Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Trabajo Fin de Grado

HETEROGENEOUS ARCHITECTURE FOR THE OPTIMIZATION OF LARGE-SCALE GRAPH PROCESSING DATA CENTRES

Clara Lucía Alonso Martín

12 de Julio de 2015
Trabajo Fin de Grado

Título: Heterogeneous architecture for the optimization of large-scale graph processing data centres

Autor: Dña. Clara Lucía Alonso Martín

Tutor: D. José Manuel Moya Fernández

Departamento: Ingeniería Electrónica

Los miembros del tribunal

Presidente: D. Rubén San-Segundo Hernández

Vocal: D. Juan Antonio López Martín

Secretario: D. José Manuel Moya Fernández

Suplente: D. Pedro José Malagón Marzo

Acuerdan otorgar al Trabajo Fin de Grado la calificación de:

Madrid, a de Julio de 2015

Fdo.:
Trabajo Fin de Grado

HETEROGENEOUS ARCHITECTURE FOR THE OPTIMIZATION OF LARGE-SCALE GRAPH PROCESSING DATA CENTRES

Clara Lucía Alonso Martín

12 de Julio de 2015
Abstract

With the Cloud Computing boom, there has been a growing increase of data processing applications and therefore, achieving more efficiency in data processing data centers has become more relevant. The object of this project is obtaining tools to analyze the viability and cost-effectiveness of designing Data Centres specialized in processing data with adapted cooling systems, architectures, etc.

Some data processing applications benefit from software architectures, while others can be more efficient in hardware architectures. As there are already some software architectures with very good results in graph processing such as XRegel, a hardware architecture will be described in VHDL in this project, implementing Google’s PageRank in a scalable way. This algorithm has been chosen due to the fact that it could be more efficient in hardware architectures, because of its characteristics, that will be explained later in this abstract.

PageRank is used to measure the importance of a page web in the internet and its position when you look for a keyword in Google’s search web. This algorithm is based on Graph Theory; every web page being a vertex of a graph; and links between pages, the edges of that graph.

In this project, an analysis of the state of the art will be performed first. XRegel PageRank implementation is considered one of the most efficient graph processing systems in time and energy. That is the reason why that implementation will be studied.

However, due to the fact that XRegel computes graph processing algorithms in general, it does not take into account some of the specific characteristics of PageRank, so the implementation is not optimal, and therefore leaves some room for improvement. This is because in PageRank, storing every message sent by one vertex is a waste of memory; they are all the same and equal to their PageRank.

The design will be described in VHDL, taking into account the above mentioned PageRank characteristic, avoiding the storage of the same message several times.

VHDL has been chosen because present operative system architectures do not scale in an efficient way. The objective is to see whether the processing would be more efficient in a new architecture.
The next step will be to write the code, making an scalable structure, because usually millions of page webs or vertexes are involved in each iteration. Afterwards, the code will be synthesised and tested in an FPGA. The last step will be the evaluation of the implementation and of possible improvements to increase energy efficiency.

It was considered a reduction in energy consumption due to the fact that nowadays large data processing is becoming increasingly relevant with Internet expansion, and the new Internet of Things. This presents a big problem in large data centres because they consume a lot of energy along with their cooling systems, so any reduction in this area is crucial to continue expanding them or just keep them going more efficiently.

Resumen

Con el auge del Cloud Computing, las aplicaciones de proceso de datos han sufrido un incremento de demanda, y por ello ha cobrado importancia lograr más eficiencia en los Centros de Proceso de datos. El objetivo de este trabajo es la obtención de herramientas que permitan analizar la viabilidad y rentabilidad de diseñar Centros de Datos especializados para procesamiento de datos, con una arquitectura, sistemas de refrigeración, etc. adaptados. Algunas aplicaciones de procesamiento de datos se benefician de las arquitecturas software, mientras que en otras puede ser más eficiente un procesamiento con arquitectura hardware. Debido a que ya hay software con muy buenos resultados en el procesamiento de grafos, como el sistema XRegel, en este proyecto se realizará una arquitectura hardware en VHDL, implementando el algoritmo PageRank de Google de forma escalable. Se ha escogido este algoritmo ya que podría ser más eficiente en arquitectura hardware, debido a sus características concretas que se indicarán más adelante.

PageRank sirve para ordenar las páginas por su relevancia en la web, utilizando para ello la teoría de grafos, siendo cada página web un vértice de un grafo; y los enlaces entre páginas, las aristas del citado grafo.

En este proyecto, primero se realizará un análisis del estado de la técnica. Se supone que la implementación en XRegel, un sistema de procesamiento de grafos, es una de las más eficientes. Por ello se estudiará esta última implementación. Sin embargo, debido a que Xregel procesa, en general, algoritmos que trabajan con grafos; no tiene en cuenta ciertas características del algoritmo PageRank, por lo que la implementación no es óptima. Esto es
debido a que en PageRank, almacenar todos los datos que manda un mismo vértice es un gasto innecesario de memoria ya que todos los mensajes que manda un vértice son iguales entre sí, e iguales a su PageRank.

Se realizará el diseño en VHDL teniendo en cuenta esta característica del citado algoritmo, evitando almacenar varias veces los mensajes que son iguales. Se ha elegido implementar PageRank en VHDL porque actualmente las arquitecturas de los sistemas operativos no escalan adecuadamente. Se busca evaluar si con otra arquitectura se obtienen mejores resultados.

Se realizará un diseño partiendo de cero, utilizando la memoria ROM de IPcore de Xilinx (Software de desarrollo en VHDL), generada automáticamente. Se considera hacer cuatro tipos de módulos para que así el procesamiento se pueda hacer en paralelo. Se simplificará la estructura de XPregel con el fin de intentar aprovechar la particularidad de PageRank mencionada, que hace que XPregel no le saque el máximo partido.

Después se escribirá el código, realizando una estructura escalable, ya que en la computación intervienen millones de páginas web. A continuación, se sintetizará y se probará el código en una FPGA. El último paso será una evaluación de la implementación, y de posibles mejoras en cuanto al consumo.

Keywords

PageRank, energy-aware data centres, Cloud Computing, VHDL, FPGA, graph processing.

Acknowledgements

To my family, who has been by my side all this time, and that has been a great support during the project. To my tutor José Manuel Moya, and all the Green LSI members, specially Pedro and Marina, who have been there through all the process to help me. Thanks for letting me be a part of this research group and for helping me in every step along the way of this project. I also want to thank my friends for being there by my side when I was so stressed and nervous about presenting the project.
# Contents

1 Introduction and objectives .......................................................... 1
   1.1 Motivation and Related work .............................................. 2
   1.2 Objectives ............................................................................ 4
   1.3 Green LSI Framework .......................................................... 4

2 Development .................................................................................. 7
   2.1 Theoretical considerations .................................................... 7
      2.1.1 *XPregel* ................................................................. 7
      2.1.2 *PageRank* ............................................................ 9
   2.2 Requirements Analysis .......................................................... 11
   2.3 *PageRank* implementation ................................................... 14
      2.3.1 Architecture design .................................................... 14
      2.3.2 VHDL implementation .............................................. 15
      2.3.3 Arithmetic issues ..................................................... 21
   2.4 RS232 Transmission ............................................................. 23
   2.5 Performance evaluation ......................................................... 25
      2.5.1 ISIM simulator ......................................................... 26

3 Results and conclusions ............................................................... 33
   3.1 Results .................................................................................. 33
   3.2 Conclusions .......................................................................... 36
   3.3 Future work .......................................................................... 38
List of Figures

1.1 Green LSI optimisation approach ........................................... 5
1.2 Proposed approach for this line of research ............................. 5

2.1 System inputs and outputs .................................................. 11
2.2 System overall ............................................................ 12
2.3 VHDL modules ............................................................ 14
2.4 Master block diagram ...................................................... 16
2.5 Master state machine ....................................................... 17
2.6 Worker block diagram ..................................................... 19
2.7 Worker state machine ..................................................... 19
2.8 RS232 protocol .......................................................... 23
2.9 FPGA-PC connection module block diagram .......................... 24
2.10 State diagram of RS232 module ......................................... 24
2.11 Test graph ................................................................. 26
2.12 Modules states caption .................................................... 27
2.13 States caption zoomed out ................................................. 27
2.14 PageRank stored in memblocks .......................................... 28
2.15 Worker states ............................................................. 28
2.16 Zoomed in worker states .................................................. 29
2.17 Initial master states ....................................................... 30
2.18 Master states to calculate the sum of messages ....................... 31
2.19 RS232 module simulation ................................................ 31
2.20 RS232 module simulation zoom ........................................ 32
2.21 RS232 module: states in between ..................................... 32
List of Tables

2.1  XRegel results ................................................................. 26
2.2  Graph table ................................................................. 29

3.1  XRegel PageRank results .................................................. 33
3.2  XPower Analyzer results .................................................. 35
3.3  Xregel results ............................................................... 36
Chapter 1

Introduction and objectives

Nowadays, the demand of data centres dedicated to data processing is increasing dramatically. This is due to the fact that most technological systems require storing, sharing and processing data: smart-phone applications, security systems, domotics, the new Internet of things and above all, the expansion of social networks. The high cost of having and maintaining a data centre, the hardware, the software updates, the refrigeration system..., make it difficult for companies to have one of their own, because it is hardly profitable. The solution to this problem is cloud-computing, which has, in addition, other advantages such as being able to scale the capacity, and only use and pay for what the company needs at that moment, depending on its system’s demand. That is the reason why data processing centres are getting more important every day. For example, data centres have increased their power usage by 63% between 2011 and 2012 [13] and, according to the Cisco forecast [12], the global data centre IP traffic this year will have an increase of 52% taking 2013 as a reference.

One of the graph processing data centres functions is to process social graphs, which has been becoming more important every day not only because of the expansion of this social networks, but also because more and more companies every year decide to advertise their products targeting their potential costumers thanks to the computing of social graphs.

Some examples of this graph processing data centres would be Google’s or Facebook’s. The challenge of this computing lies in the scale of those social graphs; they typically have millions and millions of vertexes. Another problem that makes them difficult to compute
efficiently is the fact that this social graphs are very irregular, and have parts with a very high density of links between information, and some others with very low density of links.

Graph processing means computing different chunks of information which have relations and links between them [24]. We have a group of vertexes, that are bits of information, and they are related by the links in the graph. A social network can be seen as a graph because it has pieces of information (they can be a Facebook profile, a web page, etc) that have relation between them (two profiles can be friends in Facebook, or a web page can have a link to another). Because of that, we can compute it on a graph processing data centre.

The main drawback of the proliferation of these data centres is the huge amount of energy they consume. The average consumption of a single data centre is equivalent to the energy consumption of 25,000 households [9]. The main source of consumption in a Data Centre comes from the computing power (server, storage and network equipment) and the cooling system [11].

An important part of data centres are devoted to data processing, so it would be interesting to know if creating a specialised data centre is a profitable idea. Nowadays, a large number of data centres are used for graph-processing applications [ref]. However, this facilities do not use any application-specific technique to leverage their efficiency. From a research perspective, it is interesting to assess whether a custom data centre, designed for the particular needs of this graph processing applications could increase energy efficiency and performance. If so, this would open a field of research to analyse the computational and cooling resources needed to design highly-efficient graph-processing data centres. In this sense, this thesis aims to perform a preliminary study on the benefits of using a custom hardware architecture for graph processing. If proved useful, this would enable future research in this area, allowing new hardware architectures to be used to solve complete or partial graphs, in conjunction with software solutions, in an heterogeneous way.

1.1 Motivation and Related work

This work starts by analysing some data processing algorithms, to see whether they can be optimised using a new architecture. That is the reason why, in this project, PageRank [23]
(a graph processing algorithm) has been implemented. We have chosen PageRank, because it is very used nowadays and we think it may benefit from a hardware architecture. The chosen language has been VHDL, a hardware description language, to test the algorithm in a FPGA. This is due to the fact that the most used operating systems do not have good results when it comes to scale the algorithms to bigger or smaller groups of data, so maybe we will achieve a better performance (either in time or energy) implementing a new architecture from scratch. The aim of this thesis, is to follow a new path of investigation, to test if designing data processing centres specialised is worth the investment, in comparison with the improvements achieved.

We have stated before the importance of data processing. Graph processing has proved to increase efficiency for this kind of applications. The main problem that exists with software implementation nowadays in data processing, is that there is not an efficient way of processing irregular algorithms with the architectures of current Operating Systems. The problem lies in the social networks graphs, because of their irregularity of density, and its size. Parallel computing increases efficiency, but it is not trivial to implement when processing large scale graphs, with millions and millions of vertexes [21] [16]. In this sense, an application-specific hardware architecture could increase the efficiency when processing specific graphs.

There are three major computation models for large-scale graph processing: Google’s Pregel, GIM-V (Generalized Iterative Matrix-Vector multiplication), and Xpregel [27].

Google created a system for large-scale graph processing called Mapreduce [14] in 2004 [18], and has been releasing new languages since then, because Mapreduce does not scale well in large-scale graphs. Some of the last languages released by Google are Pregel [22] and Giraph [5]. Pregel partitions the input graph to compute in parallel and therefore reducing computation time and saving energy.

Pregel has been used with a new language called X10 [26], developed by IBM. Together they form Xpregel [10], which introduces several advantages:

1. Xpregel sends messages between different parallel instances -called workers- , reducing network congestion. In some processing algorithms, vertexes only send messages to the workers where the vertex’s neighbour are located.
2. The vertexes are reassigned to the workers in each iteration, in order to reduce the number of messages sent over the network.

3. Each worker processes the vertexes in sub-partitions, and iterates them in parallel. That reduces processing time and energy consumption.

Xpregel has been chosen over GIM-V because the latter is not flexible and it is difficult to program, although it provides a very fast processing. There have been other graph processing systems designed from Pregel, such as Apache Giraph, or GPS (Graph Processing System) [25], but they do not have the same performance as Xpregel.

1.2 Objectives

The proposed goals for this thesis are the following:

- Studying the state of the art and analysing the advantages and disadvantages of implemented graph processing systems. Seeing where their bottlenecks are, what are their results, and under what conditions. Checking whether they work with all kinds of graphs.

- Studying the viability of new graph processing techniques, specifically in heterogeneous computation, using FPGAs.

- Developing a baseline hardware implementation of a graph processing algorithm, in order to have a new experimental platform where the research group can assess the benefits of hardware, and hardware-software implementations of graph-processing applications.

1.3 Green LSI Framework

This thesis studies a sub-line of research of the Green LSI group [2], whose goal is energy efficiency in data centres. That goal is pursued taking the approach shown in figure 1.1

If so, we would propose to follow the same approach than in previous research lines of Green LSI, which is the one presented in the figure 1.1
In this new research line, the approach would be computing graph processing algorithms to understand the behaviour of the architecture, to obtain data and finally optimise the...
performance (figure 1.2). First, however, the goal is to perform an initial assessment of the benefits of having custom-designed graph-processing data centres. This thesis aims to be an initial step towards that assessment.

In this thesis, we will explain firstly the development, where we will look more in depth into the previous graph processing implementations and the implemented algorithm, describing afterwards how the design works and how it is structured to finally include some tests and the architecture performance.

Then, we will show the results, giving the final conclusions and explaining the future work.
Chapter 2

Development

In this chapter we will explain our graph processing algorithm (PageRank) implementation. In the theory section 2.1 we talk about some graph processing systems, specially XPregel, and about PageRank, the implemented algorithm. Afterwards, in 2.2 we explore the system requirements and its overall operation, then in 2.3 we explain our design and implementation, in 2.4 we introduce a module to obtain the final results of the algorithm computation in the PC, to finally show a simulation of the architecture performance in 2.5.

2.1 Theoretical considerations

2.1.1 XPregel

In this subsection, XPregel will be explained in more detail, giving further information about its advantages and disadvantages.

The PageRank algorithm has been implemented in many different programming languages. The implementation in Pregel with X10 (called XPregel) achieves an important optimisation in energy consumption and has been used by some machines in the Green Graph 500, a list which orders machines by their energy performance [1]. XPregel uses Scalegraph [6], a graph library for large-scale graph processing on top of X10 programming language.

XPregel is used to process graph webs, and it consists of the following: There is one processor called the master which coordinates all the other processors named workers. At the beginning, each vertex is assigned to a worker using the round robin algorithm, which
distributes them so that all the workers have more or less the same number of vertexes, differing only in one. Each worker partitions its vertexes to compute those parts in parallel. The worker’s computation is done iteratively, and each iteration is called superstep.

Once the initialisation is done, the first superstep begins. All workers compute its vertexes in parallel. When any of the vertexes finalise their computation, they deactivate themselves and this is known as a vote to halt.

A superstep ends when all the vertexes are deactivated, no messages are being transmitted and all the workers have indicated they have finished. When that happens, the superstep ends and there is a synchronisation stage. In this stage, the master instructs the workers what to do next. After this, another superstep begins and so on.

To end the iterations, we can define a maximum of supersteps, or a maximum error, so that we stop iterating when the difference between the value of a vertex in a superstep and the next is smaller than a defined error.

In XPregel, there is not a fixed proportion between vertexes and workers, it is customised in code, tuning the level of parallelism. By increasing the proportion of workers, each of them will have a more reduced group of vertexes to compute, so the algorithm will process the graph more quickly. Implementing a number of workers higher than the number of vertexes is unnecessary and a waste of resources.

Since not every vertex has the same amount of computation, and some vote to halt before others, the workload in every worker will not be the same. For that reason, XPregel has a dynamic re-partitioning mechanism. Each worker knows the workload of their vertexes, so each superstep, the master gives permission to one of the workers to re-partition them.

Each worker has two kinds of buffers, one for the messages that it has to send (SB or Sender Buffer), and the other for the arriving messages (RB for Receiver Buffer). There is an internal SSB (it stands for Sub Sender Buffer), because the worker computes the sub-partitions of vertexes in parallel, so had it not been for the SSB, each sub-partition would have to save its data at the same time in the same SB, and that would have resulted in
more computation time (it would be a bottleneck). The worker copies the SSBs in the SB when it has ended iterating the sub-partitions.

The mentioned buffers contain the message value and the ID of the destination vertex, so when the buffer is full, the worker sends it to the other workers, that read the id of the destination vertex’s message, and puts the messages directed to their assigned vertexes on the list of the destination vertex.

Those buffers, in the case of PageRank, will store many replicas of the same message because a vertex sends the same value to all its neighbours. That is one of the aspects that can still be optimised.

2.1.2 PageRank

PageRank is a graph processing algorithm created and patented in 1998 by Larry Page (one of the two founders of Google). It is used in Google’s web searcher to decide the order in which the pages appear in the screen, in other words, it decides their importance. Google changes all web pages PageRank every month, re-indexing them.

A web’s PageRank is a measure of its importance from the number of links to it. But calculating it is not so simple, as the number of pages that have a link to it have to be taken into account, and every link does not have the same importance, a link from a web with a low PageRank will have less importance than a web with a high one.

The formula to calculate this is the following [17]

\[
PR(A) = (1 - d) + d * (PR(1)/C(1) + ... (PR(n)/C(n))
\]  

(2.1)

wherein PR(A) is the PageRank of the web page A, d the damping factor, and C(n) the number of outgoing links of page n.

It is possible to interpret PageRank as the probability of ending up in a web page when accessing the Internet. This is known as the problem of the Random walker [8]. The PageRank of a vertex depends on the number of neighbours (or vertexes pointing to it), and the PageRank of those neighbours, because it is not the same if a certain web has a link
pointing from Google web page than if it has a link from an anonymous blog with a few visits.

Each web page spreads its importance, or in other words, its PageRank, to all its neighbours, so each of them gets the PageRank of the origin vertex divided by the number of neighbours the origin vertex has. To summarise, to calculate the PR of a vertex, we have to sum all the contributions from their neighbours PageRanks, and we have to take into account the damping factor.

There is not any page that has PageRank zero because the damping factor does not allow it. That is because someone accessing the Internet can end up in a page randomly. The damping factor introduces the randomness. Therefore, analysing the algorithm from this point of view, the probability of getting to a web page that has a high number of links pointing to it is higher than getting to a web page that has a very small number of links.

The damping factor is used to 'damp down' the PageRank’s value of some of the web pages. It lessens the importance of the links in the page. The first addend, (1-d), is used so the pages that have a small or nonexistent number of backlinks (ingoing links) will not have zero PageRank.

Note that, to calculate the PageRank of a web page, the PageRank value of other webs is required. That is a problem at first, considering we do not have such value at the beginning, but the algorithm is iterative, and it converges to the value of the PageRank of each web anyway. Thus, the algorithm can be run without knowing the real value of the PageRanks, because the algorithm will calculate the solution no matter the provided value.
2.2 Requirements Analysis

To achieve the goals of this project, we need a system that calculates the PageRank results given a certain graph (figure 2.1). We store the input graph, which are the web pages and their links, in a coe file, which has the structure of a table with three columns, origin vertex, target vertex and weight. Each entry represents a link between two vertexes. The output of this system needs to be the PageRanks of the web pages in the input graph. So we also need a way of retrieving the data from the FPGA memories.

For this system, we need a main memory to store the graph, and cache memories to store the PageRank (or the messages between vertexes in each iteration, which is the same thing).

Finally, as the computation is going to be similar in some aspects to the XPregel graph processing, we also need a master and several workers. PageRank has been implemented in XPregel [20], and although this implementation has improved its execution time and energy consumption, it might still be optimised. In PageRank, the vertexes of the graph are the web pages, and the links in the graph are the links between web pages.

As some software architectures have been already implemented for graph processing, in this project we have implemented a hardware one. We have chosen to describe it in VHDL, so we can implement a small scalable system and try to see where is the bottleneck. As in XPregel, some of the vertexes are computed in parallel, which makes the system more efficient. The algorithm has been implemented in the following way (see figure 2.2):

The vertexes of the graph are grouped and assigned to workers. The workers are in charge of computing their vertexes in parallel. Finally the master controls and synchronises the workers.
The master begins the first synchronisation (before superstep zero). It calculates the sum of the messages sent to each vertex (instead of computing them one by one, which reduces memory usage and power consumption because the number of accesses to memory is smaller).

The file graph is stored in the main memory, and it has three columns; the first indicates the origin vertex, the second one is the destination vertex, and the last one is the value of the link (the weight). Each vertex spreads its PageRank between its neighbours, so the value of this last column serves to weigh the share of the neighbour to the PageRank of a vertex.

The master looks for the first vertex (zero) in the second column, when it finds it, (when there is a match between the target vertex and the one stored in the memory direction the master is looking at in that moment), the master reads the link weight from the memory, and the origin vertex. Then, it reads from the cache memory the origin vertex PageRank, and it multiplies it by the weight of the link read before.

When this has been done, the master increases the index of the target vertex, and repeats the same procedure with the next vertex, and so on until it has reached the last of them.

Once all the sums have been computed, the master activates a flag, so that the workers know it has ended the synchronisation.
After that, it is the worker’s turn to process the vertexes. Each worker reads the sum of messages whose destination is the vertex whose PageRank is being calculated. Then it reads the PageRank value from the main memory. Finally with both values, it calculates the new PageRank value.

When a worker has calculated the PageRank of every vertex assigned to it, it signals the computation is done by activating a flag. This way, the master can know when all workers have finished computing in the superstep, and when it can start the next synchronisation.

Once it finishes calculating the sum, a new superstep begins. The processing ends when the difference between the PageRanks obtained in the current iteration and the previous one is less than 0.001, or when 30 workers have iterated the algorithm during 30 supersteps.

One of the main problems encountered when implementing the algorithm has been working with decimal numbers. The solution has been working with power of ten multipliers. There have been problems with the algorithm formula due to the fact that in the formula

\[ PR(A) = (1 - d) + d \times \left( \frac{PR(1)}{C(1)} + ... + \frac{PR(n)}{C(n)} \right) \]

there are multiplications, where the two numbers are power of ten multipliers, so to add it to (1-d), a weighing has been made.

The results are taken from the memories, where the PageRanks are stored.

In this implementation, the number of messages in the network has been reduced a lot, in theory, compared to the XPregel implementation. This way, energy consumption is reduced. Nevertheless, it has a drawback, which is reducing parallelism, a fact that could increase the computing time and energy consumption.
2.3 PageRank implementation

In this section, the architecture design and structure will be shown in 2.3.1. In subsection 2.3.1 we will describe the VHDL implementation of all the architecture modules, and finally the arithmetic precision and how we achieved it will be explained in the 2.3.3 subsection.

2.3.1 Architecture design

In this subsection the design will be explained in detail:

The architectural design that has been written consists in four modules:

1. Master
2. Worker
3. Mem
4. Memblock
5. RS232

As seen in the figure 2.3, the implementation has been structured this way because of the following reasons.

- The master has access to all the memories connected to the workers, because that way it is not necessary to replicate all the data.
• The master calculates the sum of all messages directed to any vertex. This may reduce the level of parallelism and thus increase the time of computing in proportion of the size of the graph, but it also reduces the usage of memories. Had it been the workers the one calculating the sum of all the messages whose destiny is the same vertex, we would have had to implement at least two memories for each worker, connected each one of them to all the other workers. The network would have been much more complicated, and the traffic in it, higher.

• The initial graph is stored in a memory only accessed by the master, for the same reasons above mentioned.

• In the example, there are only three workers, but depending on the number of vertexes and the wanted parallelism, the number of workers can vary.

• Implementing the system this way, we are trying to benefit from PageRank’s specific characteristics, mentioned in the introduction: the PageRank of each vertex is only stored once, and distributed between its neighbours. There is no need of sending as many times as neighbours it has, like in XRegel.

• We have implemented only three workers because the example graph we want to process is simple and has only 5 vertexes, that can be easily computed with three workers.

2.3.2 VHDL implementation

In order to implement the previous architecture, we use VHDL, because it is one of the available resources that we have in the laboratory, and it allows us to make an example implementation to look for the bottlenecks in the implementations, and also let us reassign resources as needed.

In this section, each module of the implementation will be explained in detail

1. Master

As can be seen in figure 2.4, the master is connected to the main memory, where the input graph is stored (databusprin, addressprin and e_ROM to enable read). It is connected as well to the three cache memories, and has, like almost all the blocks
in the architecture, an asynchronous reset. *Terminated* is an input signal from the workers, to indicate they have ended computing in the superstep. When they want to indicate they have ended all the computation, they announce the halt. *Compute* is used to order the workers to start the next superstep.

The master has been described as a state machine (as in figure 2.5, where the inputs and outputs are not detailed because of their quantity, and will be explained in this paragraph), because it makes it easier to implement and read. After reset, the master starts on the *Idle* state.

It starts to iterate superstep zero. Next state, after a clock period, is *DataPR* (after waiting for a clock period in *waitstate* so the memory has time to put the data in the *databus*), when the master reads a data from the second column of the graph in the main memory (mem), where the destination vertexes are stored. There is a signal which indicates the vertex whose sum is being calculated. Next, in *DataIN* step, it compares the vertex read from the memory to the vertex whose sum is being
calculated. If the read vertex is not the one it is looking for, the master goes back to \textit{DataPR}.

If it is a match, the master reads the origin vertex of the message, and then waits a clock period in \textit{Waitstate2}, so that it has time to retrieve data.

![Master state machine](image)

\textbf{Figure 2.5: Master state machine}

Then it passes to \textit{getneighbours}, to store the value of the neighbour in a signal. After that, in \textit{choosingworker}, the master sees to what worker the vertex has been assigned. In this implementation, there is no reassignment, so the assignation is round robin.

The next step is \textit{getPRneighbours}, when the master asks for the PR of the vertex's neighbour to the worker memblock. In the next state, \textit{preparesuma}, it reads the value from the corresponding data bus, and goes to state \textit{adjustPR}, where it stores the neighbour’s PR from a 32 bit signal to a 24 bits one (by previously dividing it by 100), so it can be used to calculate the sum (it will be explained in detail in the next subsection).

Then, in the state \textit{suma}, it adds the contribution of that neighbour’s PageRank to the rest of the contributions of the rest of neighbours of the targeted vertex. Then it goes back to \textit{dataPR} to repeat the same process with all the vertexes.
In dataPR, when the master has read all the destination vertex of the main memory, comparing it to the target vertex, the next step is getAddress, when it calculates to which worker that vertex is associated to, and then ResetSum, when it stores the sum in the memblock. The result state is idle (after waiting a clock period in waitstate3 to update the value of the control signals to notify the workers) if it has ended calculating the sum of messages destined to every vertex, or dataPR again if it is not the case, and it repeats the process with the next targeted vertex.

When all the vertexes have voted to halt, the master stops computing and stays in the idle state. That means there will be no more supersteps.

In the master implementation, it looks in every superstep for the neighbours and weights of their PageRank in case the graph changes during computation. If it is not the case, that part of the computation could be done just in the first superstep, saving a lot of execution time and energy consumption.

2. Main memory (Mem)

This is the main memory, where the graph is stored. As mentioned before, the graph has three columns: the first one is the origin vertex, the second one is the destination vertex, and the last one is the weight of the link. The graph is stored before the master begins to iterate, and it can not be rewritten until the algorithm has finished computing the vertexes.

It has been generated automatically by the Xilinx software IPCore [7], and it is a Single-port ROM, as the master only reads from it, but does not have to write anything there.

3. Worker

This is the block diagram of the worker (figure 2.4). It is connected to its block memory (address, databus, oe_mem and write_en) and has also some connections so the synchronism can be kept (compute, terminate and halt, already mentioned in the Master implementation).
The worker implementation has been also made with an state machine in order to make it easier to compute and understand.

It begins in the state *idle*, while the master is computing. When the master signals the end of its computing, the workers begin their processing.
A worker passes from *idle* to *waitread*, an state to prepare things before retrieving data from memories. Without this state, the worker would retrieve the data that was before in the data bus, or the data that was being processed by the master.

The next state is *readsum*, when it retrieves the sum of the messages directed to a vertex to calculate its *PageRank*. After *readsum* goes *readPR*, when it reads the *PageRank* value calculated in the previous superstep.

Then, in *computevert*, the worker calculates the *PageRank* of a vertex, and after a clock cycle comes the *nextVertex* state, that compares it to the previous one in order to vote to halt if it is equal (which, in this implementation, is equivalent to an error below 10 to the power of -6) and increments the number of the target vertex.

If all the vertexes have been computed, the worker goes to the *idle* state again, signalling the computation in that superstep has ended for that worker. If that is not the case, it goes to *savePR*, to store the value of the calculated *PageRank*, and then again to *readsum*, to start calculating the *PageRank* of the following vertex.

Every superstep, the workers begin in *idle*, and will only compute if there is any vertex which is active, or if the superstep is smaller than 30. That avoids cases in which the algorithm does not converge (in *PageRank* algorithm it will not happen, as proved in [19])

A vertex deactivates itself by voting to halt, only if the error is below the one established (in this case, 10 to the power of -6). That avoids computing the same vertex again and again to obtain the same result.

When all vertexes assigned to the same worker have voted to halt, the worker votes to halt, putting a signal to high level so the master knows it has ended computing.

4. Memblock

There is one memblock for each worker. It stores the *PageRank* of its vertexes, and the sum of all the messages directed to them. It has one port, connected to a bus accessed by the worker and also by the master. It is a Single port RAM generated automatically with the *IPCore* because we think it is easier to implement.
2.3.3 Arithmetic issues

One of the main problems encountered while describing the implementation has been working with decimal numbers. As this project has been tested on an FPGA, the registers used are of 32 bits size. This gives an output of PageRank with a maximum error of 0.000001. The explanation is the following:

The maximum value of PageRank that a web page can have is 10 (it is really difficult to achieve that PageRank, but not impossible). For example, Google, which has a very high PageRank, has a value about nine. As VHDL has not a module implemented to work with decimal numbers, I have worked with power of ten multipliers, as I mentioned before. So we see that using 32 bits registers, we have:

\[ 2^{32} = 4294967296 \]

which means we can represent PageRanks from zero to ten with eight decimals. We have the following formula then, multiplied by \(10^8\)

\[ PR(A) = \frac{(1 - d)}{N} + d \times sum \]  \hspace{1cm} (2.2)

wherein \(N\) is the number of vertexes.

\[ 10^8 \times PR(A) = 10^8 \times (1 - d) + 10^8 \times d \times sum \]  \hspace{1cm} (2.3)

But because we cannot have decimal numbers, we cannot represent \(d\) (the damping factor) without scaling it. As a common value for it is \(d=0.85\), the chosen value to store it has been \(d=85\). On the other hand, the same issue is observed with the \(sum\). It is stored in 32 bits registers, and from the formula 3.2 we can calculate the maximum sum for \(d=0.85\), which is 11,58823. Bearing that in mind, we can store the \(sum\) with 8 digits again, as in the PageRanks case. So to sum up, we have the PageRank multiplied by ten to the power of 8, the damping factor \(d\) multiplied by 100 and the \(sum\) multiplied also by \(10^8\).

That way, we have to weigh the 3.2 equation in the following way:

\[ PR(A) = 10^6 \times (1 - d) + 10^2 \times d \times 10^8 \times \frac{sum}{100} \]  \hspace{1cm} (2.4)

There are also some problems with the equation to calculate the sum:

\[ sum_n = sum_{n-1} + \frac{PR(n)}{C(n)} \]  \hspace{1cm} (2.5)

wherein \(1/C(n)\) is the weight of a vertex.
As the *sum* has 32 bits, the weigh maximum value is 100 and for that we only need 7 bits, we could use a *PageRank* of 32-7-1=24 bits (note that the result of a multiplication between two bit vectors, has a size equal to the sum of both sizes plus one). With 24 bits, we can represent up to

\[ 2^{24} = 16777216 \]

so we only have to weigh that in the two equations (3.2 and 3.4). With 24 bits, we have 6 decimal digits, and therefore a maximum error of 0.000001.

This precision is quite alright for a five vertex graph, which is our test graph, but for a large scale graph we should minimise the error in the *PageRank* output, maybe considering a 64 or 128-bits architecture. It depends on the size of the graph the system is going to process. As we are considering large-scale graphs, reducing the error is necessary because lots of web pages will have a very similar *PageRank*, differing only in millionths or less, and lacking the precision needed means not being able to establish a web page ranking correctly.

When the masters retrieves a PR data from a memblock to calculate the sum of messages directed to a vertex, it divides it by 100, to store it from a 32 register to a 24, and thus being able to use it in the equation 2.5.

That was the arithmetic study we made at first, but with that design (32-bits words, d=0.85 and 6 decimals precision), the maximum input clock was around 30MHz, and the used FPGA (Nexys 4 from Digilent) has a 100MHz clock, so we introduced more registers.

In the post-route analysis report we could see where the bottleneck in frequency was: It was on the line of code where the workers calculate the *PageRank* from the sum of messages. It is an operation that takes a lot of time mainly because of the division by 100 in the formula 2.1. To speed it up, we decided to divide by a power of two. That solved the problem and increased the maximum frequency to 130MHz, but we also lost several decimals of precision there if we decided to keep the damping factor as 0.85. We have to chose then, between precision and speed with d=0.85. Changing it slightly, however, we can have the best of both aspects. The damping factor may vary, so in the best case, we will have a 6 decimal precision and work at 100MHz.
2.4 RS232 Transmission

In order to retrieve the data from the FPGA, an implementation of a module has been made. It uses the RS232 protocol to communicate with the graph processing system. To do that, the easiest way is to use a state machine. The RS232 protocol is the following:

Firstly there is a low level bit to indicate the beginning of a new data frame, then the transmitter sends the data, and finally it sends a high-level bit to indicate the end of transmission. While there is no communication, the channel remains at a high level state. The transmission speed is 115200 bps and the protocol is RS232 8N1. As we have to send 8-bits words, we divide each 32-bits data in 4 words. It is a little endian transmission (similar to networks protocols).

In figure 2.9 we can see the RS232 module block diagram, where we observe that it is connected to the three memblocks, and it has a control input signal named start, so that it can know when the computation is done and it has to start sending the results to the PC.

This is the state machine diagram designed for its implementation (figure 2.10):

First we are in Idle state, waiting for the master to end the computation. When that happens, the workers alert the RS232 module, and it passes to the GetAddress state, when it copies the address of the data we want to retrieve on the address field of the corresponding memblock, depending on the worker.

The next state is StartBit, when the output (TX) is at low level, to indicate the module is going to send a data frame.

Then, in_SendData state, the module sends the retrieved data bit by bit, through the serial port. When it has finished, in StopBit state the output is at high level to indicate
the end of the data frame. As the words are 32-bits and that means we have to send them in four parts, it checks if it has finished sending that four bytes; if not, it goes again to StartBit.

Finally, in NextData, it checks if it has finished retrieving all the data or if it has to retrieve data from another worker. If that is the case, it goes back to GetAddress, sometimes going through WaitAddress first so there is time to retrieve data from the memory; but if
it has finished, it goes indefinitely to idle.

In order to be able to see if the architecture was working correctly, we wanted to check the output data (TX) with an oscilloscope. For that, we implemented the RS232 module so it sends the output data again and again until it is reset. Otherwise, it was very difficult to see the results in the oscilloscope, but in the final implementation, the RS232 module would send the results only once.

So it is an automatic way of getting the results in the PC. When the algorithm converges, the RS232 module begins getting the \textit{PageRank} from the memblocks in order. That means, in this architecture, that first it has address zero, and retrieves the data in every memory for that address, then it increments that address to follow the same process again, and so on until it reaches the number of vertexes in the graph.

To receive the data in the PC, a library from the Green LSI has been used. It is called \texttt{mod\_rs232\_listen}, and is a server that opens a socket, and receives the data trough the serial port, given a transmission speed and a port. Then we have programmed a client in Python that can connect to the \texttt{mod\_rs232\_listen} socket to retrieve the data, joining the 4 bytes of each data to print the \textit{PageRank} results in the console.

\section{Performance evaluation}

The implementation has been simulated using the ISIM simulator, which is part of the Ise Xillinx software [15]. To do that, a simple graph has been chosen, to easily check if the results were correct. Due to the fact that the structure is scalable, if the results were correct with a small graph, they would be correct too when computing large graphs.

The example graph is shown in the figure 2.11

It has been processed in \textit{XPregel} too, in order to compare results, obtaining the following \textit{PageRanks} (see table 2.1):

See that the results in table 2.1 are coherent, because vertexes 0 and 1, which do not have any back links, have the same \textit{PageRank}, equal to 0.15/n, n being the number of vertexes. Vertex 2 \textit{PageRank} is the highest, because we can see it has the highest number of backlinks. Finally it also makes sense that vertex 4 has a higher \textit{PageRank} that vertex 3, because each of them have only a backlink, but the one to vertex 3 is from vertex 1, which
Table 2.1: *XPregel* results

<table>
<thead>
<tr>
<th>Vertex</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>1</td>
<td>0.03</td>
</tr>
<tr>
<td>2</td>
<td>0.1018621875</td>
</tr>
<tr>
<td>3</td>
<td>0.04275</td>
</tr>
<tr>
<td>4</td>
<td>0.04816875</td>
</tr>
</tbody>
</table>

has the lowest *PageRank*, while the one to vertex four is from vertex 3, which has a higher *PageRank* that vertex 1.

### 2.5.1 ISIM simulator

In this subsection, the simulation of the system in the Xilinx software will be explained in depth.

When testing the code in *XPregel*, the file in X10 called SimplePageRank.x10 has been modified to get the results, including the number of iterations, the *PageRank* of the vertexes in each iteration, the overall computing time and also in each iteration, and the memory usage. In the simulation, we can see (figure 2.13) that whenever the master is computing, the workers are in idle state and vice versa.

\footnote{This simulation has been done after changing the code to make arithmetic shifts in the register instead of divisions, so the output data is slightly different because those changes result in a change of the damping factor, which can be varied.}
Note that in this example, worker zero has two vertexes (0 and 3), worker 1 has two vertexes (1 and 4) but worker 2 has one vertex (2) because they have been assigned with round-robin scheduling. Bearing that in mind, it makes sense that in the figure 2.12 can be observed that the worker 2 ends computing before the others.

In figure 2.6, it can be noted that the synchronisation time is higher in comparison with the time the workers spend computing the vertexes. The synchronisation time gets higher in proportion when the number of vertexes increases.

We can also see, in figure 2.14 that the PageRank of a vertex is stored in the memblock connected to the worker to which the vertex is assigned. In the figure, the blue signals are the vertex PageRanks. Note that this is the only figure of this section of the implementation with $d=0.85$, so the results can be compared with those of the example graph described in the previous subsection.

Note that the signal at the top (halt) is "111", that indicates that all workers have ended computing their vertexes. So the value in the RAMs is the final value of their PageRanks, which matches the values given by the XPregel system with a maximum error of $10^{-6}$.

In figure 2.15, we can see the states in which the worker has to compute. This capture shows that worker zero signals (id is "00"). The signal where we have the PageRank read
Figure 2.14: PageRank stored in memblocks

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>control</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dh</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>revl</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbaccess</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ushutdown</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>superstopk</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sabw elit</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prime_time</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>terminate</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c3id</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>num vertex</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current_file</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next state</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read bg</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read mchr</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read spp</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read st</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read to</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read to_ha</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read to_hb</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.15: Worker states

from the memory to compare to the next one to know if the algorithm has converged is shown behind signal sum and it is called pranterior. The sum, needed to calculate the vertex PageRank, is showed in blue. Note that vote_to_halt is zero, so there is not any vertex that does not have to be iterated the next superstep. Terminate is also zero, therefore the worker has not ended computation.

We can see how it is computing the first vertex that it has assigned (in this case, as it is worker 0, when i signal is two and it is done storing the last PageRank, it will stop computing).
Figure 2.16 shows the same states zoomed in, to see more clearly the name of each state. Note that `halt` is zero, so the worker has not ended computing. Although not shown in this figure, `compute` would be at high level, that means the master is indicating the worker to compute.

The master is shown in the Figure 2.17, where we can see how it is looking in the main memory for a match with the current target vertex (2, in the red signal). The master is summing the messages from all the neighbours to the targeted vertex, and the signal `neighbour` indicated that it is summing the message (or the PageRank contribution) from vertex 1 to 2 (in signal i). The graph would be stored in the following way (see table 2.2).

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.5</td>
</tr>
</tbody>
</table>
So therefore the first column is a match, and the master reads the weight, (0.5) (see \textit{databusprin} in green).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{Name} & \textbf{Value} & \textbf{M0(0)} & \textbf{M0(1)} & \textbf{M0(2)} & \textbf{M0(3)} \\
\hline
 databusA & 00000000 & 0 & 0 & 0 & 0 \\
 databusB & 00000000 & 0 & 0 & 0 & 0 \\
 databusC & 0 & 0 & 0 & 0 & 0 \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 databusX & 0 & 0 & 0 & 0 & 0 \\
 \hline
\end{tabular}
\caption{Initial master states}
\end{table}

As the master has found the targeted vertex (figure 2.18), it has added to the messages (\textit{sum}) the multiplication of that weigh by the \textit{PageRank} of the source vertex of that message. It has done the same with all the vertex neighbours, until \( j \) became equal to the number of memory addresses in the main memory. Then the master reads which vertex is the source, in this case it is vertex 1 (because we are in the first entry of the main memory). Afterwards, it stores the \textit{sum}, adjusting the \textit{PR} first so that the result of the multiplication fits in the bits destined for the result.

In figure 2.19 we can see how the module RS232 retrieves the data from the memories, to send it to the PC. The blue signal is the serial port, that is connected to the PC.

We connect a device, the PmodRS232 [3], to the FPGA JA ports, so it can connect to a serial port in the PC.

The green signal is the data being transmitted at that moment, which is taken from the memblocks. It puts the same address, at first zero, in each worker and goes retrieving the data from that address in order (first worker0, then worker1...) until it reads the data from
the last worker. Then, it increments the address and repeats the same process. As we can see, the RS232 module will not start sending data until all workers have voted to halt.

Note that the time the systems spends computing and iterating the algorithm is way smaller than the time it takes to communicate with the PC and send the data. The compu-
tation is done, in this case (in the simulation), in 6 us, while to retrieve data it takes around 1700 us.

![Figure 2.20: RS232 module simulation zoom](image)

In figure 2.20, we can see the same signals, zoomed on the states between two `sendData` states. There is one bit to indicate the start of the transmission, and another to indicate its end.

If we zoom further (figure 2.21), we can see that there are still more states between `sendData` states. These are very short in time and were explained in the previous subsection.

![Figure 2.21: RS232 module: states in between](image)
Chapter 3

Results and conclusions

3.1 Results

We have compiled all the X10 libraries with Scalegraph, a graph processing library that uses X10, a language for parallel computation, to compile afterwards the XRegel files that contain the XRegel PageRank implementation. Then we have modified the code \(^1\) to obtain the time computing, memory usage and the PageRank results for each iteration. In the table some of this results can be seen:

<table>
<thead>
<tr>
<th>Time computing (ms)</th>
<th>Time computing (cycles)</th>
<th>Memory used (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>51.53</td>
<td>(1.74 \cdot 10^8)</td>
<td>1.32</td>
</tr>
</tbody>
</table>

We have designed and described an architecture in VHDL, using the Xilinx software, specially the program ISE WEBPACK [15]. For an input graph given, that must be a coe file (a graph file) introduced in the main memory, the system stores the results in the FPGA memories, and sends then to the PC using RS232 protocol, to retrieve the results. It has been simulated in the Xilinx software, as seen in the performance evaluation chapter, and then it has been synthesized:

HDL Synthesis Report Macro Statistics (This is the number of logic ports used for the implementation)

\(^1\)In this results section, all experiments have been carried out with a damping factor \(d=0.78125\) in order to achieve more precision while being able to make bit shifts instead of divisions in our architecture.
• # Multipliers : 3
• # Adders/Subtractors : 52
• # Registers : 51
• # Comparators : 56
• # Multiplexers : 391
• # Tristates : 270
• # FSMs : 5

This report also gives information about the used FPGA resources. This is the Device utilisation summary:

• Slice Logic Utilisation:
  Number of Slice Registers: 704
  Number of Slice LUTs: 1155
  Number used as Logic: 771
  Number used as Memory: 384
  Number used as RAM: 384

• Slice Logic Distribution:
  Number of LUT Flip Flop pairs used: 1540
  – Number with an unused Flip Flop: 836
  – Number with an unused LUT: 385
  – Number of fully used LUT-FF pairs: 319
    Number of unique control sets: 30

• IO Utilisation:
  Number of IOs: 4
  Number of bonded IOBs: 4
- Specific Feature Utilisation:
  Number of Block RAM/FIFO: 1
  Number using Block RAM only: 1
  Number of BUFG/BUFGCTRLs: 1
  Number of DSP48E1s: 1

  The utilisation of the FPGA resources is very low as expected (around 1 or 2%), because we are only computing a 5 vertex graph.

  The XPower Analyzer gives the estimated power consume, which can be seen in table 3.2.

  Table 3.2: XPower Analyzer results

<table>
<thead>
<tr>
<th>On-chip</th>
<th>Power(W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks</td>
<td>0.003</td>
</tr>
<tr>
<td>Logic</td>
<td>0.004</td>
</tr>
<tr>
<td>Signals</td>
<td>0.005</td>
</tr>
<tr>
<td>BRAMs</td>
<td>0.002</td>
</tr>
<tr>
<td>DSPs</td>
<td>0.000</td>
</tr>
<tr>
<td>IOs</td>
<td>0.003</td>
</tr>
<tr>
<td>Leakage</td>
<td>0.088</td>
</tr>
<tr>
<td>Total</td>
<td>0.106</td>
</tr>
</tbody>
</table>

  As we can see in table 3.2, most of the energy consumption is due to leakage, and the rest depends on the design. It appears that the signals utilisation, registers, maybe the number of states in the FSMs, etc is the energy bottleneck in this architecture. The next item that has presented more energy consumption is the logic, which is also related to the way the architecture is implemented.

  Then we have implemented the design (Translate, Map, Place & Route) to finally generate the bit stream file and configure the FPGA. It is a Nexys 4, from Digilent, and this is the user manual [4]. In the Place & route report, we can see the following:

  Design statistics: Minimum period: 7.554ns (Maximum frequency: 132.380MHz)
It is a satisfactory value because the FPGA Nexys 4 has a 100MHz clock.

Afterwards, in the post-route simulation we have observed that the estimated time is 5.775ns (5775 cycles) for the PageRank computation, and 1,744ms (1744 cycles) for the whole process to end (PageRank computing + sending the data to the PC through the serial port).

Table 3.3 summarises the most relevant results:

<table>
<thead>
<tr>
<th>Metric</th>
<th>XPregel</th>
<th>Our architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing time (ms)</td>
<td>51.53</td>
<td>5.775</td>
</tr>
<tr>
<td>Computing time (cycles)</td>
<td>$1.24 \times 10^8$</td>
<td>5775</td>
</tr>
</tbody>
</table>

And finally, there are some other considerations about this project:

It calculates the PageRanks of the vertexes with a precision of ten to the power of -6 for some damping factors.

We have noted there is a compromise between memory usage and parallelism. That is because in another implementation, the workers could also calculate the sum of the messages, but that would mean storing all the vertexes’ PageRanks multiple times, connecting each memory to all the workers and with all probability increasing computing time, due to the fact that several workers would have to access the same memory, and the synchronisation would be more difficult.

3.2 Conclusions

Nowadays, most architectures do not scale in a satisfactory way when computing large-scale graphs. In this project the aim was to design and implement a new hardware architecture in order to process PageRank algorithm, to assess if such a solution had the potential to address the issues encountered in software architectures.

---

2We have executed PageRank XPregel implementation in a Linux PC at 2.4 GHz
• We have studied the state of the art, observing that there are several software im-
plementations of graph processing systems such as Mapreduce, GPS, Pregel, XPregel, Giraph, etc. We have centred the study in XPregel because it has very good results and is used in some of the top Green Graph 500 list. In general, the problem with graph processing systems is that they do not scale well when computing large-scale graphs. After seeing this, we chose to implement a HW architecture to compute PageRank, because we think it may benefit from a hardware architecture. We have looked for bottlenecks then in XPregel’s PageRank implementation. We have found that it stores multiple times the same value, for it stores all the outgoing messages from each vertex in a superstep, and in PageRank those are the same and equal to its PageRank. Another disadvantage is that it cannot reassign resources for other types of processing or other algorithms while it is computing.

• A hardware architecture to compute PageRank has been proposed that complements the software ones that have already been made. In particular the XPregel one (implemented with X10, which is a language for parallel computation). The advantages of this implementation are the following:

  – A hardware architecture in a FPGA can have multiple work modes, so that if we are processing a certain algorithm and it does not use all the available resources, we can use the remaining FPGA resources to do some other work or computation.

  – In our proposed architecture, we can have different amount of bits in a word, depending on the precision needed, so there is not a waste of resources in that way.

  – We have noted that the energy bottleneck may be in the number of signals used and the registers inferred or in the number of states that the FSMs have.

• A hardware architecture, able to satisfactorily compute PageRank has been imple-
mented. Although the goal was not optimising in any way, we have assessed our solution in terms of performance and power:

  – We have achieved a reduction of computation time of 88.79% , and a reduction of the computing cycles of 99.99% comparing with the XPregel version .
– Data memory usage (72B) (It can not be compared to the results in XRegel, because the usage memory results we obtained are also due to the execution of the program and its libraries).

– An estimated power consumption of 0.106W (0.088W of leakage), using 704 slice registers and 1155 slice LUTs.

3.3 Future work

Considering the PageRank implementation carried out in this project, the following future work is proposed:

- This project opens a new research line, aligned with the previous work in this data processing field and with the Green LSI work (see figure 1.2). The steps to follow are testing the architecture in data centres to obtain data and see how it behaves with large-scale graphs, obtain some behavioural models and actuate in the data centre to optimise it. The big difference introduced with this new line of research is that, until now, the Green LSI team had developed optimisations at higher abstraction levels (reassigning workloads, designing cooling systems and prediction models, etc). This work opens a new research line on the architectural changes that can be suitable in graph processing data centres.

- Hardware-software co-design of application-specific processing architectures. Integration of our hardware architecture with a software one, so that the computation is more energy-efficient. Testing which algorithms or what kind of them benefit from software or hardware architectures, and developing a way of computing them mixing both kinds of architectures.

- Implementation of an efficient dynamic re-partitioning system, so we can reassign the vertexes to other workers while the system is computing, to even the workload on the workers, and thus increase efficiency.

- Optimisation of the implemented architecture, rethinking some of the blocks so that we can achieve more parallelism.
• Study of how can we achieve a better energy performance in relation with the energy availability and the energy generation capability in the smart grid context.

• Evaluation of the memory usage metrics, so it can be compared to other implementations in that aspect.

• Study the integration with Operating Systems and other middleware.

This project is the starting point for the submission of a European project proposal led by IMDEA Networks. Even though this work is only a preliminary evaluation of the benefits of this approach, we believe this work opens a research line on efficient graph processing that could have a high impact on the community.
Bibliography


