described in Chapter 3 when the Compute() returns the Worker behavior() will wait for the instruction of the next Superstep, or in case the computation has come to an end, it will send the calculated results to the Master.

After the implementation of Page Rank, we can conclude the following: Implementing an iterative algorithm in the Pregel model is matching really well with the inherit Superstep functionality. Using a communication layer that abstracts message sending to a simple worker.sendMessage were we only need to know the ID of the receiver, is making distributed programming much simpler. And finally, as we discuss later in Chapter 5 despite the message exchange our algorithm scales well when using multiple Worker nodes.

4.2 Breadth First Search (BFS)

After porting Page Rank, we chose another algorithm to further validate our MPI 2: a graph traversal. A graph traversal is the process of visiting all the nodes of a graph in a specific manner while collecting or updating node and edge information. A difficulty occurring in graph traversals unlike tree traversals is the possibility to encounter a visited node for a second time. This requires a form of “remembering” the visited nodes, which directs us to graph coloring.

The most common graph traversals are Breadth First search [22] and Depth First search [24]. They take different approaches to the order in which the graph nodes are traversed: BFS chooses a level based approach, while DFS chooses a depth-based approach. Graph traversals can serve many purposes, from finding the solution to a maze problem, to discovering the shortest path between two nodes, or calculating the connected components of a graph.

In this section we present an implementation of a BFS traversal in our MPI 2 model, and we discuss and analyze the performance we have observed.

4.2.1 Algorithm description

The approach of BFS is to first explore all the neighbors of a given node, before progressing to the second level neighbors.

The basic concept of BFS can be summarized as:

*Traverse the graph starting from the initial node and proceeding in a level oriented manner.*
The starting node of the graph can be randomly chosen. This node will be marked as level 0, and the search will continue to its neighbors as seen in figure 4.1. Our BFS algorithm only explores and colours the connected component of the graph that the starting node is part of. Thus, when the starting node changes, the resulting tree representation of the graph will change rapidly according to the current connected components topology. This choice will impact the performance of the algorithm due to the varying complexity of the resulting tree with respect to its height (Levels) and its width (Vertexes per Level). In this experiment we always chose the same starting node, to avoid the differences that emerge from this variation. However, performing several BFS of the graph with randomly chosen starting nodes, would give us the possibility to extract a mean measurement of the BFS calculation and thus have a better approximation on the performance of BFS on different graphs.

In order to implement the sequential algorithm, we followed a process that can be easily transformed for the distributed model of processing as we demonstrate below. First we initialize all the vertexes with the value -1, to indicate that they are unvisited, we mark the starting node as visited, by changing its current (-1) value. We modify this value to 0, indicating that the visited level of this node is 0 (i.e, this is the first visited node).

The Main loop of the algorithm runs until all the nodes from the connected component, that the starting node is part of, are reached. In every loop, the vertexes that will be processed are the ones that are marked as active in the previous loop. For every active vertex, all its edges are traversed and its immediate neighbors are marked one level higher than the source vertex. Only the neighbors that have not been visited before, will update their value as seen in line 20 in listing 4.4. After all the neighbors have been coloured, the vertex is marked as inactive and the computation will continue for any other active vertexes found. We notice that in this algorithm the last iteration will be a full graph traversal and its purpose is only to detect the ending of the calculation.

Figure 4.1: BFS traversal tree of graph starting from node "a"

<table>
<thead>
<tr>
<th>INITIALISATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 For all Vertices in list :</td>
</tr>
</tbody>
</table>
The result of this calculation can be interpreted as the tree representation of the given graph, or the optimal path length solution between the starting node and all other nodes of the graph. The second is true only for non weighted graphs. With Weighted graph, we mean a graph were the Edges can carry a value with negative or positive weight. Thus to solve the optimal path length solution a different algorithm will be needed.

### 4.2.2 Porting BFS to Pregel

Porting BFS to the Pregel model proved to be quite simple. First we designed the interface of the functions that we planned on implementing as seen in listing 4.6. The Init() function is responsible for initializing all the vertexes with the value -1 so they can be recognised as "unvisited". This function simply passes this parameter to an equivalent Init function that is already supported by the MPI$^2$ interface. Thus, the user does not need to have any knowledge regarding the traversal and updating of the Adjacency list.

Furthermore, the functionality of the Master is identical to the standard Master functionality as used in the Page Rank implementation, thus the function can be
completely reused. We did have to modify the Worker functionality according to the BFS specific needs. To avoid an unnecessary message check in the first loop, the Worker will skip the collection step and will move directly to the Compute() method, in the case that the algorithm is executing the first loop (line 42 on listing 4.6). This modification is only an optimisation trick, so, to keep it simple, the user could reuse the Worker function as used in the Page Rank algorithm.

Next, the Compute() method is the one responsible for traversing the neighbors list, and sends the messages to the corresponding Worker. The messages in this case, refer to the level propagation that happens when moving from one Vertex to it’s neighbors. The Worker that contains the father Vertex, needs to send messages to the Workers that contain the child Vertices notifying them of their BFS levels. Once again, this method is quite similar to the Page Rank one, with the difference, that the value that is send is not calculated as the propagated Page Rank, but the current Vertex value increased by one (line 60 on listing 4.6). Finally, the Collect() method, takes as an argument the number of the buffered messages as indicated by the Master, and for every message, updates the corresponding vertexes accordingly (line 79 on listing 4.6).

Listing 4.5: Functions for BFS implementation

```c++
void BFS_Init ( vertex ** head_list );
int BFS_master_behavior(mygraph ** this_graph);
int BFS_worker_behavior( int start_from, int pid, mygraph ** this_graph);
int BFS_compute (int start_from, mygraph ** this_graph);
int BFS_collect (int step_messages, mygraph ** this_graph);
```

```c++
BFS_Init

AdgacencyList_Init(-1, &Adj_adress);
BFS_Master_Behavior

DO{
    votes = 0;
    BFS_ongoing=0;
    For j = 1 ... maxWorkers{
        messages=retrieveMsgNumber (WorkersList[j]);
        WorkersList[j].InstructSuperstep(messages);
    }
    Do{
        currVote = master.listenForVote();
        master.saveInformation(currVote);
        BFS_ongoing = BFS_ongoing + currVote;
        Votes++;
    }While ( Votes == maxWorkers);
```
While (BFS_ongoing != 0) {

results = 0;
Do{
    result = master.collectResults();
    master.saveInformation(result);
    results++;
}While (results == maxWorkers);
master.printResults();

BFS_Worker_Behavior
BFS_Init(Local_Adj_list);
listenForMessage(&startingNode);

DO{
    toCollect = worker.listenForSuperstepCommand();
    if (toCollect == 0) break;

    if (i == 0)
        startComputation = hash(starting_node, TotalPartitions);
        if (pid == startComputation) {
            Adj_List.Update_Vertex(starting_node, i);
            BFS.Compute(starting_node, Local_Adj_list);
        }
    }
else
    BFS.Collect(toCollect, Local_Adj_list);
}While(toCollect > 0);
send.results();

BFS_computer
BFS_CollectMessages(toCollect);

Do{
    Current_Vertex = list.getNextVertex();
    Level = Current_Vertex.Value + 1;
    Do{
        Current_Edge = Current_Vertex.getNextEdge();
        DestinationWorker = hash(Current_Edge.ID, TotalPartitions);
        worker.sendMessage(DestinationWorker, Current_Edge.ID, Level);
    }While (Current_Vertex.checkForNextEdge() != NULL)
}While (list.checkForNextVertex() != NULL)
Overall, implementing BFS in MPI$^2$ was straightforward and did not imply major functionality changes from PageRank. In this case, the algorithm is implemented with a small optimization trick as mentioned before. Due to the simplicity of the algorithm we have no suggested modifications that would improve the performance of our algorithm.

### 4.3 All Pairs Shortest Paths (APSP)

The problem of All Pairs Shortest Paths refers to the calculation of the shortest path for every possible vertex pair in a graph. There are several algorithms that were proposed to calculate the APSP trees. The simplest approach is to repeat Dijkstra’s algorithm [30] for SSSP $N$ times, where $N$ is the number of graph vertices. The safest solution [16] for dense graphs was proposed by Floyd [30] who uses techniques of dynamic programming to divide the problem into many simpler ones. However, the state of the art in APSP is a combination of Bellman-Ford [16] used first to eliminate all negative weights so Dijkstra’s algorithm, applied to the resulting graph [27]. This combination is known as Johnson’s algorithm [30].

The APSP calculation is time consuming and computationally intensive. However, the APSP trees can be used to calculate the graphs diameter, which can be later used as a mean of analysing the graphs complexity [16]. Also, other applications that can benefit from APSP’s results are mapping and routing applications. These are widely used in both research and applications and thus we consider APSP to be an important algorithm to include in our experiments.

#### 4.3.1 Algorithm description

As we previously discussed a benefit of the BFS traversal is that it results in calculating the shortest path from the starting node to all the other graphs nodes. Thus, since the scope of this research is not aiming at implementing the fastest algorithm,
we chose to make use of our previous program in order to see how MPI2 performs in a highly resource demanding algorithm.

In order to achieve an APSP tree calculation using our BFS implementation, we need to divide our problem into many subproblems and systematically apply the BFS calculation in each of them. The subproblems will consist of choosing a different starting node in each, until all the nodes of the graph are used. Hence, for N vertexes we need to apply N times the BFS computation as seen in the listing 4.7.

```
MAIN
DO{
    currentNode=Adj_list.getNextNode();
    If (currentNode=NULL) {
        result=BFS_calculation(currentNode);
        save(result);
    }
    Else
        APSP_ongoing = 0;
}While (APSP_ongoing == 1)
```

Listing 4.7: APSP traversal

### 4.3.2 Porting APSP to Pregel

When porting APSP to MPI2, we had to modify the Master and the Worker behavior slightly in order to find the APSP trees. In listing 4.8 we show the changes in the two functions.

```
APSP_Master_Behavior
DO{
    sendToWorkers (list.getNextVertex());
    BFS_Master_Behavior();
}While (list.checkForNextVertex() != NULL);

APSP_Worker_Behavior
listenForMessage(&startingNode);
While (startingNode > 0) {
    BFS_Init (Local,Adj_list);
    DO{
        toCollect = worker.listenForSuperstepCommand();
        if (toCollect == 0) break;
    }
```

40
```java
if i==0{
    startComputation =
    hash (starting_node, TotalPartitions);
    if(pid == startComputation){
        Adj_List.Update_Vertex(starting_node, i);
        BFS.Compute(starting_node, Local_Adj_list);
    }
}
else
    BFS.Collect(toCollect, Local_Adj_list);
}While(toCollect > 0);
send.results();
listenForMessage(&startingNode);
```

Listing 4.8: Pregel like algorithm for APSP computation

As we previously discussed, APSP is closely related to the BFS implementation and thus a high level of component reuse was possible. This resulted into a straightforward Pregel implementation. However, when we attempted to execute the algorithm we came to realize that the memory consumption of our program was excessive for the cluster nodes. The memory consumed by the Adjacency lists in every Worker, was causing the performance of the nodes to drop, and our jobs were killed several times by the Operating System.

Hence, we discovered that the system’s memory consumption is MPI^2 Achilles heel when run in the DAS 4. When the nodes Operating System had too little free memory in its disposal, it would kill the most memory consuming application running. This behavior was not always consistent and was depending on the load of the nodes in each moment. Thus, sometimes our program was successfully completed and this allowed the results process to continue.

Overall we conclude that our implementation of the APSP algorithm was not taking full advantage of the cluster topology and this had impacts in the performance as we discuss in Chapter 5. Another way of implementing this algorithm, is to unroll the outer loop, spread it among the Workers and instruct separate (starting from a different node of the original graph) BFS computations in every Worker. However, this implementation would mean that the Master will need to keep track of several algorithms running and synchronise the Workers for each separate algorithm. This will introduce a level of complexity in the implementation, yet it is interesting to explore and compare performance differences with our more naive approach.
Chapter 5

Experiments and Results

The goal of Pregel and Pregel-like models is to allow common users to run various graph analytics tasks in an efficient manner. In Chapter 4, we have shown how three graph processing applications can be implemented in MPI2, focusing more on the ease-of-use aspects of the framework. In this chapter, we focus on the performance aspects of MPI2 by analyzing the performance our case-studies can achieve when using MPI2.

We present two different views of our results. First, we discuss the user view, which focused on the overall execution time and the performance gain and/or penalties that MPI2 brings. Next, we discuss the same results from perspective of the system developer, aiming to show where the performance bottlenecks are and discuss how can they be tackled.

5.1 Datasets

For the purposes of validating the implementations of MPI2 we used graphs extracted from the Stanford Network Analysis Platform (SNAP)[19], and graphs from the KGS Go server, collected and processed by the Graphitti team. These datasets were selected to provide different insights on graph processing due to differences in size, topology, connectivity and density, and were bound size-wise by the DAS-4 system limitations. Although, we did not get the chance to identify behavioral differences that are specifically connected to certain graph characteristics, we believe that this can be further explored in the future, when expanding MPI2.

The main graphs used were the following:

- Social network graphs: The SNAP repository offers a wide variety of graphs, including graphs depicting social network interaction, citation networks and web graphs. From this collection, we focus on the social interactions graphs,
which were the most interesting due to the recent growth of social networking through several platforms (Facebook, Linked-In, Wikipedia, etc.). Such datasets allow for a lot of non-computer science studies - like, for example, research regarding the connection between social links and real user interaction [33]. Therefore, we chose the Wikipedia - Vote network for our experiments. This is a directed graph that contains voting data regarding the administrator elections of the Wikipedia users until January 2009. Each Wikipedia user is depicted as a graph node, and an edge between 2 nodes is depicting that the source node voted for the node in the destination.

• Game related graphs: Another social interaction that is widely spread with the expansion of the Internet, is online gaming. We believe that graphs related to gaming activity can be really interesting to understand player behavior, preferences and to extract social structure[32]. For this purpose the Go game was chosen. Go is an online, two-player, strategy game that is traditional in China and Japan. The KGS Go graphs are undirected graphs that represent player interactions on the KGS Go Server. Each player is represented by a vertex in the graph and each player interaction is recorded by the means of a graph edge. The Graphitti team extracted this information and represented it in a specific graphs format. Furthermore, the KGS graph was split into smaller subgraphs that represent different game types. From these subgraphs we chose to use only the KGS_3 and KGS_4 subgraphs. The KGS_3 subgraph represents games that were played in a tournament and the KGS_4 represents simultaneous games between a high skilled player, against several low skilled players.

• Artificial graphs : Apart from real life graphs, artificial graphs were used that depict extreme graph topologies. For this purpose, we created a graph generator, that is capable to create artificial graphs with the topologies of a list, a loop, a star and a sink that were further described in section 2.1. Given as an input the graph type and the desired number of nodes, our generator will create an edge based representation of the described graph, that can be directly used by MPI2. Because we encountered a lot of technical difficulties and a lot of failures in our numerous attempts to run large scale graphs such as the ones in SNAP, we decided to keep these graphs small. This will not favor our parallel approach in terms of performance comparison (i.e., we expect that parallelism and MPI2 will only pay off for large scale graphs), these graphs should be large enough to indicate the performance trends.

Details regarding the size and complexity of the used datasets, are summarized in table 5.1.
Table 5.1: Datasets used

<table>
<thead>
<tr>
<th>Graph name</th>
<th>File Size</th>
<th>Number of Vertices</th>
<th>Number of Edges</th>
<th>Directed</th>
<th>Connected</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>KGS 4</td>
<td>1.4 MB</td>
<td>30978</td>
<td>111662</td>
<td>No</td>
<td>Yes</td>
<td>KGS Go Server</td>
</tr>
<tr>
<td>KGS 3</td>
<td>0.47 MB</td>
<td>4355</td>
<td>17765</td>
<td>No</td>
<td>Yes</td>
<td>KGS Go Server</td>
</tr>
<tr>
<td>Wikipedia Vote Network</td>
<td>1.04 MB</td>
<td>7155</td>
<td>103689</td>
<td>Yes</td>
<td>No</td>
<td>SNAP</td>
</tr>
<tr>
<td>List</td>
<td>0.01 MB</td>
<td>1000</td>
<td>1000</td>
<td>Yes</td>
<td>Yes</td>
<td>Graph Generator</td>
</tr>
<tr>
<td>Loop</td>
<td>0.01 MB</td>
<td>999</td>
<td>1000</td>
<td>Yes</td>
<td>Yes</td>
<td>Graph Generator</td>
</tr>
<tr>
<td>Star</td>
<td>0.01 MB</td>
<td>1000</td>
<td>1000</td>
<td>Yes</td>
<td>Yes</td>
<td>Graph Generator</td>
</tr>
<tr>
<td>Sink</td>
<td>0.01 MB</td>
<td>1000</td>
<td>1000</td>
<td>Yes</td>
<td>Yes</td>
<td>Graph Generator</td>
</tr>
</tbody>
</table>

5.2 Execution Time and Scalability

First, we analyze the overall execution time of each application prototype - i.e., Page Rank, BFS, and APSP. Our goal here is to determine what is the performance gain of our MPI''2 implementation when compared with the sequential version.

Given the parallel nature of MPI''2, we run the computation using different numbers of threads, searching for the configuration that delivers the best performance. This study allows for two interesting results. First, by comparing the best MPI''2 execution time against the execution time that the sequential application achieves, we can estimate the maximum speed-up of each prototype. Second, by analyzing the trend of the performance gain when the number of running nodes increases, we can analyze the scalability of our solution.

5.2.1 Page Rank

The results for Page Rank are presented in Figure 5.1 for the small synthetic datasets, and in Figure 5.2 for the large, real-life datasets. Note that the results recorded for one compute node are in fact the results of the execution of the sequential algorithm (i.e., on a single node, without using MPI''2). When the number of computing nodes is equal to \( n \), we are running the MPI''2 program with one node acting as Master and \( n - 1 \) Workers.

Also note that the computation time for the MPI''2 execution is only covering the computation and communication performed inside the framework. This includes the Page Rank calculation and the message communication between Workers and Master. The environment setup time, as well as the input graph parsing, are excluded from the calculation. We will discuss their impact in the Section 5.3. Input graph parsing is also excluded from the calculation in the sequential execution.

When comparing the sequential execution with the MPI''2 that run with one Worker (i.e., two nodes), we expect the MPI''2 run to show some overhead due to the communication time. This is because in both cases, there is one Worker that solves the entire graph. This is indeed confirmed by the synthetic graphs and the KGS_3 graph. However, for the Wiki-Vote and KGS_4, which are our biggest datasets, MPI''2 shows better performance than the sequential algorithm - by 5%
and 53%, respectively. This happens due to the different way with which the two implementations traverse the graph to update the Page Rank value. In the case of the sequential implementation, the ranking is computed by simply traversing the list of nodes and checking the adjacency list of each of them in order to update the Page Rank. This is a graph-centric approach for computing the page ranking. In the case of MPI2, the node-centric nature of the implementation leads to a more efficient traversal of the graph. In the case of these large graphs, the cummulated gain in performance becomes significant.

In terms of scalability, we observe that with the increase in the number of nodes, the total computation time decreases linearly. For Page Rank, the available parallelism depends a lot on the dataset. For example, for the large, real-life datasets, the sufficient number of nodes for ranges between 12 and 16 (with very small performance differences between these numbers). For the synthetic graphs, the List and Loop datasets show no more performance improvement for more than 12 nodes, while 8 nodes are sufficient for reasonable performance. The Star and Sink datasets are, as expected, showing even less parallelism due to their topology.

In the Star graph, since we only have one source, it will take a single super-step for the Page Rank to be calculated. However, our implementation does not check for convergence, but continues for 30 iterations. Also the Star graph should
be represented as only one vertex entry, since our adjacency list depicts a vertex centered representation. However, when the algorithm reaches the edges of this graph, it will attempt to send a message to them to update their Page Rank. Thus the neighbors of the star graph should have an entry in the adjacency list, even if that contains no edges. This implementation choices make the results of the Star graph to be increasing in execution time with the increasing number of Workers rather than staying constant. This is due to the fact that the list of non connected vertexes will be distributed to the N Workers and in every superstep they will have to collect the messages that the source node will send and update the Page Rank value. We explain the rapid increase in execution time due to communication overloading, meaning that with more Workers the collecting of the messages and the adjacency list updating will introduce extra overhead. Similarly, in the case of the Sink dataset, only the Page Rank of the center node will be affected and will be calculated in one superstep. This reaches the faster execution time when using 4 nodes and then keeps fairly constant as the Worker number increases.

![Figure 5.2: Computation time for KGS and Wiki Vote datasets. Note that the execution time of KGS_4 for one compute node exceeds the time for two nodes by over 7 times. Thus, we chose to reduce the upper bound of y-axis to achieve a visually acceptable result.](image)

Overall, we note that for Page Rank, our MPI2 scales well with the number of nodes up to the limit provided by the dataset. Furthermore, in terms of perform-
ance, it leads to a better graph traversal than the one provided by a simple iterative
traversal over the adjacency list (as implemented by the sequential version), which
leads to significant performance improvements for large graphs.

5.2.2 BFS

We have performed the performance and scalability experiments for the BFS pro-
totype as well. Our results are again presented in two different figures: Figure 5.3
presents the execution time of BFS for the small, synthetic benchmarks, while Fig-
ure 5.4 presents the results for the larger, real-life graphs.

Figure 5.3: The progress of the Computation time of the BFS algorithm over the
increasing number of Worker nodes for the small artificial datasets

For the smaller graphs, we notice three different execution patterns. The List
and the Loop datasets behave similarly: as the number of nodes increases, the per-
formance decreases. This is because of the very low level of parallelism that the
dataset provides. Take for example the difference between using one and using two
Workers. In the case of a single Worker, the whole graph fits in its memory, and it
will be sequentially traversed, in $N - 1$ iterations. When using two Workers, the
list is split in two, hoping that the execution on the two nodes will start in parallel.
As the splitting is done in block fashion - i.e., first 500 vertexes for Worker 1 and
last 500 for Worker 2, the second Worker will only proceed to execute when the
first Worker has finished executing, essentially leading to the Workers running in
sequence, not in parallel. Thus, this is equivalent with the execution on a single node, with the added overhead of the communication. The variations appearing between 10 and 16 processors are likely due to unfriendly division of work.

In the case of the Sink graph, we note a constant execution time. This is consequence of the constant number of steps that need to be taken. No matter from which node (except the sink) the BFS starts, the execution will take one single step to traverse the graph. Only one single node will execute this operation, independently on the number of nodes that are included in the computation. Finally, for the Star dataset, the number of messages required for the traversal is also constant, $N - 1$. However, the more nodes are available in the system, the more inter-node messages are required by the MPI$^2$, leading to increased overhead, clearly visible in Figure 5.3.

What we also see in these graphs is that for all the synthetic datasets, the use of MPI$^2$ does not really pay off. This probably happens due to the fact that the datasets are small enough to fit in a single node cache, leading to a very fast sequential execution, unbeatable by any distributed algorithm.

For the large datasets, we see the expected scalability given by the reasonable

Figure 5.4: The progress of the Computation time of the BFS algorithm over the increasing number of Worker nodes for the KGS and Wiki-Vote datasets

![BFS tree calculation](image-url)
distribution of the computation. We note that parallelism is limited to lower numbers of nodes than in the case of Page Rank, mainly because of the much lower computational intensity (bordering 0, in fact) of the BFS application. Thus, KGS_3 needs only 6 nodes, Wiki_Vote requires 8 to reach peak performance, while KGS_4 only needs 10.

5.2.3 APSP

When attempting to perform the same experiments for the APSP prototype, we have encountered several technical issues. In our APSP implementation, the number of iterations increases so much that the execution exceeds the limitations imposed by DAS4. For example, for 7 Worker nodes, running APSP on the Wiki-Vote dataset is about 82,000 times longer than the execution of BFS (note that this behavior is expected due to the naive parallelization we have used for APSP). Due to this, our experiments were terminated unexpectedly by the systems task manager.

We note that we have encountered the same problems for the sequential application, which was never able to finish processing.

We do not include any results for the synthetic graphs here, as they are fairly uninteresting - the same trends as seen in Figure 5.3 are visible in these results. For the larger graphs, the only dataset that has delivered some reliable results was Wiki-Vote. Thus, we present them in Figure 5.5, which only includes results for Wiki-Vote, achieved with 6, 7, 10, 11 and 22 nodes.

In an attempt to estimate the performance of the APSP implementation for all Master-Worker configurations, we have attempted a simple performance model: we estimate the execution time for the Wiki-Vote based on the performance of the BFS algorithm for the same dataset. As discussed in Chapter 4, the APSP solution we have chosen is based on repeating a parallel BFS for all vertexes of the graph sequentially being chosen as starting nodes of the computation. For our estimation, we assume that every BFS takes roughly the same time, and therefore the execution of the APSP takes \(|V|\) times more than the a single BFS.

We have expected that the difference between the estimated and the actual measured execution time will be relatively small, allowing us to extrapolate from a single BFS measurement to any BFS estimate. Unfortunately, the gap we see is too large to make any real prediction for the APSP execution time. Instead, the increasing gap - 46% faster for the measured performance for 6 Workers, 47% for 10 Workers, and 62% for 22 Workers - gives us some indirect information about the Wiki-Vote graph structure: on average, the execution time of a BFS from a

---

1. As indicated by the system administrator, when the operating system of the nodes estimated that it did not have enough free memory in its disposal, it would shut down the most memory "hungry" application. In our case, the APSP was this application.
random vertex in the graph is significantly shorter than the one we have used in our estimate.

![Figure 5.5: The progress of the Computation time of the APSP algorithm over the increasing number of Worker nodes for the Wiki-Vote dataset](image_url)

We note that the execution time of APSP slightly decreases when more nodes participate in the computation, but this slight performance increase is less than expected from a highly parallel APSP. This behavior is the artifact of the solution we have chosen for implementing APSP - in essence, adding more nodes can only benefit the inner loop (i.e., the parallel BFS); in turn, the parallelism of the BFS is limited by the parallelism of the dataset, which may not be sufficiently high to require more working nodes. In other words, if the BFS tree is very narrow (i.e., the out-degree of its nodes is low), there is little benefit to add more nodes for the parallel BFS. Therefore, we can conclude that our naive implementation is not very efficient on this cluster. In fact, it will only work efficiently for graphs where the out-degree per node is very high. The alternative solution is to parallelize the outer loop - i.e., partition the starting vertexes among the Workers, and let each Worker do a sequential BFS for each of the Vertexes in its partition. This solution is less practical: either the graph needs to be replicated in each node - not entirely possible for very large graphs, or the graph is truly partitioned, but then every BFS will require a lot of inter-node communication. When choosing to reuse the BFS implementation, we limited our options to this parallelization strategy. In case we do not reuse our BFS prototype, an implementation of the Floyd-Warshall algorithm [3]
are probably more efficient for parallelization in a cluster-like environment. This exploration is however left for future work.

5.3 Time distribution

In this Section we take a closer look at the way the execution time is spent in different phases of the of our MPI2 implementation of two out of the three case-studies we made - namely Page Rank and BFS. Note that APSP has been left out due to the very little experimental data we have been able to collect for this application.

Specifically, we investigate the way the execution time is divided among Communication, Input and Setup and Computation throughout our experiments. The Input and Setup time refers to the time the Master takes to read and convert the input graph to an adjacency list. Computation time refers to the time that is spent only in the superstep loop, and Communication time refers to the time spent for sending and processing messages. Since the message sending is asynchronous, it is not possible for us to measure the actual time the message takes to reach the destination. Instead, we measure the time that each Worker takes to receive and process all messages in the beginning of each superstep, because these messages represent the most important component of the communication overhead.

5.3.1 Page Rank

The results for Page Rank are presented in Figure 5.6 and Figure 5.7. From these figures, we observe that the Computation time is indeed decreasing with the increase in the number of Workers. This is an encouraging result, as it shows that the parallelization does help in improving the computation time. In our large datasets scalability is limited to 10 nodes, and in the small datasets Computation time reaches the optimum with the use of 5 Worker nodes. There is the exception of the Star and the Sink graph that reach the minimum with only 2 Worker nodes, for the topology related reasons that were explained in the previous section.

We also notice that the Input and setup time is fairly constant, without really being influenced by the number of Workers. This is somewhat expected, as (1) the Master (one single node) is doing all the work, and (2) the distribution of the data, a one-time only operation, is not that much affected to the number of Workers. This also means that the overhead of the Pregel model on the Master side is roughly constant for tens of nodes.

Finally, we notice that the Communication time is decreasing when more nodes are added. This can seem counter-intuitive, as more nodes should implicitly mean
Figure 5.6: Time distribution of the Page Rank algorithm with the small artificial graphs
Figure 5.7: Time distribution of the Page Rank algorithm with the KGS and Wiki-Vote graphs

more messages. However, in the same time, size of the subgraphs that each Worker is responsible for is decreasing as the number of Workers is increasing. Since message passing is asynchronous (and taken care of by MPI), the only overhead that we measure is related to the processing of these messages. As the number of Workers increases, the processing decreases since each Worker has to go process less Vertices. In other words, the number of messages in the system remains constant, but they are better distributed among the Workers.

5.3.2 BFS

We have performed a similar analysis for the time distribution of the BFS execution. The results are presented in Figure 5.9 and in Figure 5.8 for the large graphs.

We note here that the trends are similar to those of the Page Rank. We can still make two interesting observations. First, we note that the time spent on computation represents a significantly smaller part of the total execution. This is expected, given
Figure 5.8: Time distribution of the BFS algorithm with the KGS and Wiki-Vote graphs
Figure 5.9: Time distribution of the BFS algorithm with the small artificial graphs
that there is virtually no computation in our BFS implementation. Second, we notice that the setup time is a larger proportion of the entire execution time than in the case of Page Rank. This is again the result of the very little computation to be performed by BFS.

5.4 Discussion

When we started implementing MPI^2, we had in mind the Page Rank algorithm, and thus the interface was affected up to a certain level by this influence. After our first successful implementation, we modified the interface to allow portability and component reuse. We created separate libraries for communication, the Master and the Worker helping functions, and the handling of the Adjacency list. This was done to ease the development of the next algorithms, and to turn MPI^2 into a generic Pregel-like implementation.

Implementing the BFS and the APSP algorithms on the MPI^2 model gave us a good insight on how easy is to use the stand alone interface to incorporate a new algorithm. We found the results acceptable: the user still needs some knowledge beyond the graph processing world, but the effort is significantly lower than rewriting by hand.

As with every system, our experience with MPI^2 is advancing with the continuous use. The development time that we spent while porting BFS, was greatly reduced when we ported APSP. As expected, code reuse was possible up to almost 90% and we were able to employ code segments that were serving the same purpose in previous algorithms with small or even no modifications.

In more detail, we can conclude that the Master behavior can be reused almost in its fullest. New additions could be possible graph information that the user would like to collect, or possible information that will need to be communicated to the Workers. The Worker behavior can also be reused, but certain modifications may be needed in the code to incorporate algorithm specific conditions or changes in the communication with the Master. Furthermore, the Collect function is identical with regards to the collecting of the messages, and only the algorithm specific message processing will need to be adapted. Finally, the Compute method is the one in which we would expect the most changes. However, we noticed similarities also there, because our algorithms were all traversals, and in every superstep would send messages to all the discovered node neighbors. We would experience different behavior with an algorithm like, for example, bubble-sort, that instead of sending messages to all the neighbors in every step, would try to move the processing vertex to the appropriate sorted position.

Having implemented Page Rank, BFS and APSP algorithms, we feel confident that similar algorithms will be fairly straightforward to implement by an "average" user. Such algorithms can be the Single Source Shortest Paths (SSSP)\[30].
calculated with Dijkstra’s[30] or Belman-Ford’s[30] algorithm, Depth First Search (DFS) traversal [24], and also round based election algorithms like Afek and Gafni’s[6] or Peterson’s algorithm[26]. The user must fully understand the goal of the algorithm that is about to be implemented, be able to find similarities with the already implemented ones, and reuse the code by applying only a few modifications.

Using MPI’2, one can handle graph processing tasks knowing in depth only the logical steps of the target algorithms and understanding the functionality of the entities of the MPI’2 as described in Chapter 3. Platform specific knowledge, like node communication and coordination is not needed. This makes the graph processing task transparent and possible for a broad group of users.

However, we must note that the performance delays, caused by the memory consumption of our MPI’2, can cause unpredictable behavior depending on the platform’s available resources. This issue will be targeted in future development of our system that will employ a more efficient memory usage plan, making MPI’2 resilient to systems with limited resources.
Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

In this day and age, information overloading is a daily occurrence. We gather information virtually about any topic available. All this information is stored and later processed in order to extract desired information and reach conclusions. The relational nature of this information allows it to be represented and stored in the form of graphs. These graphs are continuously growing in size and complexity and thus their processing is a challenging task. This task is no longer doable by a single desktop or even server computer within a practical time window. Thus, the data analytics community focuses on parallel and distributed computing. In this work, we focus on the particular case of distributed computing, more suitable for the increasing size of the graphs to be studied.

In order to perform graph processing in a distributed environment, one needs to master several system specific requirements including parallelism and architectural knowledge. To make graph processing more transparent and straightforward for the regular user, many systems attempted to abstract it in a level that most system specific details are automatically handled, and the user is left to deal with algorithm specific issues. Such a system is Google’s Pregel\cite{21}, a computational model that expresses programs from the perspective of vertexes, which run a series of iterations; distribution and parallelization details remain hidden behind an abstract API.

In this thesis, we designed and implemented a system that follows the basic principles of Google’s Pregel, which we call MPI’2. MPI’2 is implemented in C++ and uses MPI for message passing. The structure and the computational model, described thoroughly in chapter 5 is very similar to Pregel, with very few small deviations where necessary.

Along our implementation, we stumbled upon several difficulties. At first, there was a certain degree of ambiguity in the Pregel paper, regarding specific details like
partitioning and sharing the input graph or the exact role of the aggregators. Coming up with a smart partitioning plan that minimizes the time spent on information exchange, and successfully processes and shares the graph information between Workers and nodes, was the first milestone.

After the design was complete, we faced a lot of challenges and limitations, typically arising when programming a distributed system in an MPI-like model. The shared output increased the difficulty of tracing back bugs with traditional message printing. Furthermore, relying on debuggers like gdb, was not as straightforward as when programming in a single node system. The main bugs that were occurring in our program proved to be very diverse, ranging from algorithm-specific details (rounding differences between nodes) to communication-specific MPI issues. Due to the lack of tools, debugging and validation of the algorithms correctness required a lot of effort.

Additionally, several issues that affected our program were related (in some cases) to our clusters traffic and limitations. During working hours the job runtime was limited to 15 minutes to prevent cluster monopolization. That forced us to execute our longer experiments on non working hours. However even on late hours DAS-4 was quite busy due to its wide use of MSc and PhD students that had to meet paper and thesis deadlines. Thus executing long jobs in our cluster proved to be problematic.

Another issue that we encountered while using DAS-4 was sudden termination of our jobs due to memory consumption. When the operating system of the node, could not guarantee correct functioning due to lack of available memory, it was killing the most memory-hungry application. This was disturbing most of the times our experiments, when processing large datasets, due to the high memory and swap demand of MPI^2.

To study the usability and the performance of our system we chose to implement three graph processing algorithms. The first such algorithm, Page Rank, allowed us to compare our results with an algorithm that was already discussed by the Pregel paper[21], and to gain insight into how efficient MPI^2 can be for a complex graph processing computation that is widely used throughout the web. We noticed that the system scales well when increasing number of Workers. However the available parallelism is limited according to the dataset’s topology, and we have noticed a performance plateau for all our datasets. Also, we noticed that the communication overhead of the Pregel model remains fairly stable and is causing little performance penalty.

For our second algorithm we chose to address the popular problem of calculating the shortest path between two nodes, with the BFS graph traversal. Our choice was driven by the wide use of BFS in the graph processing field, as well as by its
usability in real life (for travel, logistics, or relationships). Furthermore, a traversal allowed us to test our system with virtually no computation overhead, and solely focus in the communication performance and overhead. We observe that from the user’s perspective, algorithm development in MPI$^2$ is straightforward, allows for a lot of flexibility, and does not require specific system knowledge. Regarding the performance, our BFS performs similarly to Page Rank with regards to scalability but also system overhead.

Finally, we implemented APSP, a highly repetitive algorithm, with many similarities to our BFS algorithm. Our approach to compute All Pairs Shortest Path was based on a repeated execution of the parallel BFS for every graph node in our datasets. With this naive approach, we stumbled upon several system limitations that forced us to reduce the size of the datasets, and caused sparse results. Nevertheless, our APSP implementation proved once again the ease of use of MPI$^2$ and demonstrated that the Pregel model is not resilient to system limitations.

Based on the design and implementation effort for these algorithms, we can state that the Pregel model is indeed much easier to program for than using MPI for implementing applications from scratch. However, one needs to adjust to expressing the computation task at hand into a vertex-based algorithm. While this is not necessarily trivial, this much higher level of abstraction that Pregel (and, of course, our implementation thereof, MPI$^2$) allows for a much easier and quicker trial-and-error approach towards the correct algorithm, even for non-computer scientist users.

Based on our experiments, we can say that the performance results and the scalability of the system are promising, but care needs to be taken when handling the system resources. However, we acknowledge that our implementations were hardly optimized. Our goal was to implement and improve our MPI$^2$ rather than the applications that can be programmed in it. It is clear that there are multiple ways to implement the same algorithms, which might as well outperform these first attempts.

Overall, we believe the main contribution of this thesis is the design and implementation of a functional prototype for a Pregel system using MPI. The system will require additional tuning and potential architecture-specific improvements to become efficient, but it is already able to run graph processing Pregel-based applications.

6.2 Future Work

MPI$^2$ opens the door for a lot of possibilities for research and development. Due to limitations of our cluster, and due to the great memory requirements of our imple-
mentation, we could only use input graphs up to a certain size. The biggest dataset that was used for our experiments was a bit over 1 MB. This is too limited to allow to draw definitive conclusions about graph processing for large scale graphs. As a next step, the current prototype needs to be modified such that it allows an optional load of information in the systems memory, depending on the systems resources. This will allow MPI² to process larger graphs using such restricted clusters.

As a next step, we would like to run MPI² in bigger clusters and supercomputers like IBM’s Blue Gene that offer larger memory capabilities. Then we can compare the performance of MPI² using larger datasets that represent real life large scale graphs. Furthermore, we can see how the system performs having part of the graph loaded in memory and part stored in the file system. Will the overhead of dynamic loading from the file system cause poor performance, or will allow possibilities for processing even larger datasets?

In this implementation, fault tolerance was not implemented and was not absolutely necessary due to the small size of the datasets and the relative short run times. However, with the expansion of our system, and the use of larger input graphs, node failure will happen with bigger probability. Thus, implementing a fault tolerant system as described in the Pregel paper[21] will result in a more reliable system that will also resemble Pregel more closely. The initial approach can be fault tolerance by checkpoints. However, taking into consideration the large size of the graphs, refined recovery could result in a more efficient system and this is certainly worth researching.

Finally, after applying the previous improvements in our system, we will have the required infrastructure to support changes on the dataset (addition / removal of new nodes and edges), and changes on the Worker set (addition / removal of Workers), on the fly. This will be really important and will result in a system with more real life uses. Most of the graphs that are subject to processing are result of peoples interactions. These interactions, either being bank transactions, social site connections or web activity, are mostly continuous. This results in graphs that are continuously growing. In order to process those graphs, we can either extract them and perform a computation in the subgraph, or we can support on-the-fly graph changes and continuously update out results depending on the new information.

The same scenario can be imagined for the underlying hardware platform (i.e., our distributed system). Most systems will have a subset of computer nodes available and assigned to each process. The number of available nodes can change according over time, depending on the load of the cluster, the execution time of other applications, or even the time of the day. Thus, there can always individual nodes that remain unused. With on-the-fly node addition of computational node (known as an elastic platform from the field of cloud computing) we will be able to employ more nodes in the moment they become available, and release them when
they are needed again by other system users. In this way we can exploit the systems resources to the fullest. Both dynamic graph and Worker changes are interesting long-term research directions.
Bibliography


[34] Ian H. Witten and Eibe Frank. *Data mining: practical machine learning tools and techniques.*