EXPLORING VECTORISATION FOR PARALLEL BREADTH-FIRST SEARCH ON AN ADVANCED VECTOR PROCESSOR

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2017

By
Mireya Paredes López
School of Computer Science
Contents

Abstract 12
Declaration 13
Copyright 14
Dedication 15
Acknowledgements 16

1 Introduction 17
   1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
   1.2 A Glance at Graph Theory . . . . . . . . . . . . . . . . . . . . . . 19
      1.2.1 How Graphs are Represented by Computers . . . . . . . . 21
      1.2.2 Graph Algorithms . . . . . . . . . . . . . . . . . . . . . . . 23
   1.3 The Evolution of Parallel Computers . . . . . . . . . . . . . . . . 24
      1.3.1 The Rise of The Multicore Architecture . . . . . . . . . . . 25
      1.3.2 Challenges of Parallelism . . . . . . . . . . . . . . . . . . . 27
   1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   1.5 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   1.6 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2 Breadth-First Search Related Work 32
   2.1 Serial Breadth-First Search Algorithm . . . . . . . . . . . . . . . 33
   2.2 Layer Synchronous breadth-First Search . . . . . . . . . . . . . . 35
      2.2.1 Top-Down Breadth-First Search Algorithm . . . . . . . . . 36
      2.2.2 Parallel Top-Down Breadth-First Search Algorithm . . . . 37
   2.3 Parallel Architectures . . . . . . . . . . . . . . . . . . . . . . . . 40
      2.3.1 Massively Multithreaded Systems . . . . . . . . . . . . . . 40
3 The Intel Xeon Phi

3.1 Architecture .............................................. 52
3.2 A Shared Memory System ......................... 53
  3.2.1 Threads Synchronisation .................. 54
  3.2.2 Cache Memory ................................... 55
3.3 Vector Processing ................................. 56
3.4 Programming Model ............................... 58
  3.4.1 Multithreading ............................... 58
  3.4.2 Vectorisation .................................... 59
  3.4.3 Compilation ..................................... 61
3.5 Performance Counters ........................... 62
3.6 Summary ........................................... 63

4 The Experimental Development Framework ..... 64

4.1 The Graph500 Benchmark ....................... 64
  4.1.1 Graph Generator ............................. 65
  4.1.2 Graph 500 Experimental Suite ............. 69
4.2 Experimental Development Framework .......... 74
  4.2.1 Class Hierarchy Model ..................... 75
  4.2.2 Class Diagram .................................... 77
4.3 Summary ........................................... 78

5 Top-Down BFS Vectorisation .................. 80

5.1 Parallel Top-Down Breadth-First Search .... 81
5.2 Data Structures .................................. 82
5.3 The non-SIMD Breadth-First Search .......... 83
5.4 The SIMD Breadth-First Search ............... 85
  5.4.1 Description of the Algorithm ............. 87
  5.4.2 Xeon Phi Optimisations .................... 91
5.5 Which layers are vectorised? .................. 96
5.6 Experimental Setup ........................................ 97
  5.6.1 Top-Down Settings .................................. 97
  5.6.2 Thread Affinity Settings ............................ 97
5.7 Top-Down Vectorisation Analysis .......................... 98
5.8 Thread Affinity Analysis ................................ 99
5.9 Conclusions and Future Work .............................. 100
5.10 Summary .................................................. 100

6 The Hybrid BFS Vectorisation ................................. 102
  6.1 Bottom-Up Breadth-First Search ......................... 103
  6.2 Hybrid Breadth-First Search ............................ 103
    6.2.1 Online Heuristic .................................. 105
    6.2.2 Metrics ........................................... 107
    6.2.3 Switching Points .................................. 109
    6.2.4 Description of the Algorithm ...................... 112
  6.3 Bottom-Up Vectorisation ................................. 112
    6.3.1 Algorithm: Setting Multiple Parents ................ 113
    6.3.2 Which layers are suitable for vectorisation? ....... 118
  6.4 Experimental Setup ..................................... 123
    6.4.1 Bottom-Up Settings ................................ 124
    6.4.2 Hybrid Breadth-First Search Settings .............. 125
  6.5 Bottom-Up Analysis ..................................... 125
    6.5.1 Choosing a Starting Vertex ......................... 126
    6.5.2 PAPI Counters Analysis ............................ 129
    6.5.3 Performance Analysis ................................ 135
  6.6 Hybrid Breadth-First Search Analysis .................... 135
  6.7 Conclusions and Future Work ............................ 140
  6.8 Summary ................................................ 143

7 Conclusions and Future Work ................................ 144
  7.1 Summary ................................................ 144
  7.2 Conclusions ............................................. 146
    7.2.1 Top-Down Breadth-First Search Vectorisation ...... 146
    7.2.2 Hybrid Breadth-First Search Vectorisation ........ 147
  7.3 Future Work ............................................. 147
    7.3.1 Towards the Heterogeneous Breadth-First Search ... 147
List of Tables

1.1 Thesis structure. .............................................. 31

2.1 Previous studies related with the vectorisation of BFS algorithm on the Intel Xeon Phi. .......................... 46

2.2 Graph analysis packages and frameworks currently under active development. ..................................... 50

3.1 The Intel Xeon Phi features. ................................. 52

3.2 Development of SIMD instructions set developed by Intel. ........ 57

3.3 Experimental settings. ......................................... 61

3.4 PAPI events available for the Intel Xeon Phi. ............... 62

4.1 Graph 500 output quartiles for the BFS algorithm of time, number of edges and TEPS. ............................ 72

5.1 Traversed vertices per layer for a 1,048,575 million vertices graph, SCALE 20 and edgefactor 16. ............... 96

5.2 Performance SIMD version by setting thread affinity for a graph size of SCALE = 20 and edgefactor = 16. .......... 99

6.1 Traversed vertices per layer for a 1,048,575 million vertices graph, RMAT graph of SCALE 20 and edgefactor 16. ............................................ 107

6.2 Example of the transition between the top-down and bottom-up approaches of the hybrid BFS for a graph size SCALE = 18 and edgefactor = 16 using α = 14 and β = 24. ......................... 110

6.3 Example of the transition between the top-down and bottom-up approaches of the hybrid BFS for a graph size SCALE=18 and edgefactor = 16. ............................................ 111
6.4 Parameters used in the *online* heuristic: \(\alpha\) and \(\beta\). Developed by Beamer *et al.* [BAP12] and Gao *et al.* [GLZS14].

6.5 Average of the number of edges processed per visited vertex using the *bottom-up* BFS algorithm per layer. The graph size is \(SCALE = 18\) and the \(edgefactor = 16\).

6.6 Example of the execution of the hybrid BFS for a graph size \(SCALE = 18\), \(edgefactor = 16\) and starting node = 2005, showing the number of vertices and the number of non-visited vertices per layer.

6.7 Example of the execution of the hybrid BFS for a graph size SCALE=18, edgefactor = 16 and starting node = 2005, showing the fraction threshold set to 0.9 to choose between either the vectorised or the non-SIMD version of the *bottom-up* algorithm.

6.8 Example of the execution of the hybrid BFS for a graph size \(SCALE = 18\), \(edgefactor = 16\), varying the vectorisation threshold for the non-visited vertices. The number of GTEPS are the average out of 5 iterations.

6.9 Number of vectorised executions with the threshold set to 8. This example was executed with the bottom-up BFS using SIMD-range with threshold for the non-visited vertices set to 8. Graph size SCALE=18, edgefactor=16 for starting vertex 2005.

6.10 Experimental input parameters.

6.11 PAPI events for instructions counting and cache accesses.

6.12 Experimental input parameters.

6.13 Terminology for different versions of the BFS algorithm.

6.14 Results of the hybrid BFS with a “good” starting vertex \textbf{20025}, using \(\alpha = 1024\) and \(\beta = 64\) for a graph size of \(SCALE = 18\) and \(edgefactor = 16\).

6.15 Results of the hybrid BFS with a “bad” starting vertex \textbf{70108}, using \(\alpha = 1024\) and \(\beta = 64\) for a graph size of \(SCALE = 18\) and \(edgefactor = 16\).

6.16 Hybrid BFS execution for starting vertex 12, 119 using the *bottom-up* non-SIMD version. PAPI was used to get the hardware counters for cycles, instructions and CPI(cycles/instructions). \(SCALE = 18\) and \(edgefactor = 32\).
6.17 Hybrid BFS execution for starting vertex 12, 119 using the \textit{bottom-up vectorising per layer} (SIMD-layer). PAPI was used to get the hardware counters for cycles, instructions and CPI (cycles/instructions). \textit{SCALE} = 18 and \textit{edgefactor} = 32. .............. 132

6.18 Hybrid BFS execution for starting vertex 12, 119 using the \textit{bottom-up vectorising per range} (SIMD-range). PAPI to get the hardware counters for cycles, instructions and CPI (cycles/instructions). The graph size is \textit{SCALE} = 18 and \textit{edgefactor} = 32. .... 132

6.19 Hybrid BFS execution for starting vertex 12, 119 using the \textit{bottom-up \textit{non-SIMD version}}. PAPI was used to get the hardware counters for L1 and L2 cache misses and vector instructions. The graph size is \textit{SCALE} = 18 and \textit{edgefactor} = 32. .............. 134

6.20 Hybrid BFS execution for starting vertex 12, 119 using the \textit{bottom-up vectorising by layers} algorithm. PAPI was used to get the hardware counters for L1 and L2 cache misses and vector instructions. The graph size is \textit{SCALE} = 18 and \textit{edgefactor} = 32. .... 134

6.21 Hybrid BFS execution for starting vertex 12, 119 using the \textit{bottom-up vectorising by range} algorithm. PAPI was used to get the hardware counters for L1 and L2 cache misses and vector instructions. The graph size is \textit{SCALE} = 18 and \textit{edgefactor} = 32. .... 134

A.1 Example of the Graph 500 calculation of the harmonic mean for a graph size of \textit{SCALE} = 18 and \textit{edgefactor} = 16. .............. 153

A.2 Example of the Graph 500 calculation of the harmonic mean for a graph size of \textit{SCALE} = 18 and \textit{edgefactor} = 16. Continuation of Table A.1. .............. 154
List of Figures

1.1 General diagram of entities involved in the graph analysis of information. 20
1.2 Example of a social network for the terrorism attacks analysis. 21
1.3 Example of a simple graph. 21
1.4 An adjacency list representation. 22
1.5 An adjacency matrix data structure. 23
2.1 Example of the BFS traversal by layers. 35
3.1 The Intel Xeon Phi microarchitecture. 52
3.2 A basic shared memory system. 53
3.3 Examples of shared data between threads. 54
3.4 Example of a byte requested in a cache line. 55
3.5 Single Instruction Multiple Data (SIMD) processing. 56
4.1 Example of vertex degree distribution for a graph size of $scale = 10$ and $edge\_factor = 16$. 66
4.2 Examples of the adjacency matrices of the graphs $A(G_1)^{\otimes 3}$ and $A(G_1)^{\otimes 4}$, created with $G_1$ as graph initiator. 67
4.3 Edge distribution in the adjacency matrix based on $a$, $b$, $c$ and $d$ probabilities. 68
4.4 Graph 500 compressed sparse row (CSR) representation. 69
4.5 Sketch of the Graph 500 Experimental Suite. 70
4.6 Example of the BFS tree and its predecessor array for a graph of 8 vertices. 70
4.7 Boxplot graph representation. 73
4.8 Conceptual scheme of the integration of multiple BFS implementations with the Graph 500 Experimental Suite. 75
4.9 Class hierarchy model of the development framework. 76
4.10 Class diagram of the BFS graph kernel development. .......................... 78

5.1 Example of an array of integers (32 bits) represented by a bitmap array. .............................................. 82
5.2 Example of data benign race condition. ........................... 83
5.3 Example of the visited bitmap array race condition. .... 84
5.4 Example of vectorising the adjacency list exploration. .......... 88
5.5 Gather and scatter operations, to load non-contiguous visited array. 88
5.6 Illustration of the techniques utilised to cope with less-than-full-vector loops for the 64B data alignment of the Xeon Phi. ........... 92
5.7 BFS experimental optimisations results for a graph of SCALE 20 and edgefactor=16. .......................... 95
5.8 BFS non-SIMD and SIMD experimental results for SCALE values of 18, 19, 20 and edgefactor = 16. In Figure (c) the dashed line represents the best MTEPS reported in [GLZS14] with a maximum number of threads of 228. .......................... 101

6.1 Example of the bottom-up adjacency list exploration. .......... 104
6.2 Comparison between number of vertices to be explored in the layer and the number of non-visited vertices in the graph. Notice that to make comparable the two sets of numbers, the number non-visited vertices was divided by $\alpha = 1024$ value, the number of RMAT graph generated by Graph 500 using SCALE = 20 and edgefactor = 16. ........................................ 106
6.3 Sketch of the hybrid BFS algorithm proposed by Beamer et al. [BAP12]. ............................................. 108
6.4 State diagram to switch between top-down and bottom-up approaches in the hybrid BFS algorithm. ......................... 109
6.5 Example of gathering the adjacent vertices for the iteration number 1 of the search through the adjacency list for specific input vertices. 114
6.6 Boxplot of the third layer of the 64 bottom-up BFS executions for a graph size of SCALE = 18 and edgefactor = 16. ............. 117
6.7 Example of the bottom-up vectorisation process by processing 16 input vertices at a time. .............................. 120
6.8 Boxplot graph representing the variation in number of TEPS of the 64 BFS executions among 10 experimental iterations for each of the 64 random starting vertices. The implementation uses the non-SIMD version of the *bottom-up* algorithm on 228 threads. The graph size is $SCALE = 18$ and $edgefactor = 32$. ... 127

6.9 Boxplot graph representing the variation in number of TEPS of the fourth execution out of the 64 BFS executions with the starting vertices 12, 119 over 10 experiment iterations. The BFS implementation uses the non-SIMD version of the *bottom-up* algorithm on 228 threads. The graph size is $SCALE = 18$ and $edgefactor = 32$. 130

6.10 Boxplot graph for the three implementations: *non-SIMD*, *SIMD_layer* and *SIMD_range*. Each boxplot is a result of 64 iterations showing the minimum quartile, first quartile, median, third quartile and maximum of number of TEPS among 10 experimental iterations. The graph size is using $SCALE = 18$ and $edgefactor = 32$. ... 136

6.11 Hybrid BFS performance results comparing the *non-SIMD* versus the *SIMD-range* bottom-up version for different RMAT graph sizes, using $SCALE$ from 14 up to 20 and edgefactor of 16, 32 and 64, generated by *Graph 500* ... 138

6.12 Hybrid BFS performance results for the *SIMD-range* version and the state-of-the-art results presented by Gao et al. [GLZS14] for different graph sizes, using $SCALE$ from 14 up to 20 and edgefactor of 16, 32 and 64, generated by *Graph 500*. ... 141
Abstract

Exploring Vectorisation for Parallel Breadth-First Search on an Advanced Vector Processor
Mireya Paredes López
A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy, 2017

Modern applications generate a massive amount of data that is challenging to process or analyse. Graph algorithms have emerged as a solution for the analysis of this data because they can represent the entities participating in the generation of large scale datasets in terms of vertices and their relationships in terms of edges. Graph analysis algorithms are used for finding patterns within these relationships, aiming to extract information to be further analysed.

The breadth-first search (BFS) is one of the main graph search algorithms used for graph analysis and its optimisation has been widely researched using different parallel computers. However, the BFS parallelisation has been shown to be challenging because of its inherent characteristics, including irregular memory access patterns, data dependencies and workload imbalance, that limit its scalability.

This thesis investigates the optimisation of the BFS on the Xeon Phi, which is a modern parallel architecture provided with an advanced vector processor using a self-created development framework integrated with the Graph 500 benchmark. As a result, optimised parallel versions of two high-level algorithms for BFS were created using vectorisation, starting with the conventional top-down BFS algorithm and, building on this, leading to the hybrid BFS algorithm. The best implementations resulted in speedups of 1.37x and 1.33x, for a one million vertices graph, compared to the state-of-the-art, respectively. The hybrid BFS algorithm can be further used by other graph analysis algorithms and the lessons learned from vectorisation can be applied to other algorithms targeting the existing and future models of the Xeon Phi and other advanced vector architectures.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Dedication

To Valentina

This thesis is dedicated to my 8 year old beloved daughter, Valentina. She has been nothing but the greatest company of my life, who always has been keen to follow me. Together, we moved to a new country, with the help of my mum, full of uncertainty but also full of the curiosity about the unknown. To me, my PhD has been a continuous process of learning, which I feel very pleased to have shared with Valentina. We not only learnt English but also the tricks of living in an expensive, cold and rainy country, which help us to appreciate the sunshine. We met friends from all over the world including Greece, Cyprus, Spain, China and Russia, everybody telling us amusing stories about their cultures. Despite the natural downs of doing a PhD, having Valentina playing around was always the bright side that cheered me up. She looked very happy and that has been the biggest motivation to always keep trying. Although this experience has reached to its end, we shall keep learning.

Valentina, gracias por estar a mi lado, estoy muy orgullosa de ti. Te dejo esta frase, que resume lo que aprendí en mi Doctorado y que creo te podría servir a lo largo de tu vida.

“By understanding simple things,
you will discover new things beyond your imagination”.

Never stop learning.
Acknowledgements

I feel very lucky to have not only one but two excellent supervisors that guided my PhD studies, Mikel Luján and Graham Riley. The sharp vision of Mikel helped me to look forward in my research, giving it direction, whereas Graham patiently helped me to understand more about the challenges of parallel programming. Both have been key to the development of this thesis but also molding me as a scientist. Despite sometimes struggling with my presentations, the intense but assertive feedback of Mikel helped me to improve and that is something I really appreciate. Systematically, Graham taught me the importance of being systematic while running experiments by showing me the significance of carefully keep track of them. For these reasons, I really appreciate all the support and guidance from my supervisors. But also I would like to thank to the members of the APT group that helped me through their technical support and valuable feedback. I would like to thank to all my colleagues, also PhD students, for their company and support during my PhD. This thesis was funded thanks to the National Council for Science and Technology of Mexico (CONACyT).

I would like to thank my family for their love and all their support, especially my mum, who always has proactively supported my decisions. Also to my friends in Mexico and the ones that I met here in Manchester that help me to cope with the hard periods of my PhD. To Diana Alvarez, Garibaldi García and Athanasios Stratikopoulos, who were always very helpful and supportive friends during my PhD. I want to thank to my friend, but also my proofreader James Knight, who has always been keen to help me. Last but not least, to Luca Minciullo for all his support and patience during the last phase of my PhD.
Chapter 1

Introduction

This thesis is an investigation of parallel vectorisation techniques applied to graph analysis algorithms. Specifically, the goal is to identify practical methods for best applying vectorisation to optimise an important graph analysis algorithm, the Breadth-First Search (BFS), using the recent parallel architecture Intel Xeon Phi provided with advanced vector processors.

This chapter begins with a motivation in Section 1.1, describing why graph analysis algorithms have recently gained attention and the main approaches that have been applied for their development. Section 1.2 introduces the concept of graphs, including graph analysis algorithms. Section 1.3 presents the evolution of parallel computers over time, starting with the first vector processor to the recent manycore and heterogeneous systems. Section 1.4 summarises the contributions of this thesis, followed by its publications in Section 1.5. Finally, the structure of this thesis is presented in Section 1.6.

1.1 Motivation

While 68 years ago, in Manchester (U.K.), the first small scale experimental machine, nicknamed the “Baby” [Cop11], could execute a small program electronically stored in up to 256 bytes of memory, today the current challenge is to handle much larger datasets. Modern applications, including social networks, health-care records and computational sciences, generate large amounts of data [EJRB13]. As an example, Facebook, the popular social network, has 901 million users [WT13], with relationships among them. To represent this information in a computer using 64-bit data types, it would take over 2.4 TB of memory [EJRB13],
which is probably more than the creators of Baby could anticipate. However, the storage of large datasets is not the only problem, but also the complexity inherent to the analysis of this information, which is driving the development of technology beyond the current capabilities of computers.

Currently, big corporations, universities and organisations such as, Google, Facebook and MIT Lincoln Laboratory, are pushing forwards to address the challenges of dealing with large datasets [BS13]. One common approach has consisted of not only the recognition of the entities participating in the generation of the information, but also their relationships. By studying these interactions, different activities, events or patterns can be detected.

A natural way to abstract the entities and their relationships is by using graphs. In a graph, an entity can be represented as a vertex and a relationship as an edge. For example, an underground transportation network can be represented by a graph, where each station is a vertex and each connection between two stations is an edge in the graph. Trying to search for the route that would take minimum time from one station to another could be calculated by traversing the graph, adding up the time associated with each edge and finding the lowest result between all the traversed routes. Traversing a graph is the basic algorithm involved in the solution of more advanced graph algorithms for pattern detection. A very popular traversal algorithm is the BFS, which works as a building block for many graph analysis algorithms.

Recently, significant efforts have been dedicated to build software frameworks providing a set of core graph analysis algorithms, focusing on two factors: an analyst-friendly interface and performance. Firstly, the interface has been approached by using Domain Specific Languages (DSL) [EJRB13], which are high-level languages that allow programmers (in this case analysts) to describe algorithms according to a specific domain (graph analysis) in a way that they feel familiar and comfortable. Green-Marl [HCSO12] is a good example of a DSL for graph analysis. Secondly, the improvement in terms of performance has been based on the optimisation of graph algorithms using emerging parallel architectures. However, graphs generated by large datasets are often unstructured and highly irregular, which are key characteristics for their limited scalability and performance. By using a graph DSL, the details of the parallelisation of graph analysis algorithms, which can be using different hardware architectures, are hidden from the analyst perspective. In addition, graph DSL allow graph
algorithms optimisations to be developed in an independent way. Due to the challenges presented in the parallelisation of graph algorithms, the Graph 500 benchmark [MWBA10] has emerged as an environment to evaluate and compare the performance of graph analysis algorithms over different parallel architectures. Graph 500 provide a set of key graph algorithms, including the BFS.

The research presented in this thesis is focused on the investigation of the optimisation of an important graph algorithm for analysis of information, the BFS, targeting the manycore architecture with vector advanced processors, the Intel Xeon Phi. The results are evaluated in the context of the Graph 500 benchmark and compared with the previous best results in the literature. Figure 1.1 illustrates the multiple entities involved in the development of graph analysis algorithms, highlighting the scope of this thesis.

This chapter addresses the following questions:

1. Why are graph algorithms important?

2. What is the role of parallel computers and vector processing?

\subsection*{1.2 A Glance at Graph Theory}

Pictorially, a graph is represented by a collection of points connected to each other by lines. Graphs are commonly used to abstract complex problems. For instance, network transportation routing or interactions among people in public social networks. Figure 1.2 shows an example of a graph of a social network.

Formally, a graph $G$ is a finite set of vertices $V$ and edges $E$; each edge has two vertices as ends, associating them [Eve11]. The total number of vertices in a graph is denoted by $|V|$ and the total number of edges by $|E|$. A loop is when an edge connects a vertex to itself. Two edges having the same pair of endpoints are parallel. The degree of a vertex is the number of edges related to that vertex. A simple graph is one that contains neither loops nor parallel edges. Figure 1.3 shows an example of a simple graph with four vertices and five edges.

A graph $G(V, E)$ is directed if each edge in the graph has an associated direction. If the edges do not have associated direction then it is an undirected graph. A path is a sequence of edges that connects one vertex to another. Paths are very useful in search algorithms because they can be studied to find basic structural
Figure 1.1: General diagram of entities involved in the graph analysis of information.

information related to the graph. A tree is a undirected graph, where any two vertices are connected by only one path. The distance between two vertices is the number of edges in the shortest path between them. The diameter of a graph is the greatest distance between any pair of vertices. If a graph has associated weights, usually a number, with each edge is called a weighted graph. A graph is connected if every pair of vertices is connected by a path. A connected graph is called highly irregular if given any vertex, its neighbours have different degree. Most of the large scale datasets represented by a graph are composed by vertices with different degrees. The graphs studied in this thesis are highly irregular and their characteristics are presented in Chapter 4.
Figure 1.2: Example of a social network for the terrorism attacks analysis. Source image [Org13].

Figure 1.3: Example of a simple graph.

1.2.1 How Graphs are Represented by Computers

Graphs are often represented by dynamically allocated memory data structures. There are two well-known data structures: adjacency list for sparse graphs (those
CHAPTER 1. INTRODUCTION

for which $|E|$ is significantly less than the $|V|$), and adjacency matrix for dense graphs (those for which $|E|$ is close to the $|V|$). A graph is static when the data input has a fixed sequence of nodes and edges. An example of a static graph results from analysis of the underground transportation network, where the stations can be represented as vertices and the rails as edges. Neither stations nor rails are updated frequently. On the other hand, if the structure of the graph changes with time, the graph is dynamic. An example of a dynamic graph are streaming graphs [EJRBB13].

In the following sections, the typical data structures used to represent for static graphs, adjacency list and adjacency matrix, are described.

**Adjacency List**

An adjacency list is a data structure that associates a list of the vertices connected to each of the vertices in the graph. Figure 1.4 illustrates a small graph (5 vertices) stored with an adjacency list data structure. In this thesis, a variation of an adjacency list data structure, the compressed sparse row (CSR), is used further, which is introduced in Chapter 4.1.1.

![Figure 1.4: An adjacency list representation. Source image [CLRS09].](image)

**Adjacency Matrix**

An adjacency matrix consists of a matrix of size $|V| \times |V|$. A generic element $M_{i,j}$ is set to 1 if there is an edge between vertex $i$ and vertex $j$ and is set to 0 otherwise (Equation 1.1). Sparse graphs, which have significantly fewer edges compared to the number of vertices, result in many zero entries in the adjacency matrix, which produces waste of memory space and can lead to low performance. Figure 1.5
CHAPTER 1. INTRODUCTION

shows an example of the mapping of a graph to a matrix representation.

\[
M_{i,j} = \begin{cases} 
1 & \text{if } (i, j) \in E \\
0 & \text{otherwise} 
\end{cases}
\] (1.1)

Figure 1.5: An adjacency matrix data structure.

1.2.2 Graph Algorithms

Graph algorithms are procedures that systematically solve problems related to graphs [CLRS09]. For example, solving search problems can reveal information about the structure of the graph. For this reason, graph search algorithms are the basis for various other algorithms used to extract structural information.

Common problems involved in large graph analysis are finding paths, clusters, partitions, matching, patterns and ordering [BS13]. To solve those problems several advanced graph algorithms are used, falling in the following categories.

- **Traversal algorithms:** These algorithms refer to the order in which the vertices of the graph are visited. The most simple and popular traversal algorithm for graph analysis is the BFS [EJRB13]. Given a source vertex \( s \) of a graph \( G = (V, E) \), the BFS algorithm consists of visiting systematically all the edges of \( G \) to trace all the vertices reachable from \( s \), resulting in a BFS tree for the graph. The BFS algorithm is introduced in more detail in Chapter 2.

- **Shortest path algorithms:** As the name indicates, these algorithms find the shortest possible path from a given source vertex \( s \) to a destination
vertex $t$. Shortest path usually means the path that obtains the minimum weight among all the possible paths.

- **Max flow algorithms:** Given a weighted graph, these algorithms determine by trial and error (going forward and backwards), the path connecting two vertices, $s$ and $t$, having the maximum weight.

- **Spanning tree algorithms:** A *spanning tree* is a tree that reaches all the vertices of a graph. The algorithm that produces the *minimum spanning tree* is the one whose weight is no larger than the weight of any other spanning tree.

- **Topological analysis:** Based on the characteristics in the topology of the graph such as: *vertex degree distribution*, *centrality* and *community structure*, where *centrality* is an indicator of the most important vertices in the graph and *community* is a set of vertices having interaction between them collaborating to the same purpose. Valuable insights related with the structure and function of the interacting data entities can be obtained [BM08].

The core of graph analysis algorithms are also called *graph kernels* and they are built based on basic traversal graph algorithms such as the BFS (BFS).

### 1.3 The Evolution of Parallel Computers

The history of parallel computers goes back to the fifties with the first electronic computers [HP11]. In 1964, the *Iliiac IV* was the first supercomputer composed of large-scale multiprocessors with up to 256 processors dedicated to work on large datasets. The Iliiac IV, later became categorised in Flynn’s taxonomy [Fly72] as a *single instruction multiple data* (SIMD) machine. Vector processors are the main class of SIMD machines. In 1966, Michael Flynn classified all computers with a simple but useful model that still remains applicable today [HP11]. This classification is divided in two main categories based on the number of concurrent instructions streams and the data streams available in the computer architecture: *instructions streams* and *data streams*, where they can be either *single* or *multiple* [Fly72].
1. Single Instruction, single Data (SISD) - This is a machine that can execute one instruction over a single data, which is the uniprocessor.

2. Single Instruction, multiple Data (SIMD) - These machines can execute one instruction over multiple data. This category stands for parallel computers such as the Illiac IV, which was composed of a large array of processors, each of them with its own memory, capable of executing the same instruction over multiple data at the same time. Nowadays, advanced vector processors are based on the early SIMD machines.

3. Multiple Instruction, single Data (MISD) - The rational behind these machines is having multiple instructions executed over one single data. Up to date, there are no commercial machines of this type.

4. Multiple instruction, multiple Data (MIMD) - These machines execute multiple instructions over multiple data. MIMD machines contain several processors where each one fetches its own instructions and executes them over its own data. Usually, these processors are off-the-shelf microprocessors, which comprise the central processing unit (CPU) functions on a single integrated circuit.

Despite the Illiac IV being a breakthrough for the development of supercomputers, the project was not successful because it did not fulfill the initial expectations in terms of cost and performance [HP11]. Driven by the market, the semiconductor industry pushed forwards the development of the technology of microprocessors, and MIMD machines became popular, mainly because of their flexibility as general-purpose computers that could be built on off-the-shelf microprocessors. These machines are flexible and affordable, as opposed to supercomputers, which are provided with a high degree of parallelism but built based on specialised processors that make them rare and expensive.

1.3.1 The Rise of The Multicore Architecture

In 1965, Gordon Moore, a founder of the Intel semiconductor company, predicted that the number of transistors in an integrated circuit will double roughly every 18 to 24 months [SL13]. This observation has driven the evolution of computer architecture in the three decades following with a scaling of the performance and reduction in the cost of their construction. Around ten years ago, there was a shift
to the multicore architectures, which typically consist of a single chip containing several single processing units (cores) \cite{EBA11}. Since then, the trend has been to increase the number of cores along with the number of transistors to keep the scaling proportion in performance.

Multicore architectures are generally classified depending on the number of cores, memory organisation and interconnect strategy \cite{HP11}. In terms of memory organisation there are two main models: \textit{shared memory} and \textit{distributed memory}. In a \textit{shared memory} model, there is a centralised memory that all the cores share using, for example, a bus for the interconnection between the cores and the memory. Cores share information by accessing to the same memory locations. In contrast to \textit{shared memory}, in the \textit{distributed memory} model, memory is physically distributed among the cores rather than centralised. All cores access a global interconnection and they exchange information by passing messages to provide a shared memory view. Multicore systems can be combined into larger systems, e.g., clusters or supercomputers such as the Hopper \cite{BM11}. These systems can be programmed by using Message Passing interfaces such as MPI library \cite{LHSS09} to exchange data and they can be referred to as \textit{distributed memory MP systems}.

Regarding the number of cores in a multicore architecture, a new classification, called \textit{manycore}, has recently been introduced. Manycores are usually single processors which have a high degree of parallelism and they have a large number of simple processors. The term \textit{manycore} can be used relative to the number of single processors in the architecture, in this thesis the term manycore will be used to refer to multicore architectures with more than 16 single processors. Not long ago, in 2012, Intel announced the Intel Xeon Phi, which is not only a manycore architecture with up to 60 cores but also each core contains its own advanced vector processor based on the early vector processors. Compared to a program running on a sequential processor a vector instruction replaces many sequential instructions making programming simpler and leading to the implementation of more efficient vector-oriented memory. For this reason, vector processors give higher performance than scalar processors because of the simple memory access patterns implicit in vector operations \cite{HP11}. Apart of the vector unit, each core in the Xeon Phi can support up to four hardware threads. The current version of the Xeon Phi (known as Knight’s Corner) is the focus of the work in this thesis and is discussed in detail in Chapter 3.
Furthermore, a new generation of heterogeneous systems is coming. A heterogeneous multicore computer system could be defined as an example of “computers with multiple processors that have different capabilities” [AMD08]. This means that instead of having the traditional multicore system, heterogeneous computers also contain several accelerators. An accelerator is a specialised hardware used to improve the performance of specific processing tasks. An example of an accelerator is a general-purpose GPU (Graphics Processing Unit), which processes non-specialised computations conducted by the CPU [OLG+07].

The development of the BFS algorithm has been tackled using several of the parallel architectures introduced in this section, including distributed memory, multicore and heterogeneous systems. Chapter 2 introduces different approaches conducted over the parallelisation of the BFS algorithm.

1.3.2 Challenges of Parallelism

Despite the fact that often parallel computers are a synonym of enhancing programs’ performance, it has been proven that is not always the case. According to Amdahl’s law [Amd67], the efficiency of a parallel computer relies on the overall utilisation of the $N$ processors. For instance, for a program that is composed of a sequential part and a parallel part that can be executed by $N$ processors, where each part takes 50% of the time when executed on a single core, then the benefit of parallelising this program will be determined only by the parallel speedup and the rest will remain the same. Formally, the speedup of a parallel program can be expressed by Equation 1.2

$$\text{speedup} = \frac{1}{(1 - P) + \frac{P}{N}},$$

(1.2)

where $P$ refers to the parallel fraction of the program and $(1 - P)$ is the sequential fraction, $N$ is the total number of processors and $\frac{P}{N}$ is the parallel part executed by the $N$ processors.

Besides the challenge of having insufficient parallelism in a program, there is another challenge related with the cost of the communication spent on data movement and synchronisation in terms of time. This communication can happen between processors but also between the processor and the memory. Memory
bandwidth and latency are two concepts related to the movement of data. Memory bandwidth refers to the rate in which data can be read or stored into memory by a processor. Memory latency refers to the waiting time during the transmission of data. The investigation conducted in this thesis aims at the optimisation of the BFS algorithm to achieve better speedup than previous related work exploiting the two levels of parallelism provided by the Xeon Phi, multithreading and vectorisation, while using data structures and other techniques that help to reduce memory latency. The parallelisation of the BFS algorithm is described in Chapters 5 and 6.

1.4 Contributions

As a result of the exploration of efficient algorithms for the BFS algorithm exploiting the parallel architecture of the Xeon Phi, including its vector unit with SIMD programming support, two paths were taken. Firstly, the conventional parallel top-down BFS algorithm was studied to efficiently apply vectorisation. The complete description of this study and the analysis of the results are introduced in Chapter 5. Secondly, the state-of-the-art hybrid BFS algorithm was also investigated. The description, analysis and results of this study are presented in Chapter 6. Both studies lead to the following contributions:

- A new Experimental Development Framework to facilitate not only the development of new implementations of graph kernels but also to support their execution across different architectures. Using this framework, introduced in Chapter 4, the development and experiments of the studies in Chapter 5 and Chapter 6 were undertaken.

- An implementation of the parallel top-down BFS algorithm exploiting vectorisation is presented in Chapter 5. The results of this investigation are better in terms of performance in comparison with the state-of-the-art in the literature.

- An investigation of the state-of-the-art hybrid BFS algorithm leading to an implementation exploiting vectorisation. Compared with the state-of-the-art hybrid BFS implementation exploiting vectorisation in the literature, the implementation described in Chapter 6 has higher performance.
• A methodical evaluation and analysis of the vector utilisation of the bottom-up approach of the hybrid BFS algorithm is provided in Chapter 6. This analysis consists of the comparison of three bottom-up implementations. The first implementation does not exploit any vector instruction and acts as a basis for comparison (this version is called no-simd). The other two implementations use different approaches to exploit vector instructions and are named simd-layer and simd-range.

1.5 Publications

The material presented in Chapter 5 has been published in the following conference.


1.6 Thesis Structure

The next chapters are organised as follows:

Chapter 2 starts by introducing the serial BFS algorithm, followed by the layer synchronous BFS algorithm and its parallelisation. Multiple parallel architectures have been used to solve the BFS such as: massively multiprocessing systems, distributed memory systems, multicore shared memory systems and recent heterogeneous architectures. Moreover, the parallel frameworks used for graph analysis are introduced. This chapter also compares the work that has been done in the literature and emphasizes the ones that are more related with the main research line of this thesis, which is multicore shared memory systems. Finally, a critical summary of the previous studies approaching vectorisation to solve graph algorithms using the Intel Xeon Phi is presented.

Chapter 3 introduces a detailed description of the Xeon Phi architecture. Furthermore, since this architecture is a shared-memory system, a model of this kind is introduced to explain related concepts such as multithreading, race conditions, among others that will be referred further in Chapters 5 and 6.

Chapter 4 presents the Graph 500 benchmark including its graph generator, the experimental setup and the output performance information provided. Also,
CHAPTER 1. INTRODUCTION

This chapter introduces the design of the Experimental Development Framework (EDF) as the mechanism for integrating the Graph 500 experimental suite into the development of the new BFS developments. This framework was used for the development of the BFS implementations in Chapter 5 and 6.

Chapter 5 reviews how to vectorise the top-down BFS, starting by introducing the parallel top-down BFS algorithm. This is followed by a full description of the methodology used for vectorisation. To investigate the benefit of vectorisation, an experimental comparison between the parallel top-down algorithm without using vectorisation and the top-down algorithm using vectorisation was undertaken. The results are compared against the state-of-the-art implementation of the top-down BFS using vectorisation in the literature.

Chapter 6 presents how to combine both the top-down and the bottom-up BFS algorithms resulting in a new hybrid BFS algorithm. Firstly, the bottom-up BFS algorithm is defined and then the hybrid BFS algorithm is introduced by describing the mechanism to switch either to the top-down or the bottom-up algorithm. Additionally, the vectorisation of the bottom-up BFS algorithm is systematically analysed with the help of two different implementations using SIMD vectorisation at different levels. Finally, the results of applying vectorisation in the hybrid BFS algorithm are compared with the state-of-the-art.

Chapter 7 summarises the conclusions and the specific future work identified in the contributions Chapters 5 and 6. Additionally, the general insights from the global perspective of the work and the suggestions of some possible directions for future research are given.

Appendix A presents an example of the Graph 500 output for the BFS graph kernel after the execution of experiments.

Appendix B contains the source code of the vectorisation of the top-down BFS algorithm.

Appendix C contains the source code of the vectorisation of the bottom-up BFS algorithm.

Appendix D contains the list of the vector intrinsic functions used in this thesis.

Appendix E contains the list of the acronyms used in this thesis.

Finally, Table 1.1 shows the overview of the structure of this thesis.
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>Introduction, Motivation, Contributions, Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 2</td>
<td>BFS Related Work (background and literature review)</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>Intel Xeon Phi Architecture (background)</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>Experimental development framework (methodology)</td>
</tr>
<tr>
<td>CHAPTER 5</td>
<td>Top-Down BFS vectorisation (results and conclusions)</td>
</tr>
<tr>
<td>CHAPTER 6</td>
<td>Hybrid BFS vectorisation (results and conclusions)</td>
</tr>
<tr>
<td>CHAPTER 7</td>
<td>Conclusions and future work</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>Harmonic mean example</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>Top-Down BFS vectorisation source code</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>Bottom-Up BFS vectorisation source code</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>SIMD Intrinsic functions</td>
</tr>
<tr>
<td>APPENDIX E</td>
<td>Acronyms</td>
</tr>
</tbody>
</table>

Table 1.1: Thesis structure.
Chapter 2

Breadth-First Search Related Work

The BFS is a building block of graph analysis algorithms including the shortest path, betweenness centrality, among others algorithms [EJRB13]. Due to its importance, the parallelisation of the BFS has been a popular case study of several investigations.

This chapter introduces the definition of the BFS algorithm. Starting with the classic serial algorithm in Section 2.1, following by the conventional layer-synchronous BFS in Section 2.2. The layer-synchronous BFS algorithm can be implemented by using two lists that contains the vertices waiting to be processed. The two list implementation for the layer-synchronous BFS algorithm is described in Section 2.2.1 and it is used as basis for the parallel version of the BFS, which is a multithreaded list-based implementation described in Section 2.2.2.

The parallelisation of the BFS is challenging because of the irregular data access patterns, data dependency and workload imbalance issues that it presents, explained in Section 2.2.2. For this reason, different studies have been conducted to optimise the BFS algorithm in terms of execution time. The key optimisations have focused towards the improvement of multithread workload balance, exploitation of memory locality and the reduction of the usage of expensive atomic operations [LGHB07]. This chapter introduces the literature review in which the BFS has been implemented by using different parallel architectures. The literature is classified by the nature of the parallel architecture used; there are four main categories: massively multithreaded, distributed memory, multicore systems and recent heterogeneous architectures. The studies conducted in these categories
are introduced in Section 2.3 and a compendium of the ones using vectorisation is presented in Section 2.3.3. Moreover, parallel available frameworks for graph analysis are presented in Section 2.4. Finally, a summary of this chapter is given in Section 2.5.

2.1 Serial Breadth-First Search Algorithm

The BFS algorithm consists of systematically traversing all the vertices in a graph, starting from a starting vertex [CLRS09]. To identify the vertices that have been processed during the traversal, they are marked as visited. Initially, all the vertices are marked as non-visited. The process starts by discovering all the adjacent vertices associated with the starting vertex. This association is based on the list of edges that links the starting vertex to the adjacent vertices. Every discovered vertex that is found to be non-visited is then marked as visited and is stored in an ordered list to be further processed. A vertex is taken from that list and proceeds to discover its adjacent vertices. This process is repeated until there are no more vertices in the list to explore. The graph traversal constructs an output BFS and its root is the starting vertex. During the traversal, every time a vertex is marked as visited, it is also set as child of the vertex being processed. In addition, not required by the BFS algorithm but commonly used in the literature, this parent-child relation is stored in a list called the predecessor list.

Algorithm 1 shows the pseudocode of a variation of the classical serial BFS using a first-in first-out (FIFO) queue \( Q \) as an auxiliary structure [CLRS09].

The input parameters of the algorithm are, the graph \( G \) and the starting vertex \( s \). First of all, there are two lists, \( \text{vis} \) and \( \text{P} \) (stands for predecessor list), used as auxiliaries in the algorithm. The first list, \( \text{vis} \), is an integer array that is used to mark the visited vertices, where each index in the array represents each vertex in the graph. The second list, \( \text{P} \), is also an integer array that helps to build the predecessor list, which is delivered as the output of the algorithm. The elements of both arrays, \( \text{vis} \) and \( \text{P} \), are initialised to integer values. Each element of the \( \text{vis} \) array is initialized to an integer corresponding to \text{NOVISITED} (zero value) to identify that the vertex has not been visited. The \( \text{P} \) array is initialised to a -1 value, just as a way to differentiate between the vertices whose parents have been set from the ones that have not been. Similarly, \( Q \) is initialised as empty and the root vertex \( s \) is set. By convention, the starting
vertex is also set in the predecessor list as parent of its own parent. Then, the exploration starts by iterating through the vertices contained in the queue. Each adjacency of each of the vertices, identified as \( \text{Adj} \), in the queue is processed. If any adjacent vertex was found to be non-visited, it is set as visited and becomes child of the vertex being explored. The result of the algorithm is the predecessor list created by the \( P \) array.

**Algorithm 1  Serial BFS algorithm.**

1: procedure BFS\((G, s)\)
2: for \( u \in V(G) - \{s\} \) do \( \triangleright \) Initialising the visited and the predecessor list.
3: \( \text{vis}[u] = \text{NOVISITED} \)
4: \( P[u] = \text{−1} \) \( \triangleright \) Signifies no predecessor for the \( u^{th} \) element of \( V(G) \).
5: end for
6: \( Q \leftarrow \emptyset \) \( \triangleright \) The queue is set as empty.
7: Enqueue\((Q, s)\) \( \triangleright \) Enter vertex \( s \) into queue.
8: \( \text{vis}[s] = \text{VISITED} \)
9: \( P[s] = s \) \( \triangleright \) Setting the root of the predecessor list, \( s \) as predecessor of itself.
10: while \( Q \neq \emptyset \) do \( \triangleright \) While the queue is not empty.
11: \( u \leftarrow \text{Dequeue}(Q) \) \( \triangleright \) Getting the first element of the queue.
12: for \( v \in \text{Adj}[u] \) do \( \triangleright \) Exploring the adjacency list of vertex \( u \).
13: if \( \text{vis}[v] = \text{NOVISITED} \) then \( \triangleright \) Found a non-visited vertex.
14: \( \text{vis}[v] = \text{VISITED} \)
15: Enqueue\((Q, v)\) \( \triangleright \) Enter vertex \( s \) into queue.
16: \( P[v] = u \) \( \triangleright \) Setting \( u \) as a parent of vertex \( v \).
17: end if
18: end for
19: end while
20: end procedure

The computational complexity of Algorithm 1 is related to the number of vertices set in the queue during execution of the algorithm, which is the total number of vertices \( |V| \) in the graph, and also to the number of edges associated with the explored vertex, or in other words the length of the adjacency list. The sum of all the lengths of each adjacency lists is \( |E| \) and because all edges are processed, the total number of computations spent during the scanning of the adjacency list is \( O(E) \). Thus the computational complexity of the serial BFS algorithm is \( O(V + E) \) [CLRS09]. This computational complexity is taken as baseline and further optimisations of the BFS algorithm during its parallelisation aim to improve its performance.
2.2 Layer Synchronous Breadth-First Search

Given the definition of the serial BFS, the conventional way to traverse the graph is by layers. A layer consists of a set of all vertices with the same distance from the source vertex. Processing vertices by layers allow them to be explored in any order as long as they are in the same layer, key feature for further parallelisation. However, each layer has to be processed in sequence; that is all vertices with distance \( k \), (layer \( L_k \)) are processed before those in layer \( L_{k+1} \), which is the reason why the algorithm is named \textit{layer-synchronous}.

Figure 2.1 shows an example of the traversal of the BFS algorithm requiring three layers. The exploration of the graph starts from the starting vertex 1 and reaches all the vertices in the three layers illustrated by a, b and c in the right side of the drawing. There are many equally valid BFS trees, but only one is required. Dotted lines represent edges linked with already explored vertices.

![Figure 2.1](image)

The layer-synchronous BFS algorithm can be described using queues per layer but in the following section, the algorithm is described using lists, as opposed to queues, to facilitate the introduction of the parallel algorithm further in Section 2.2.2.
2.2.1 Top-Down Breadth-First Search Algorithm

By processing the graph by layers it is possible to explore independently the vertices in the layer, which is useful for further parallelisation of the algorithm described in Section 2.2.2.

The implementation of the serial BFS algorithm to process vertices by layers is known as the conventional top-down BFS. This algorithm uses two lists to set up the concept of a layer. The first list contains all the vertices to be processed in the current layer and it will be called input list or frontier. The second list is the output list, but which for consistency with the literature will also be sometimes referred to as the output queue, holds a sequence of vertices that after processing the layer will be swapped with the frontier to be processed in the next layer. When a vertex has been processed it is marked as visited, otherwise, it remains non-visited. Each vertex has an associated set of adjacent vertices to which it is connected by an edge, known as an adjacency list. Only the vertices that were found to be non-visited vertices are put into the output queue. The result of the algorithm is a BFS tree represented by a list of the predecessors or parents ($P$) of the traversed vertices.

Algorithm 2 shows the pseudocode of the serial top-down BFS algorithm. Firstly, the algorithm receives as input the graph denoted by $G$ and the starting vertex by $s$. $G$ is assumed to be represented by an adjacency list, where $\text{Adj}(u)$ refers to all vertices adjacent to $u$. Secondly, the pseudocode uses four data structures: input list ($in$), output list ($out$), visited list ($visited$), and $P$, predecessor list; all data structures are initialised at the beginning. The visited array is used to mark vertices as visited during the exploration process. Initially all the vertices are set as non-visited. The predecessor array is used to store the output BFS tree and is initialised with very large values. In practice, this value can be an integer bigger than the number of vertices such as $2^{31}$ the maximum value in a 4-byte integer. The exploration starts when the input list $in$ has at least one element (line 7). Thus, the starting vertex $s$ is first placed in the input list, visited array and in the predecessor array set as its own parent (lines 4-6).

In lines 7 to 17, every single vertex $u$ in the input list $in$ is explored, while it is not empty. This exploration consists of checking each adjacent vertex $v$ of $u$ that has not been visited. If it is the case, then they are put into the output list $out$, marked as visited and the parent for the vertex $v$ in the $P$ array is set to $u$. By checking for the non-visited vertices first, some extra work is avoided by not
putting previously processed vertices in the list \([\text{APPB10}]\). In line 16, the input and the output lists are swapped and the output list is cleared. The algorithm ends when all vertices reachable from the input vertex have been marked as visited (so the input list \(\text{in}\) for the next layer is empty). The output BFS tree is the predecessors list \(P\), which contains a list of the parents of each vertex found during the graph traversal.

### Algorithm 2 Serial Top-Down BFS\((G, s)\)

**Initialise:** \(\text{in.init()} \quad \text{out.init()} \quad \text{vis.init()}\)

1. for vertex \(u \in V(G) - \{s\}\) do
   - \(P[u] \leftarrow 2^{31}\) \(\triangleright\) Setting the maximum value of a 4-byte integer.
2. end for
3. \(\text{in.add}(s)\) \(\triangleright\) Adding the starting vertex \(s\) to the input list.
4. \(\text{vis.Set}(s)\) \(\triangleright\) The starting vertex \(s\) is set as visited.
5. \(P[s] = s\) \(\triangleright\) Setting the root of the predecessor list, \(s\) as predecessor of itself.
6. while \(\text{in} \neq \emptyset\) do \(\triangleright\) while \(\text{in}\) is not empty.
7. for \(u \in \text{in}\) do
8. for \(v \in \text{Adj}[u]\) do
9. if \(\text{vis.Test}(v) = 0\) then
   - \(\text{vis.Set}(v)\) \(\triangleright\) Setting the vertex \(v\) to visited.
10. \(\text{out.add}(v)\) \(\triangleright\) Setting the vertex \(v\) to the output queue.
11. \(P[v] = u\) \(\triangleright\) Setting \(u\) as a parent of vertex \(v\).
12. end if
13. end for
14. end for
15. end while
16. \(\text{swap}(\text{in}, \text{out})\) \(\triangleright\) Clearing the output queue out.
17. \(\text{out} \leftarrow \emptyset\)

### 2.2.2 Parallel Top-Down Breadth-First Search Algorithm

The naive parallelisation of the serial top-down BFS consists of exploiting two levels of parallelism. The first is a relatively coarse-grain level in the outer loop for processing the input list (line 8 in Algorithm 3); the second, finer-grain level, is in the inner loop where the adjacency list is explored (line 9). All threads are synchronised every layer by an implicit barrier at the end of the outer loop.
Algorithm 3 shows the parallel BFS version by augmenting Algorithm 2 with parallel for loops, indicating where the race conditions among multiple threads take place. The major changes needed to parallelise the algorithm are in the initialisation of the lists, the visited and the predecessor array data structures because multiple threads can do the work independently, and in the outer and inner loops, lines 8 and 9, where all the vertices are explored in parallel.

Algorithm 3 Parallel Top-Down BFS($G, s$)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for parallel vertex $u \in V(G) \setminus {s}$ do</td>
<td>Multi-threaded parallelism</td>
</tr>
<tr>
<td>2</td>
<td>$P[u] \leftarrow 2^{31}$</td>
<td>Setting the maximum value of a 4-byte integer.</td>
</tr>
<tr>
<td>3</td>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>in.add($s$)</td>
<td>Adding the starting vertex $s$ to the input list.</td>
</tr>
<tr>
<td>5</td>
<td>vis.Set($s$)</td>
<td>The starting vertex $s$ is set as visited.</td>
</tr>
<tr>
<td>6</td>
<td>$P[s] = s$</td>
<td>Setting the root of the predecessor list, $s$ as predecessor of itself.</td>
</tr>
<tr>
<td>7</td>
<td>while $in \neq \emptyset$ do</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>for parallel $u \in in$ do</td>
<td>Multi-threaded parallelism</td>
</tr>
<tr>
<td>9</td>
<td>for $v \in Adj[u]$ do</td>
<td>Exploring the adjacency list of vertex $u$.</td>
</tr>
<tr>
<td>10</td>
<td>if vis.Test($v$) = 0 then</td>
<td>If vertex $v$ is non-visited</td>
</tr>
<tr>
<td>11</td>
<td>$vis.Set(v)$</td>
<td>Race conditions</td>
</tr>
<tr>
<td>12</td>
<td>out.add($v$)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>$P[v] = u$</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>swap($in, out$)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>$out \leftarrow \emptyset$</td>
<td>Clearing the output queue out.</td>
</tr>
<tr>
<td>19</td>
<td>end while</td>
<td></td>
</tr>
</tbody>
</table>

The Challenges

This parallel version presents multiple challenges that potentially limit its scalability and performance, ending in an asymptotic complexity of work identical to the serial algorithm [BM11]. First, workload imbalance may be present in line 8 because while processing vertices $u$, the number of vertices in their respective adjacency list can vary. This can lead to having some threads busy while others are not doing much work. Second, irregular data access pattern can be detected.
in line 9, where the vertices in the adjacency list of the vertex being processed are tested to verify if they have been previously visited. Essentially, this random access implies gathering data from memory that might be located in different physical locations with different access times, which does not benefit from data locality. Third, data dependency is presented from lines 10 to 12, where multiple threads could try to update the same vertices at the same time. This problem could lead into some race conditions and the lack of correctness in the output BFS tree. Specifically the race conditions presented in the parallel BFS are benign and so they do not affect the correctness of the output BFS tree [LS10]. However, benign race conditions add overhead by making redundant work. In a shared-memory system, this problem can be mitigated by using atomic operations which can be expensive. Due to the complexity of the problems inherent in the BFS parallelisation, it has been amply studied. According to [BM11] the development of the BFS has been oriented towards the optimisation of the level-synchronous algorithm but considering the features of different parallel architectures. The main three optimisations applied for the BFS are focused to cope with the following challenges:

**Workload balance** When the work is divided among multiple threads, usually the aim is that the workload is equal. However, in the BFS algorithm the processing time of exploring a vertex cannot be known a priori, mainly because the number of edges of each vertex (degree) varies.

**Improving locality** The problem of the BFS is that when a vertex is being explored, the adjacency list can be holding vertices that are physically stored in different segments of memory (non-contiguous). By having these random accesses memory latency become a problem making a negative impact on performance. To overcome this problem, new ways of organising data for irregular data access patterns need to be investigated.

**Atomic operations** These operations are presented usually applied to avoid race conditions between multiple threads. Usually, they are expensive because it causes a bottleneck when multiple threads try to update the same data. Despite the conventional parallel BFS presents a data race condition, it is benign, which implies that atomic operations are not needed to maintain the correctness of the algorithm but the impact lies on having
CHAPTER 2. BREADTH-FIRST SEARCH RELATED WORK

redundant work. New designs of the BFS algorithm aim to avoid atomic operations because they are expensive [LS10].

2.3 Parallel Architectures

In this section, a summary of the different parallel architectures used to parallelise the BFS algorithm is presented. The classification is based on the taxonomy introduced by Buluç and Madduri [BM11], which consists of four main categories including, massively multithreaded systems, distributed memory systems, multicore systems, more recent heterogeneous architectures. In addition, parallel frameworks for graph analysis are also presented. Since the main contribution of this thesis is related to the vectorisation of the BFS algorithm in a multicore system, a compendium of the previous studies of the BFS algorithm vectorisation over the Intel Xeon Phi is presented in Section 2.3.3.

2.3.1 Massively Multithreaded Systems

The first implementation of the BFS in a machine supporting large numbers of threads, massively multi-threaded architecture, was described by Bader and Madduri [BM06] in 2006. The architecture was a 40-processor Cray MTA-2 and the results were impressively fast, a scale-free graph of 400 million vertices and 2 billion edges are processed in less than 5 seconds, and an absolute speedup close to 30 was achieved. The parallelisation of the BFS was the layer synchronous parallel algorithm applying parallelism at both levels (coarse and fine grains), exploring each vertex in the frontier and along the adjacency list exploration. This can be done because of the fine-grained synchronisation mechanisms of the MTA [ABH+03]. Another advantage is that the architecture does not use cache memory, which mitigates the need to optimise locality.

The recent general purpose GPU (Graphical Processing Unit) processors are similar to the Cray in terms of the capability to execute a massive amount of threads to hide memory latency [BM11]. These processors required that data access is contiguous, but because of the nature of the BFS (random memory access patterns), this can be challenging. Harish and Narayanan [HN07] explored various fundamental graph algorithms on a Nvidia GPUs, resulting in outperforming the CPU approaches. However, due to the restriction in memory size of CUDA (GPUs) only graphs with small degree were tested. Luo et al. [LWH10] improved
the performance by adding a hierarchical data structure to optimise the data memory access demonstrating results 10 times faster than Harish and Narayanan [HN07].

2.3.2 Distributed Memory Systems

In distributed memory Message Passing systems, essentially the graph, i.e. the vertices and the edges, is partitioned among the number of processors. Processing the graph involves communication among processors which can be expensive. A major challenge in these systems is to find a data layout based on the structure of the graph that increases local memory accesses and decreases processors intercommunication.

The general approach to implement the layer synchronous BFS algorithm in distributed memory systems has been tackled in the fine-grained parallelism, where the adjacency list of each vertex in the frontier is explored. Since the graph is split among the memory of multiple processors, the visited checks involved during the adjacency list exploration process would require a mechanism to keep data consistent. That mechanism would gather the information of the vertices from any of the processors it is located and scatter it back. The strategy used for the the adjacency list processing is known as edge aggregation-based [BM11], which consists of processing the vertices that are located to the memory of the local processor and storing in a list the non-local vertices. At the end, the accumulated list is propagated to the respective processors, where the memory of such vertices reside, through an all-to-all communication process. However, this all-to-all inter-processor communication has been proved to be a significant performance bottleneck [ACT05]. Yoo et al. [YCH+05] aimed to decrease the all-to-all interprocess communication by partitioning the graph in a two-dimensional layout. The benefit of this layout reduced the inter-communication among processors at most $\sqrt{p}$ processors, so expensive all-to-all communication would not be necessary anymore. Despite, the scalability of the algorithm in terms of number of processors to run, the two-dimensional layout has some restrictions such as the graphs need to have a regular degree distribution. Hence, this layout may not be applicable for large scale graphs with skewed degree distributions. Recently, Buluç et al. [BM11] proposed a hybrid approach of a simple vertex-based partitioning of the graph with the two-dimensional graph partitioning for skewed
degree distribution graphs demonstrating that it is possible to reduce communication time.

2.3.3 Multicore Systems

Another popular parallel architecture to exploit the BFS parallelism has been by using current multicore systems. Multicore systems offer a wide range of parallel capabilities according with the number of cores and simultaneous multithreading (SMT), which can vary from 2 to 32 cores and 2 to 4 SMT. These systems appear to be able to handle coarse-grained parallelism better than the previously introduced multithreaded systems. In addition, vector processors have been added to these architectures for fine-grained parallel processing, introduced further in Section 3.3, with different vector unit widths including 128-bit for SSE and 256-bit for AVX.

Despite the high degree of parallelism parallel that multicore architectures can offer, due to the irregular memory access pattern of the BFS, the memory hierarchy model of multicore systems seem to limit its performance. Hence, the BFS implementations in multicore systems have been oriented to improve not only the inter-socket communication but also aiming to exploit memory locality and to reduce the synchronisation cost as described next.

In 2010, Agarwal et al. [APPB10] came up with key optimisations that would lead into memory locality improvement, and the reduction of expensive atomic operations by using a quad-socket Intel Nehalem system built from commodity processors. The results shown speedups greater than the ones previously published by Bader and Madduri for the supercomputer MTA-2 [BM06]. The implementation is based on the same edge-aggregation approach used in distributed memory systems [BM11], where the graph is split across multiple sockets. Thus, local vertices in a socket are updated atomically, whereas the non-local vertices are aggregated in a list that is updated at the end by an inter-socket communication. Bitmaps are used as the data structure to keep track of visited vertices, which not only helps to reduce memory size but also increases memory locality. The usage of bitmaps is an important optimisation that has been used later by several studies (see Table 2.1). Xia and Prassana [XP09] also conducted a study related with the optimisation for the layer-synchronous BFS in the same multicore architecture as Agarwal et al. (dual-socket Nehalem system). However, they focused on the exploration of a topological adaptive barrier, which consists of the
estimation of the number of threads participating in each layer of processing. This estimation is based on a workload prediction that is dynamically calculated after each layer.

In 2010 Leiserson and Scharld presented a multithreaded implementation based on the layer synchronous algorithm that uses a multiset data structure [LS10]. The data structure used to replace the FIFO queue in the BFS is called “bag” and it is designed to be partitioned recursively using the “divide and conquer” fashion with help of the multithreaded programming language Cilk++ [Cor09] allowing a dynamic scheduling. The result is an algorithm that scales linearly with the number of processors. Despite the fact that this approach sounds promising, in this work, this multithreaded implementation was explored using OpenMP tasks but found to be less efficient than Cilk++.

Chhugani et al. [CSK+12] is another example of a successful implementation of the BFS on a multi-socket architecture. Their optimisations consist of decreasing the impact of memory latency by the addition of an atomic-free and cache-resident mechanism to the visited data structure of the BFS. This atomic-free mechanism is used as basis of the design of the relaxed BFS algorithm presented in Section 5.4.1. As previous studies did, they also proposed an algorithm to distribute work among the sockets while keeping inter-socket communication low.

In 2012, Beamer et al. [BAP12] added an important optimisation to the conventional level synchronous top-down algorithm. This optimisation consists of traversing the graph in a combination of exploring the vertices with the “top-down” fashion and the “bottom-up” direction. This hybrid approach lead to accelerating the search by reducing the total number of edges explored in comparison with the traditional top-down approach. To swap from one approach to the other one, a dynamic heuristic is calculated at runtime. In terms of algorithmic optimisations, this is the state-of-the-art of the parallel BFS and is described in detail in Chapter 6 where it is explored the hybrid BFS implementation developed in this thesis.

**Manycore Systems**

Recently, in 2012 Intel announced the Xeon Phi as a massive parallel architecture consisting of a single chip manycore architecture, including a unique and powerful
512-bit wide vector unit [Rah13]. For this reason, this architecture is interesting to explore the challenging parallelisation of graph algorithms.

Saule and Çatalyürek [SC12] were the first ones starting to explore graph algorithms before the Xeon Phi accelerator actually came out to the market. They explored parallelisation of different graph algorithms, including the BFS, in an Intel Xeon Phi prototype. The prototype consists of 31 computational cores each supporting up to four hardware threads, a 512 bit wide vector unit and 1 GB of GDDR5 of main memory, which is the similar to the commercially released Xeon Phi used in this thesis, see Chapter 3. The experiments consisted of exploiting the Xeon Phi by using three different programming models: Cilk Plus, OpenMP and Intel’s TBB [JR13]. Each of them resulting in three different BFS implementations. The first one is based on the bag implementation proposed by Leiserson et al. [LS10]. The second one is the BFS implementation presented in the SNAP library [BM08] that exploits a thread-local storage. This technique consists of keeping a local queue per thread, using OpenMP, to avoid synchronisation overhead when the queue is updated by different threads. At the end of each layer, all local queues are merged into the global queue. Finally, the third implementation is their own BFS implementation based on a concurrent queue data structure partitioned in blocks. The results of the three implementations show that the Intel Xeon Phi prototype was a promising parallel architecture to achieve speedup on irregular memory access patterns. Nevertheless, they did not make explicit usage of the potentially beneficial vector unit incorporated to the architecture.

After the Intel Xeon Phi was introduced in 2012, Gao et al. [TYG13] came up with an initial implementation of the BFS aiming to exploit the vector unit by making use of explicit vector function calls (intrinsic functions\(^1\)). This work was intended to be used as a general evaluation of the Xeon Phi for data-intensive applications. The implementation is based on the layer synchronous top-down BFS algorithm using bitmap arrays for the input, output and visited data structures. Specifically, vectorisation is applied in the adjacency list exploration. The use of bitmaps is an optimisation previously introduced by Agarwal et al. [APPB10]. However, using bitmaps during the vectorisation leads to race conditions because the vector unit is incapable of updating words at the bit level as discussed in this thesis in Section 5.3. To cope with that problem they relax the algorithm by

\(^1\)C style assembly functions that provide direct access to Intel’s vector instructions
allowing race conditions and then adding an extra process to recover the data consistency. The implementation was designed to use both modes of the Xeon Phi, native and offload (described in Section 3.4), and the results show a maximum speedup of 3.4 times.

In the same year, [SPR+14] present a vectorisation based on the version of the top-down BFS implementation that appears in the Graph500 benchmark. Their approach was based on vectorising the sequential version and the OpenMP parallel version in Graph 500 using vector intrinsics. The specific part that is vectorised is the adjacency list traversal where they are loaded into the vector unit. Contrary to [TYG13], Stanic et al. [SPR+14] does not make use of the optimisation related with the use of bitmap arrays for the input, output and visited lists, instead they use array of integers. The results presented show that vectorisation can be beneficial for graph algorithms based on a single thread analysis. Despite the simplicity of the experiments, the aftermath of this analysis indicates that prefetching is a crucial optimisation for the Xeon Phi. Prefetching is an optimisation technique utilised in the vectorisation of the BFS algorithm presented in Section 5.4.2.

One year later, in 2013 Gao et al. [GLZS14] presented a complete set of experiments related with their previous BFS implementation, where they add the hybrid version of the BFS algorithm. This version resulted in the fastest vectorised version on the Xeon Phi prior to the work in this thesis.

In 2014, Golovina et al. [GSF14] developed a parallelisation of the hybrid BFS algorithm on the Xeon Phi. The implementation is based on the layer synchronous BFS algorithm using queues and read-based as data structures. The main optimisations applied to improve performance on the Xeon Phi are loop unrolling and prefetching. They made a comparison between the Intel Xeon E5-2660 processor and the Intel Xeon Phi 7120P (a recent version of the Xeon Phi) [JR13]. The results demonstrated that the Xeon Phi has an average speedup of 137 % over the Xeon processor. They claim to achieve 4366 MTEPS for a graph of scale 25 and this result is listed in the 89th position of the Graph500 list [gra]. Nevertheless, it is not clear which type of vectorisation (automatic or manual) is applied to the algorithm.

Recently in 2015, [WLZ+15] present a study about the workload balance of the BFS algorithm on the Xeon Phi. They argued that the high variation of the degree of the vertices to be process leads into some multithread workload
imbalance issues. So, they propose an optimisation based on processing vertices with small degree separately from the ones with large degree. In contrast, in this thesis the scheduling based on the chunk parameter of the OpenMP library as described in Section 3.4.

Table 2.1 presents a summary that conveys previous studies aiming at the vectorisation of graph algorithms on the Intel Xeon Phi.

<table>
<thead>
<tr>
<th>Year</th>
<th>Reference</th>
<th>Approach</th>
<th>optimisation</th>
<th>vectorisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>Saule and Çatalyürek [SC12]</td>
<td>top-down</td>
<td>no optimisation</td>
<td>automatic</td>
</tr>
<tr>
<td>2013</td>
<td>Gao et al. [TYG13]</td>
<td>top-down</td>
<td>bitmaps</td>
<td>intrinsics</td>
</tr>
<tr>
<td>2013</td>
<td>Stanic et al. [SPR+14]</td>
<td>top-down</td>
<td>prefetching</td>
<td>intrinsics</td>
</tr>
<tr>
<td>2014</td>
<td>Gao et al. [GLZS14]</td>
<td>hybrid</td>
<td>bitmaps</td>
<td>intrinsics</td>
</tr>
<tr>
<td>2014</td>
<td>Golovina et al. [GSF14]</td>
<td>hybrid</td>
<td>loop unrolling</td>
<td>automatic</td>
</tr>
<tr>
<td>2015</td>
<td>Wang et al. [WLZ+15]</td>
<td>hybrid</td>
<td>vertex degree</td>
<td>intrinsics</td>
</tr>
</tbody>
</table>

Table 2.1: Previous studies related with the vectorisation of BFS algorithm on the Intel Xeon Phi.

The key contributions of this thesis builds on the studies carried out by Gao et al. in [TYG13] and [GLZS14]. In the first study, they present the vectorisation of the top-down BFS algorithm using vector intrinsic functions. Similar to them, as it is presented in Chapter 5, the graph is stored in a Graph 500 CSR format using bitmap arrays for the frontier, the visited array and the output queue. These data structures are used to improve the exploitation of data locality. Furthermore, for multithreading they came up with an extra process to avoid the bitmap race conditions as it is described in Section 5.3. Finally, they used intrinsic functions to have full access to the vector unit. Despite the fact that they present the vectorisation process for the top-down BFS algorithm, several details that impact performance remained unclear including prefetching, thread affinity and vector unit usage rate. Hence, in Chapter 5 all the details of vectorisation are targeted, first to understand them and second to improve performance, resulting in an implementation, published recently [PRL], that outperforms the implementation of Gao et al. [TYG13]. The second study is related with the vectorisation of the hybrid BFS algorithm. Similarly to the top-down BFS implementation in the first study, Gao et al. [TYG13] present the process of vectorising not only the top-down but also the bottom-up approach of the hybrid BFS algorithm.
Again, little detail of their implementation is provided, so Chapter 6 presents the vectorisation of the **bottom-up** approach of the hybrid BFS algorithm with a focus on a systematic analysis of the vector unit utilisation, etc. The results of the hybrid BFS algorithm presented in Section 6.6 are better in terms of performance compared against the one presented in [GLZS14].

In addition, the work done by [GSF14] about the vectorisation of the BFS algorithm is listed in the June 2016 Graph 500 list [gra]. The experiments were conducted in an Intel Xeon Phi (5110P) platform, similar to the one used in this thesis, resulting in 1.80 GTEPS for a graph size of scale 23. Section 6.6 presents a comparison between these results with the results of the vectorisation of the hybrid BFS algorithm implemented in this thesis.

### 2.3.4 Heterogeneous Systems

Gao et al. [GLZS14] present a heterogeneous BFS algorithm of the hybrid BFS algorithm using the Intel Xeon Phi as accelerator in an **off-load** mode. The algorithm consists of executing the **top-down** BFS algorithm only by the CPU but the **bottom-up** BFS algorithm by both, the CPU and the accelerator, since vertices can be divided and process independently.

Mayank Daga et al. [DNM14] proposed a heterogeneous version of the hybrid version of the BFS algorithm. According to them, this algorithm can be naturally accelerated by processing the **top-down** and the **bottom-up** BFS algorithms in a heterogeneous processor where each algorithm can be executed by a specific parallel architecture. Specifically, they used an APU (Accelerated Processing Unit) which comprises a CPU and a GPU (Graphics Processing Unit) processor on a single chip. The benefit of this architecture is that the memory space between the CPU and the GPU is unified, which reduces the transference time of data between both devices. Furthermore, the shift to heterogeneous architectures has been oriented to maximise performance under restrictive power and thermal constraints [AMD08].

Umuroglu et al. [UMJ15] also proposed a heterogeneous version of the **top-down** BFS algorithm using an architecture composed by CPU and FPGA (Field Programmable Gate Array). FPGA are devices that can be customised to deal with irregular memory access patterns, which are inherent in the BFS algorithm. The algorithm uses a matrix for the adjacency list representation, showing that
most random access of the BFS can be reduced and data can be kept in memory exploiting data locality.

2.4 Parallel Frameworks for Graph Analysis

In terms of software, several frameworks have been developed for the parallelisation of problems in graph analysis. Table 2.2 presents several current graph analysis frameworks. This table is based on the survey of frameworks presented by [EJRB13] in 2012, but including new packages. Packages are ordered by the number of edges that can be managed according to what has been published in the literature. However, for some new developments, an end-to-end analytics solution has not been published and it is difficult to know the maximum graph size supported so far. Until now, different packages such as Pajek [BBMM03], R (igraph) [CN06], Tulip [AAB12] and UCInet [BEF02] have been developed for graph analysis. Nevertheless, they run as a stand-alone applications and are limited by the main memory of the workstation. Thus, these packages only can handle small graphs (from several thousand vertices to a million of edges) without taking advantage of parallel systems.

Development of massive graph analysis frameworks can be classified according to its architecture. Most of them are designed to work on distributed parallel machines such as: Pregel [MAB10], The Parallel Boost Graph Library (PBGL) [GL05], The Knowledge Discovery Toolbox (KDT) [vMDC12], Pegasus [KTF09], Hama [Fou13b], Giraph [Fou13a], Graph Processing System (GPS) [GPS11] and Signal/Collect [SBC10].

Pregel was introduced by Google in 2012 for large-scale graph processing. It is based on the Bulk Synchronous Parallel programming model [Val90]. This model results in a flexible vertex-centric programming approach to handle graphs with an interface which is easy to program. PBGL is a C++ library and it is an extension to the Boost Graph Library for parallel distributed memory. KDT has a simple API in Python for large graph analysis that can be implemented on computer clusters. KDT’s high performance relies on the Combinatorial BLAS [BG11], which consists on a set of linear algebra primitives for graphs. Pegasus is a peta-scale graph mining library on the top of Hadoop, which is an open source framework for processing massive unstructured data for distributed applications. Apache Hama is a computing framework based on the Bulk Synchronous Parallel
model (BSP) on top of Hadoop for massive scientific computations such as matrix, graph and network algorithms. It was inspired by Pregel, but is different in the sense that it is purely BSP and is a general model, not just for graphs. Apache Giraph is an open source iterative graph processing system built for datasets at scale. It was inspired by Pregel, so it is also based on the BSP model but with additional features. GPS is an open source distributed message-passing system for large-scale graph computations. It is similar to Pregel and Giraph in three different aspects: scalability, fault-tolerance and ease of programming. Signal/Collect is a framework for synchronous and asynchronous parallel graph processing. It allows programmers to the many algorithms on graphs in a concise and elegant way and it is developed in Scala language.

Fewer shared memory frameworks exist: GraphCT [EJR13], MultiThreaded Graph Library (MTGL) [BHK07] and SNAP (Small-world Network Analysis and Partitioning) [BM08]. GraphCT is a collection of scalable graph algorithms for data analysis. It uses the benefits of the Cray XMT massively multithreaded architecture to scale graph algorithms, like, low overhead context switches and lightweight and fine-grained synchronisation primitives. MTGL is a small prototype C++ library inspired by BGL graph for query software for processing semantic graphs on multithreaded computers. It also uses the Cray (MTA-2) architecture to scale graph algorithms. SNAP is an open-source framework for exploratory study and partitioning graph analysis. It is implemented in C for shared memory, targeting sequential, multicore, and symmetric multiprocessors. For clarification, in Table 2.2 the term analytics refers to libraries containing a set of graph analytics algorithms, where as the term frameworks is used to define a complete platform that contains all the components participating in the processing of the graph in terms of communication, integration and execution of graph analytics.
CHAPTER 2. BREADTH-FIRST SEARCH RELATED WORK

<table>
<thead>
<tr>
<th>Package</th>
<th>Interface</th>
<th>Parallel</th>
<th>Memory</th>
<th>$O(\text{Edges})$</th>
<th>Analytics</th>
<th>Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel [MAB\textsuperscript{10}]</td>
<td>C++</td>
<td>X</td>
<td>Distributed on-disk</td>
<td>127 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MTGL [BHKK07]</td>
<td>C++</td>
<td>X</td>
<td>Shared (Cray XMT)</td>
<td>35 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>GraphCT [EJRB13]</td>
<td>C</td>
<td>X</td>
<td>Shared (Cray XMT)</td>
<td>17 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PBGL [GL05]</td>
<td>C++</td>
<td>X</td>
<td>Distributed in-memory</td>
<td>17 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>KDT [vMDC12]</td>
<td>Python</td>
<td>X</td>
<td>Distributed in-memory</td>
<td>8 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Pegasus [KTF09]</td>
<td>Hadoop</td>
<td>X</td>
<td>Distributed on-disk</td>
<td>6.6 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NetworkX</td>
<td>Python</td>
<td></td>
<td>Shared</td>
<td>100 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SNAP [BM08]</td>
<td>C</td>
<td>X</td>
<td>Shared</td>
<td>32 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Pajek [BBMM03]</td>
<td>Windows</td>
<td>Shared</td>
<td></td>
<td>16 billion</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>igraph [CN06]</td>
<td>R</td>
<td>Shared</td>
<td>Millions</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Green-Marl [HCSO12]</td>
<td>C++/Scala</td>
<td>X</td>
<td>shared memory (CPU)</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Hama [Fou13b]</td>
<td>Hadoop</td>
<td>X</td>
<td>Distributed on-disk</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>GPS [GPS11]</td>
<td>Hadoop</td>
<td>X</td>
<td>Distributed</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Giraph [Fou13a]</td>
<td>Hadoop</td>
<td>X</td>
<td>Distributed</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Signal/Collect [SBC10]</td>
<td>Scala</td>
<td>X</td>
<td>Distributed</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2.2: Graph analysis packages and frameworks currently under active development.

2.5 Summary

This chapter introduced the definition of the serial BFS algorithm and different variants for its parallelisation including the layer-synchronous algorithm and the parallel BFS. Different parallel architectures have been used to optimise the BFS algorithm due to its relevance for graph analysis algorithms. Those parallel platforms have been classified in four categories: massively multithreaded, distributed memory, multicore and heterogeneous systems. Furthermore, a summary of the parallel frameworks used for graph analysis is given. Particularly, the study of this thesis is focused on the investigation of the vectorisation of the BFS on the Intel Xeon Phi, which is a manycore architecture within the multicore system category. Hence, Section 2.3.3 introduces the previous studies in the literature that have already targeted the Intel Xeon Phi as parallel architecture for optimising the BFS algorithm. Specifically, Gao et al. [GLZS14] presented two implementations, the top-down and the hybrid BFS algorithms that are used for further comparisons in Chapter 5 and Chapter 6, respectively.
Chapter 3

The Intel Xeon Phi

The Intel Xeon Phi architecture is a powerful manycore architecture on a single chip, introduced in 2012 [Rah13]. The Xeon Phi architecture supports two levels of parallelism, a coarse-grained level by having up to 60 cores and a fine-grained level within a vector processing unit (VPU) integrated per core. The combination of both hardware resources make it possible to exploit a high degree of parallelism. Moreover, the 512-bit vector width of the VPU allows it to process up to 16 (32-bit) elements or 8 (64-bit) elements with a unique set of instructions, specialised for vector advanced processing, and is key hardware functionality that has to be exploited to obtain high performance.

This chapter introduces the architectural features of the Intel Xeon Phi in Section 3.1, including describing the number of cores, the memory system and the intercommunication between cores. Since the Xeon Phi is based on a shared-memory system, Section 3.2 presents a model of a shared-memory system to help to introduce the fundamental concepts involved, including multithreading, thread synchronisation, cache memory and the concept of race conditions. Furthermore, Section 3.3 introduces the way vector processing works, the basic functionality and how the instructions set for vector processing has been upgraded in the Xeon Phi. Section 3.4 describes the programming model and the compilation options for the shared-memory system and for vectorisation. Finally, Section 3.5 presents the performance counters supported, which provide a mechanism to get metrics of specific events associated with the hardware for the analysis of parallel programs aiming to improve performance. A summary of the chapter is given in Section 3.6.
3.1 Architecture

The Intel Xeon Phi coprocessor, also known as MIC, which stands for Intel Many Integrated Core has up to 61 cores. Specifically, The MIC (5110P) used in this thesis is composed of 60, Pentium-based cores [Rah13] and a shared main memory of 8 GB (GDDR5). The cores supports 4 hardware thread contexts per core, which is an architectural feature known as hardware multithreading [JR13], allowing a total of 240 threads before oversubscription occurs [HCC+14]. A threaded program is a program that is being executed by multiple processors sharing the code and also the data [HP11]. Each core contains a powerful 512-bit vector processing unit and a cache memory divided into L1 (32KB) and L2 (512KB) kept fully coherent by a global-distributed tag directory (TD), coordinated by a variant of the cache coherency MESI protocol [RH13]. Cores are interconnected through a high-speed bi-directional ring bus [Rah13] as is shown in Figure 3.1. The maximum memory bandwidth is quoted as 320GB/s. Table 3.1 summarises all these features.

<table>
<thead>
<tr>
<th>Model</th>
<th>5110P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>60</td>
</tr>
<tr>
<td>Number of threads</td>
<td>240</td>
</tr>
<tr>
<td>L1 cache / core</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 cache / core</td>
<td>512 KB</td>
</tr>
<tr>
<td>Main memory</td>
<td>8GB</td>
</tr>
<tr>
<td>Max. memory bandwidth</td>
<td>320 GB/s</td>
</tr>
<tr>
<td>Vector width</td>
<td>512-bit</td>
</tr>
</tbody>
</table>

Table 3.1: The Intel Xeon Phi features.

Figure 3.1: The Intel Xeon Phi microarchitecture.
CHAPTER 3. THE INTEL XEON PHI

Each vector processing unit (VPU) is composed of 128 512-bit vector registers and 32 16-bit mask registers, shared between 4 threads [Int13]. Each vector register can process either 16 (32-bit) operations or 8 (64-bit) operations at a time. A vector mask consists of 8 or 16 bits (depending on the use of single word\(^1\) 32-bit or double word 64-bit) that controls the operations applied to the elements loaded into the vector register. Only those elements whose mask bits are set to 1 are updated in the vector register, the ones with 0 value in the mask remain unchanged. Additionally, the Xeon Phi contains both hardware and software prefetching, which in some cases can help to reduce memory latency [JR13].

3.2 A Shared Memory System

The Xeon Phi architecture is based on a shared memory system, which consists of having multiple processors accessing a communal memory. The conventional programming model for this architecture exploits parallelism by executing multiple threads. Therefore, each processor executes a thread and the communication between them is done by exchanging data through the shared memory [JáJ92]. Figure 3.2 illustrates the basic concept of the shared-memory system used by the Xeon Phi. Notice that cache memory is introduced further in this section.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{shared_memory_system.png}
\caption{A basic shared memory system.}
\end{figure}

\(^1\)A word is a basic unit of data in hardware architecture designs and is a 32-bit length [HP11].
3.2.1 Threads Synchronisation

A problem arises when multiple threads are trying to modify the same memory address (i.e., update), which is also known as a race condition [HS08]. The problem is that, for a thread to modify data, two basic operations, read and write, are required. If another thread tries to modify the same data in between the two operations of the first thread, it can lead to inconsistencies in the state of the shared data. To keep data consistent, it is required to avoid any other thread accessing the shared data during its modification. For instance, Figure 3.3 illustrates two cases, an inconsistent and a consistent memory access while sharing data among two threads T1 and T2. In the first case, thread T1 reads the shared data but before T1 writes back to memory, thread T2 reads the shared data and writes it back straight away without interruption, only then does thread T1 write back to the shared memory. The interference of thread T2 while thread T1 was modifying the memory, results in an inconsistent state of the shared data (a missed write in this case). In the second case, the shared data is modified (read/write) by thread T1 and after that, thread T2 accesses the data to also modify it (read/write). After both modifications, shared data remains consistent because both threads execute the operations to modify (read/write) the shared data without interfering with each other.

![Threads race condition](image1)

![Updating shared memory consistently](image2)

(a) Threads race condition.  
(b) Updating shared memory consistently.

Figure 3.3: Examples of shared data between threads.

Having inconsistencies in the shared data leads to incorrect result of the program. Maintaining the correctness [HS08] of the program while improving performance is the main goal of parallel programs. To avoid this problem, there is a mechanism in multiprocessor architecture to execute both operations, read and
write, atomically, in other words without being interrupted by any other thread trying to modify the same data. This mechanism is known as an atomic operation. Particularly, the atomic operation is used further in Section 5.3 to avoid data race conditions is the hardware instruction `sync_fetch_and_or` part of built-in functions for atomic memory access of the GCC compiler [SD09]. In addition to thread synchronisation, there is another problem to mention, data dependencies, which happens when a set of instructions in a program need to follow certain execution order, otherwise the result is incorrect. When having multiple threads executing instructions where the order matters, it is necessary to synchronise all them. One mechanism to manage this problem is the well known barrier [HS08] and is commonly implemented in multiprocessor architectures.

### 3.2.2 Cache Memory

While over 40 years processors were doubling their performance roughly every 18 months, the main memory, random access memory (RAM), did not evolve at the same pace, causing a significant disparity between the CPU speed and the time to access memory. In other words, even though CPUs could execute instructions at high speed, it would take hundreds of cycles to fetch a byte from RAM, causing the CPU to wait. To cope with this problem a smaller but faster memory was introduced in modern processor architectures called cache memory [HP11].

The fundamental idea of cache memory is simple; when data is requested from memory, it does not only fetch the requested data but also a segment of contiguous data, known as a cache line, expecting it to be used in the following instructions [Nys14]. For instance, if one byte is requested from memory, the cache memory will end up with all the adjacent bytes that fit in a cache line as is illustrated in Figure 3.4.

![Figure 3.4: Example of a byte requested in a cache line.](image)

When the next instruction is a request for data and the data is successfully found in the cacheline, is called cache hit and when it is not found it is known as a cache miss. The general strategy to reduce the time to fetch data from main
memory has been to organise the data in a way to increase cache hits and decrease cache misses. A simple solution is exploiting *data locality*, *spatial locality* consists of the organisation of data structures in a way that instructions process data that are located next each other in memory [Nys14]. *Temporal locality* refers to data that is more likely to be needed in the near future [HP11]. The Xeon Phi has a cache memory hierarchy. Each processor has two cache memories, L1 and L2, of 32 KB and 512 KB sizes respectively. In addition, to add extra performance *prefetching* is a technique to help to hide memory latency by fetching data that is expected to be used in the next instructions. The Xeon Phi supports two types of prefetching, by software or hardware, whether prefetching at hardware is applied depends on a heuristic looking for memory access patterns. However, hardware prefetchers typically support only simple memory access patterns and software prefetchers can be used to help to improve performance [JR13].

### 3.3 Vector Processing

As was introduced in Section 1.3, the SIMD programming model refers to the capability to process multiple data utilising the same instruction, also known as vector processing. An example of a simple vector processing is shown in Figure 3.5, where one instruction is given to four processing units (PU), each of them accessing their own data.

![Figure 3.5: Single Instruction Multiple Data (SIMD) processing.](image)

Despite the Illiac IV, previously introduced in Section 1.3, being the first attempt for parallel processing based on SIMD programming, the vector computer was not very successful as a mainstream computer in the market, mainly because it consisted of specialised and high cost processors such as the Cray [HP11]. Nevertheless, recently new vector units have became versatile, and they have started to be included in commodity desktops as well as in supercomputers. The first Intel
Table 3.2: Development of SIMD instructions set developed by Intel.

<table>
<thead>
<tr>
<th>Year</th>
<th>Instruction Set</th>
<th>Data width</th>
<th>32-bit integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996</td>
<td>MMX</td>
<td>64 bits</td>
<td>2</td>
</tr>
<tr>
<td>1999</td>
<td>SSE</td>
<td>128 bits</td>
<td>4</td>
</tr>
<tr>
<td>2008</td>
<td>AVX</td>
<td>256 bits</td>
<td>8</td>
</tr>
<tr>
<td>2013</td>
<td>AVX-512</td>
<td>512 bits</td>
<td>16</td>
</tr>
</tbody>
</table>

SIMD architecture was introduced by Intel in 1996 by extending the instruction set of Pentium processors with a specific set of vector instructions called MMX (Multi Media Extensions) with a length of 64 bits for integer arithmetic [Yu97]. Since then, the instruction sets for programming the vector unit in commodity computers have been continuously being upgraded. Table 3.2 shows a list of how SIMD instructions sets have evolved through time [SSDK14]. The Intel Xeon Phi uses the recent instruction set AVX-512 optimised for vector instructions, allowing to process either 16 32-bit (single precision) or 8 64-bit (double precision) elements at a time.

Apart from having a wide vector of 512 bits, the AVX-512 instruction set is optimised for specific hardware operations that benefit the performance of vectorisation including vector masks, scatter/gather operations and data alignment, which will be discussed next.

**Vector mask:** This is a set of bits, where each bit corresponds to an element in the vector unit. The vector instruction for only the elements in the vector unit that has associated a bit value of 1 in the vector mask will be executed. The elements having a zero bit value remains the same. The vector mask for 32-bit elements has 16 bits and 8 for 64-bit elements.

**Scatter and Gather operations:** These operations are two instructions that the Xeon Phi use to deal with non-contiguous memory loads (gather) to the vector register and data stores back (scatter) to memory. Both operations receive, as argument, a list of indexes to be scattered to/gathered from in the array.

**Data alignment** This is a data specification to assist the compiler in the creation of memory objects on specific byte boundaries. The optimal data alignment for the Xeon Phi is on 64 byte boundaries [JR13].
3.4 Programming Model

Since the Xeon Phi provides two levels of parallelism, where the first one can be exploited by multithreading and the second one by using vectorisation. This section describes the details required for programming both levels: *multithreading* and *vectorisation*.

3.4.1 Multithreading

OpenMP is a versatile interface to program shared-memory multithreaded systems [CJP07a] such as the Xeon Phi. The parallelisation consists of setting directives over the segments of code in the program that will be parallelised, those directives are read and transformed by the compiler, so different threads can execute those instructions. The OpenMP directives are also known as *pragmas*. An example of the parallelisation of a simple for loop using a *pragma* is shown in Listing 1. In this case, it can be seen that the *pragma* is before the *for* loop, indicating to the compiler that different threads will be executing that segment of code within the *for* loop.

```c
int N=10;
//Parallel region
#pragma omp parallel for
for(int i=0;i<N;i++)
{
    task(i);
}
```

Listing 1: OpenMP program example.

**Scheduling**

The concept of scheduling refers to the organisation of the division of work among multiple threads, generally aiming to achieve a balanced workload over the threads. In other words, threads keep working evenly without having threads either executing only a few tasks or too many tasks. However, keeping the workload balanced among threads is a difficult area that is still being investigated [HS08]. OpenMP has a mechanism for threads scheduling, providing 5 different options that are set in the parameters of the *pragma* directive [CJP07b]:

```c
int N=10;
//Parallel region
#pragma omp parallel for
for(int i=0;i<N;i++)
{
    task(i);
}
```
CHAPTER 3. THE INTEL XEON PHI

static This scheduling divides the number of tasks into equal-sized chunks.

dynamic In this scheduling each thread is assigned to a task, of a chunk size, from a queue of work. When a thread has finished, the scheduler proceeds to assign more tasks, from the queue.

guided This scheduling is similar to the dynamic one but the chunk size is large at the beginning and it get smaller along the execution of the program.

auto This scheduling gives full control to the compiler to do the workload balance.

runtime This scheduling is set on a variable environment to allow the setting of the scheduling type at execution time.

Thread Affinity

Thread affinity is a feature in architectures that enables the pinning of threads to specific logical cores on the physical cores. On the Xeon Phi, each core supports up to 4 hardware threads and thread affinity enables the user to affect the use of shared resources in a core (e.g. cache and memory bandwidth). The Xeon Phi provides three strategies for controlling thread affinity: compact, scatter and balanced:

compact This affinity assigns free threads according to the location of the thread in each core, as close as possible (i.e. to logical cores on the same physical core).

scatter This affinity distributes the threads as widely as possible across the entire set of physical cores, using one logical core per physical core for each thread placed up to the number of physical cores and then cycling through the physical cores again.

balanced This is similar to scatter but places threads with adjacent thread ids on the same core.

3.4.2 Vectorisation

The vector unit of the Intel Xeon Phi can be used by using two programming approaches: automatic and manual vectorisation, described next.
CHAPTER 3. THE INTEL XEON PHI

Automatic Vectorisation

In automatic vectorisation the compiler is activated to identify and to optimise the candidate code without intervention of the programmer. Then the compiler detects the segments of code that can be vectorised and internally transform the sequential code into a vector friendly code. Despite the idea of having automatic vectorisation being ideal, there are some obstacles that makes automatic vectorisation difficult to achieve.

**non-contiguous data** When the data is located adjacent in memory, then automatic vectorisation is easy to apply by just loading the complete set of elements directly to the vector unit. However, if the data is non-contiguous it is necessary to fetch (gather) the data independently, which can be less efficient.

**data dependencies** Vectorisation can only be applied if the data of the elements is independent each other. For example, if the data in the first element is dependent on other elements, it is not secure applying vectorisation because it might cause incorrect results.

An extra level of programmer control for automatic vectorisation is by using OpenMP pragma SIMD directives [CJP07b], which are used as hints. For example, using hints it is possible to force the compiler to use vectorisation without checking safety restrictions when data dependencies happen to be there. For instance, `#pragma ivdep` (ignore vector dependencies) directive ignores data dependencies in the code and it proceeds for vectorisation. However, the usage of these `pragma` directives still requires an extra effort to transform the code to make it vector friendly.

Manual Vectorisation

Intrinsic functions are a set of assembly functions that are handled specially by the compiler and include hardware optimisations for the *scatter* and *gather* functions. Intrinsic functions give the programmer full low-level control over the vector unit, which leads to having to be aware of different architectural requirements including, *data alignment*, *vector masks* and *prefetching*. The set of intrinsic functions used for the BFS implementations in Chapters 5 and 6 is presented in Appendix D.
3.4.3 Compilation

The Intel Xeon Phi coprocessor has two ways of programming, the first one is known as the native mode and it refers to when the program is compiled to run directly on the coprocessor. The second one is offload mode and it is when the Phi is used as a coprocessor and is programmed in a heterogeneous fashion cooperating with the host processor, which is typically a standard Intel CPU. Both programming modes can be cross compiled using the intel C++ compiler and intel fortran compiler by setting the compiler flag -mmic in Linux operating system [Int12d].

The OpenMP library was selected as a multithreading parallel platform for the Intel Xeon Phi and it can be activated in Linux by adding the flag -fopenmp to the compilation line. Regarding vectorisation, firstly automatic vectorisation can be activated by the compiler by setting the optimisation flag -02 or higher [Int12a]. In addition, by setting the flag -vec-report the compiler will display a diagnostic about the segments of code candidates for vectorisation. Secondly, vectorisation using OpenMP pragmas, which provides programmer guided vectorisation [CJP07b]. Thirdly, to vectorise with the usage of intrinsic functions the header file zmmintrin.h needs to be included to access to the AVX-512 SIMD instruction set [Rah12].

For the experiments presented in Chapter 5 and Chapter 6, the compilation of the code used the Intel C++ compiler (version 14.0.0) with the optimisation flag -02 and -fopenmp. Table 3.3 shows a summary of the programming options used during the setting up of the experiments.

<table>
<thead>
<tr>
<th>compiler</th>
<th>Intel C++</th>
<th>14.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt flag</td>
<td>-02</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>native mode</td>
<td></td>
</tr>
<tr>
<td>multithread</td>
<td>OpenMP</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>vectorisation</td>
<td>intrinsic functions</td>
<td></td>
</tr>
<tr>
<td>scheduling</td>
<td>static</td>
<td>chunk size = 8</td>
</tr>
<tr>
<td>thread affinity</td>
<td>balanced</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Experimental settings.
3.5 Performance Counters

The performance optimisation of a program is a complex task that involves different levels of knowledge [Int12b]. Starting from the algorithmic design to the specific characteristics of the hardware platform being used for execution, aiming to find hot spots in the source code that might impacting performance. For instance, data structures are key factors to obtain performance because with a cache friendly layout, data locality can be exploited, resulting in an improvement of performance. Thus, the optimisation includes the analysis of those hot spots at the architecture level for further tuning. To help this process, there are tools to get access to real-time hardware performance information such as the PAPI (Performance Application Programming Interface) library [oT16]. This library provides access to various hardware performance counters by tracing different events, while the program is being executed. The PAPI counters available for the Xeon Phi are listed in Table 3.4. Only two performance counters can be monitored in a single program per execution. Five of the PAPI counters in this list were used for the performance analysis of the experiments in Chapter 6. The reasons why these counters were selected is specified in Section 6.5.2.

<table>
<thead>
<tr>
<th>Events</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_TLB_DM</td>
<td>Data translation lookaside</td>
</tr>
<tr>
<td>PAPI_TLB_IM</td>
<td>Instruction translation</td>
</tr>
<tr>
<td>PAPI_L2_LDM</td>
<td>Level 2 load misses</td>
</tr>
<tr>
<td>PAPI_BR_MSP</td>
<td>Conditional branch instructions</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>Instructions completed</td>
</tr>
<tr>
<td>PAPI_LD_INS</td>
<td>Load instructions</td>
</tr>
<tr>
<td>PAPI_SR_INS</td>
<td>Store instructions</td>
</tr>
<tr>
<td>PAPI_BR_INS</td>
<td>Branch instructions</td>
</tr>
<tr>
<td>PAPI_VEC_INS</td>
<td>Vector/SIMD instructions</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>Total cycles</td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>Level 1 data cache accesses</td>
</tr>
<tr>
<td>PAPI_L1_ICA</td>
<td>Level 1 instruction cache accesses</td>
</tr>
</tbody>
</table>

Table 3.4: PAPI events available for the Intel Xeon Phi.
3.6 Summary

This chapter introduced the architectural features of the Intel Xeon Phi, which is the parallel platform used for the study purposes of this thesis. Starting with the description of a shared-memory model, similar to the one utilised by the Xeon Phi, the main concepts and synchronisation problems involved during the parallelisation of a program, including race conditions, atomic operations and barriers were discussed. These concepts are the basis for a smooth introduction of the parallel algorithms studies in this thesis. Moreover, the programming model of the shared-memory system and the vector unit presented in Section 3.4 are the ones utilised in the experiments sections of Chapter 5 and Chapter 6. Finally, performance counters are a key tool supporting the analysis of the performance of algorithms, thus the counters described in Section 3.5 are further utilised for the analysis of the bottom-up BFS algorithm in Section 6.5.
Chapter 4

The Experimental Development Framework

Graph algorithms have been amply investigated due to their importance for analysis of modern applications but there was no standard benchmark to evaluate them, until 2010 when the Graph 500 benchmark emerged to fulfill this gap [MWBA10]. The Graph 500 benchmark consists of an experimental suite aiming to facilitate the creation and manipulation of large graphs, allowing the performance of graph algorithms to be measured across multiple parallel architectures.

This chapter describes the Graph 500 benchmark, including the graph generator, the experimental suite and the output performance information in Section 4.1. To facilitate the integration of the Graph 500 experimental suite with new graph kernel implementations over multiple architectures in a scalable way, the Experimental Development Framework (EDF) was created and described in Section 4.2. EDF is a framework that extends the Graph 500 and it is one of the contributions of this thesis. Finally a summary of the chapter is given in Section 4.3.

4.1 The Graph500 Benchmark

The Graph 500 benchmark is a benchmark to measure graph processing performance. There are three graph kernels measured: concurrent search (BFS), optimisation (single source shortest path), and edge-oriented (maximal independent set). The benchmark provides a graph generator, a suite to run experiments related with the graph kernel and the output performance information [gra10].
The Graph 500 benchmark is useful not only for evaluating and comparing different graph algorithms implementations but also over multiple architectures.

4.1.1 Graph Generator

Graph 500 creates graphs using the Kronecker generator [LCK+10], which aims to generate synthetic graphs with similar characteristics to graphs generated from real data. Real graphs generated by complex networks exhibit two key characteristics: small diameters and heavy-tailed degree distribution [CZF04], [Mil67]. First, a graph has small diameter when any vertex can be reachable from another one by traversing a small number of edges. An example of this topology can be seen in social networks, where it has been demonstrated that there are short paths in global friendship network. The experiment consists of finding randomly selected “target” people in a social network, starting from distant individuals. The result of the experiment showed that the “target” people were separated from the distant individuals by only six steps, which led to the concept six degrees of separation or small-world phenomenon [Mil67]. Second, the heavy-tailed degree distribution is associated with how the edges in a graph are spread around the vertices. The density of edges in a real graph can be described by a power-law distribution, which implies that the number of edges per node grows linearly over time [LKF05]. The size of the graph is specified by two variables: the scale and the edgefactor. The total number of vertices in the graph is equal to $2^{\text{SCALE}}$ and the total number of entry edges is equal to $2^{\text{SCALE}} \times \text{edgefactor}$. Figure 4.1 presents the vertex degree distribution for a graph size of $\text{scale} = 10$ and $\text{edgefactor} = 16$, generated by Graph 500. The vertex degree, which is the number of edges related to that vertex, varies in a range of 1 to 485 edges, most of edges are spread around the edgefactor (16), and they decrease along a long tail until only having one vertex, thus the maximum degree vertex is 485. Note that despite the total number of entry edges is $2^{10} \times 16 = 16,384$, Figure 4.1 only shows the ones created by the graph generator.

The Kronecker generator is a simple model based on a recursive construction of matrices by a process known as the Kronecker product or tensor product. Leskovec et al. [LKF05] showed that this model is well-suited to the generation of graphs with low diameter and heavy-tailed degree distribution in a fast and scalable fashion. Using this method, large graphs are generated by recursively multiplying smaller, self-similar graphs. The process starts by defining an initiator graph $G_1$
Figure 4.1: Example of vertex degree distribution for a graph size of \( \text{scale} = 10 \) and \( \text{edgefactor} = 16 \).

with \( N \) vertices and \( E \) edges, represented by the \( N \times N \) adjacency matrix \( A(G_1) \) for \( N = 2 \).

\[
A(G_1) = \begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix}
\]

The Kronecker product of two graphs is the tensor product of their adjacency matrices. For instance, the Kronecker product of two graphs, \( G \) and \( H \), can be described as the tensor product of the adjacency matrix of \( A(G) \) and the adjacency matrix of \( A(H) \), \( A(G) \otimes A(H) \) where \( N \times N \) represents tensor product.

**Definition 4.1.1 (Kronecker product of matrices)** Given two matrices \( A \) and \( B \) of sizes \( n \times m \) and \( n' \times m' \) respectively, the Kronecker product \( C \) is defined by the tensor product between \( A \) and \( B \). For \( 2 \times 2 \) matrices this gives:

\[
C = A \otimes B = \begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix} \otimes \begin{bmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
\end{bmatrix}
\]

\[
C = A \otimes B = \begin{bmatrix}
  a_{1,1} & b_{1,1} & a_{1,2} & b_{1,2} \\
  a_{1,1} & b_{2,1} & a_{1,2} & b_{2,2} \\
  a_{2,1} & b_{1,1} & a_{2,2} & b_{1,2} \\
  a_{2,1} & b_{2,1} & a_{2,2} & b_{2,2}
\end{bmatrix}
\]

To generate a larger graphs, the Kronecker product is recursively applied to the initiator graph \( A(G_1) \) by applying the tensor power defined as follows:
Definition 4.1.2 (Tensor power) Given the adjacency matrix \( A(G_1) \) and \( n \), a non-negative integer, the \( n \)-th tensor power of \( A(G_1) \) is denoted by the \( n \)-fold of the matrix \( A(G_1) \).

\[
A(G_1)^\otimes n = A(G_1) \otimes \cdots \otimes A(G_1)
\]

For example, having an initiator graph defined by the adjacency matrix \( A(G_1) \), the resulting tensor product is \( A(G_1)^\otimes 2 \).

\[
A(G_1) = \begin{bmatrix}
1 & 1 & 0 \\
1 & 1 & 1 \\
0 & 1 & 1
\end{bmatrix}, \quad A(G_1)^\otimes 2 = \begin{bmatrix}
A(G_1) & A(G_1) & 0 \\
A(G_1) & A(G_1) & A(G_1) \\
0 & A(G_1) & A(G_1)
\end{bmatrix}
\]

Again using \( A(G_1) \) as initiator graph, Figure 4.2 shows two examples of the corresponding adjacency matrix of the graphs \( A(G_1)^\otimes 3 \) and \( A(G_1)^\otimes 4 \) generated using \( A(G_1) \) as initiator graph.

![Figure 4.2: Examples of the adjacency matrices of the graphs \( A(G_1)^\otimes 3 \) and \( A(G_1)^\otimes 4 \), created with \( G_1 \) as graph initiator. Source image [LCK+10].](image)

The distribution of the edges within the graph is defined by four probabilities used to spread the edges along the graph: \( a, b, c \) and \( d \) with \( a + b + c + d = 1 \). The process of distributing the edges across the graph begin by populating the initiator graph. The initiator graph is subdivided into 4 sections, where each section is related with one of the probabilities. Based on these probabilities, edges are uniformly distributed within each section. While the recursive tensor product
is applied to create larger graphs, the proportion of the edges remains constant as matrices get bigger. Figure 4.3 shows an example of an initiator matrix with probabilities $a$, $b$, $c$ and $d$ that maintains the probability distribution along the process of enlarging the graph.

![Edge distribution in the adjacency matrix based on $a$, $b$, $c$ and $d$ probabilities.](source_image)

The graph generator input parameters are the size and the four probabilities to generate the edge distribution. As it was mentioned before, the size of the graph is specified by two variables: the scale and the edgefactor. The total of number vertices in the graph is equal to $2^{SCALE}$ and the total number of edges is equal to $2^{SCALE} \times edgefactor$, including self-loops and repeated edges. The probabilities are specified by ($a$, $b$, $c$ and $d$). The standard set of parameters defined by Graph 500 are ($a=0.57$, $b=0.19$, $c=0.19$ and $d=0.05$). The result of the graph generator is a list of tuples, where each tuple contains an edge denoted by the starting vertex and the ending vertex. This list can be either loaded directly to memory at execution time or can be directly stored into a file.

**Compressed Sparse Row**

As was mentioned in Section 1.2.1, a graph can be represented by either an adjacency list or an adjacency matrix. A Compressed Sparse Row (CSR) is a variation of the adjacency list, which is efficient to store sparse graphs. Graph 500 uses a CSR implementation to load the graph, which is composed by two integer arrays: rows and colstarts. The rows array contains the adjacency list of every vertex and its size is the total number of edges in the graph. The colstarts stores the start and the ends indexes related to every vertex pointing to the rows.
array. Thus, the size of the colstarts is the total number of vertices times two because it contains the starts and the ends indexes of each vertex in the graph.

An example of the two arrays used in the CSR is illustrated in Figure 4.4.

### 4.1.2 Graph 500 Experimental Suite

The Graph 500 Experimental Suite consists of three modules: the input parameters, the experiments and the output. Firstly, the input parameters consist of an input graph, generated using the generator and loaded as a CSR, and a set of 64 starting vertices. The starting vertices are randomly chosen and they are also known as roots. Secondly, the experiments related with the BFS graph kernel consist of 64 BFS executions using each one of the starting vertices as input. To avoid trivial searches, the starting vertices are required to be connected to some other vertex by having a degree of at least one. After each BFS execution, the output BFS tree is verified by calling the validation process with the starting vertex as input. Every BFS iteration is timed and after the 64 iterations have been executed, the output performance information is presented. Figure 4.5 shows a sketch of the modules of the Graph 500 Experimental Suite.

**Breadth-First Search Graph Kernel**

The BFS graph kernel receives as input: the graph and the starting vertex, and as output, an array to hold the output BFS tree, which is a sequence of vertices and their parents representing the BFS tree, where the starting vertex is the only one vertex set as its own parent. An example of a small BFS tree and its predecessor array is shown in Figure 4.6.
CHAPTER 4. THE EXPERIMENTAL DEVELOPMENT FRAMEWORK

4.1 PERFORMANCE ANALYSIS

New BFS implementations can be integrated into the Graph 500 Experimental Suite as long as they meet the input and output requirements. A new software framework was developed to facilitate the integration of multiple implementations into the experimental suite. This is called the Experimental Development Framework (EDF) and it is one of the contributions of this thesis, a detailed description is presented further in Section 4.2.
Validation

The benchmark also contains a method to verify that output BFS trees are valid. A full check of the output BFS tree would involve finding all the possible BFS path options so as to verify that the output BFS tree is among them. However, this would be very time consuming. Instead, Graph 500 uses a “soft” check of the output. This is not intended to provide a full check of the output BFS tree, but instead provides the following checks as described in [gra10].

1. The BFS tree is a tree and does not contain cycles.
2. Each tree edge connects vertices whose BFS layers differ by exactly one.
3. Every edge in the input list has vertices with levels that differ by at most one or that both are not in the BFS tree.
4. The BFS tree spans an entire connected subset of the vertices.
5. Each vertex and its parent are joined by an edge in the original graph.

Output Performance Information

Along with the experiments, performance information is collected and output statistics are presented. In every BFS execution, the number of traversed edges is counted and the time is measured. The relation between both measurements results in a metric called TEPS (traversed edges per second). After the 64 iterations have been completed, statistics related with number of edges, time and the TEPS are the output performance information produced by the Graph 500 experimental suite.

To compare the performance of different BFS implementations on multiple architectures, Graph 500 came up with a metric that measures the number of Traversed Edges Per Second (TEPS). This metric is a ratio between the number of edges that have been traversed along the BFS execution and the time that has lapsed (in seconds). Equation 4.1 shows the definition for the TEPS metric, where $m$ is the number of edges and $time_{BFS}(n)$ refers to the measured execution time for the BFS graph kernel for the $n^{th}$ experiment.

$$TEPS(n) = \frac{m}{time_{BFS}(n)}$$ (4.1)
In addition, graph 500 also produces statistics to help to analyse the BFS performance. Table 4.1 lists the quartiles (min, first quartile, median, third quartile, max) related with three metrics: time, number of edges and TEPS. Also the mean and the standard deviation are calculated for the time and the number of edges. Since TEPS is a rate, the harmonic mean is deemed appropriate for calculating the average of rates [Nor40]. An example of the Graph 500 output statistics is presented in Appendix A.

<table>
<thead>
<tr>
<th>Time</th>
<th>Number of Edges</th>
<th>TEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_time</td>
<td>min_nedge</td>
<td>min_TEPS</td>
</tr>
<tr>
<td>first quartile_time</td>
<td>first quartile_nedge</td>
<td>first quartile_TEPS</td>
</tr>
<tr>
<td>median_time</td>
<td>median_nedge</td>
<td>median_TEPS</td>
</tr>
<tr>
<td>third quartile_time</td>
<td>third quartile_nedge</td>
<td>third quartile_TEPS</td>
</tr>
<tr>
<td>max_time</td>
<td>max_nedge</td>
<td>max_TEPS</td>
</tr>
<tr>
<td>mean_time</td>
<td>mean_nedge</td>
<td>harmonic mean_TEPS</td>
</tr>
<tr>
<td>stddev_time</td>
<td>stddev_nedge</td>
<td>harmonic stddev_TEPS</td>
</tr>
</tbody>
</table>

Table 4.1: Graph 500 output quartiles for the BFS algorithm of time, number of edges and TEPS.

The statistics given by Graph 500 can be represented by a boxplot. A boxplot is a technique to display the distribution of data in five sets: first quartile, third quartile, minimum and maximum value and the median. In the box, the rectangle spans from the first quartile to the third quartile, also called interquartile range or IQR. In this range, 50% of the total data is located here, including the median. The two lines vertically extending from the boxes, called whiskers, indicate the variability out of the box (first and third quartile) up to the maximum and minimum extremes. Additionally, there are outliers, which are points that are distant to the observations [MTL78]. Figure 4.7 illustrates an example of a boxplot pointing out the five sets and the outliers, which are represented by crosses (+) and circles (○). Later, in Chapter 6 boxplots are used to show the variability of the number of TEPS (traversed edges per second) of the experiments.

**Definition 4.1.3** The harmonic mean of a sequence of positive independent chance variables denoted by \( x_1 = x_1, x_2, \ldots, x_n \), is presented in Equation 4.2.

\[
HM = \frac{1}{x_1 + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} = \sum_{i=1}^{n} \frac{1}{x_i}
\]  
(4.2)

Hence, the harmonic mean of the number of TEPS can be interpreted as in the Equation 4.3, where \( n \) is equal to the 64 samples for the randomly selected starting vertices.

\[
HM_{\text{TEPS}} = \frac{n}{\sum_{i=1}^{n} \text{TEPS}_{\text{BFS}}(i)}
\]  

(4.3)

The harmonic mean of a list of elements tends towards the least element in the list. The harmonic mean of a list of TEPS should therefore lie in the range of TEPS in the list. In the graph 500 experimental suite, among the 64 BFS executions, approximately 10% typically result in zero TEPS. The reason is because their starting vertices are unconnected to the main subset of the graph being traversed. The impact of having zero TEPS to calculate the harmonic mean result in undefined divisions (1 divided by zero) for those starting vertices. To avoid these undefined divisions, Graph 500 set them to zero. However, this result in an inaccurate harmonic mean that sometimes overpasses the range of the TEPS in the list because even when the undefined divisions are set to zero, the total number of executions is kept as 64. Appendix A shows an example of the zero TEPS along the 64 executions. Despite this problem, the mean used by different authors [GLZS14], [GSF14], [SPR+14] and [WLZ+15] to present their results still remains the harmonic mean. Hence, to be able to compare the results presented in this work with previous work, the harmonic mean is used. However,
in some indicated experiments the maximum number of TEPS, also in the output
statistics, can be used to present a different view of the results.

4.2 Experimental Development Framework

The development framework is one of the contributions of this thesis and it
emerged from the necessity of wishing to develop multiple implementations of
the same graph kernel, all sharing the Graph 500 experimental platform. At the
moment, the Graph 500 experimental suite is programmed based on structured
programming using C language based on a set of programs separated in differ-
ent files. Due to the necessity of having to implement multiple implementations
of the BFS algorithm in this thesis, a development framework based on Object
Oriented Programming (OOP) was created as an extension of the Graph 500 ex-
perimental platform to facilitate the integration not only of multiple graph kernel
implementations but also across various architectures in a scalable fashion.

On the one hand, multiple graph kernels implementations can differ according
to their architectural features, algorithmic design and compilation options. On
the other hand, the Graph 500 experimental suite includes modules such as the
d graph generator, the 64 experimental setup, the validation process and the output
performance information that are common to all implementations. By integrating
the Graph 500 experimental suite with the scalable development framework, the
creation and evaluation of new graph kernel implementations becomes easier.
Figure 4.8 presents the conceptual scheme of this integration.

The design of the framework is based on the Object Oriented Programming
(OOP) model [BME⁺07], which allows to structure the code in modules that are
easy to handle and maintain. The OOP model consists of identifying different
entities in a program, analyzing them and modeling their interaction. Abstractly,
the entities are identified as classes and their specific instances are objects. The
interaction between classes is described by a class hierarchy model. In addition,
software design patterns can be used to help to identify the relationship and inter-
actions between the classes as the parts of code that can be reusable [GHJV95].
In the following section, the class hierarchy model of the development framework
is described.
4.2.1 Class Hierarchy Model

The class hierarchy model of the development framework consists of a main class called Graph. This class is an interface that integrates the Graph 500 suite with the multiple BFS implementations. The interface consists of several methods to get access to the Graph 500 experimental suite functions including generating and loading of the graph and the verification process of the output. Additionally, the Graph class contains the methods that will be further implemented by the new implementations of the graph kernels. For instance, the BFS graph kernel is expected to have different implementations over multiple architectures.

Figure 4.9 presents the class hierarchy model of the development framework. At the top, there is the graph 500 experimental suite, which has a direct interaction with the Graph class. This class contains an internal representation of the graph in a CSR format, introduced previously in section 4.1, the functions to access the Graph 500 functions and a virtual method for each graph kernel. A virtual method in OOP is a function whose implementation (body function) can be overridable at runtime. This specific feature of the OOP makes it possible to actually have different implementations of the same function and swap between them at execution time. A graph kernel is shown containing a virtual method identified as graphKernel() in the Figure 4.9. Finally a set of classes inheriting from the Graph class illustrates the different implementations that one graph
CHAPTER 4. THE EXPERIMENTAL DEVELOPMENT FRAMEWORK

kernel may have. As an example, the implementations can be developed over different architectures such as ARM, Intel and a heterogeneous architectures (e.g. AMD with CPUs and GPUs).

![Class hierarchy model of the development framework.](image)

By using this framework each new graph kernel implementation is created in a separated class that can be easily integrated into the model by inheriting from the Graph class. This inheritance allows the sub-classes (new implementations) to have direct access to the graph 500 functions without major changes. The source code generated using the framework becomes modular and easy to maintain.
4.2.2 Class Diagram

The development of the BFS graph kernel implementations presented in Chapters 5 and 6 was created using the development framework. A description of each one of the classes utilised in those implementations is listed below. The class diagram containing those classes is illustrated in Figure 4.10.

**Graph** This is the base class and it contains the graph and the Graph 500 functions required to measure and validate the experiments. This class accesses those functions that are part of the Graph 500 suite and creates an interface to allow the graph kernel implementations to access them. These functions are listed in the Graph class as the following methods: `generate()`, `load()` and `verify()`. The graph is loaded and represented as a CSR member of the class and is shared among the derived classes of the graph kernel implementations.

**Bitmap** This class holds an array of 32-bits integers, where each bit represents two states of an element by setting either zero or one. The interface of the class consists of basic methods to initialise, reset, swap and free the bitmap arrays. Furthermore, it contains the methods to set and get a specific bit in the array by passing the position in the bitmap array. This class is used by other classes as internal data structure including TD, TDSIMD, Hybrid and HybridSIMD.

**TD** This class is an implementation of the BFS graph kernel. This implementation uses the parallel *top-down* version of the algorithm and it is introduced in Section 5.3.

**TDSIMD** This class holds an implementation of the BFS graph kernel search. This implementation uses the vectorised version of the *top-down* using SIMD programming and is described in Section 5.4.

**Hybrid** The hybrid class is an implementation of the BFS graph kernel, which is the state-of-the-art implementation. It consists of the combination of the *top-down* version of the algorithm with a *bottom-up* approach. A thorough explanation of this implementation is located in Section 6.1.
HybridSIMD This is the hybrid class for the BFS graph kernel using SIMD programming. A complete description of this implementation is introduced in Section 6.3.

Figure 4.10: Class diagram of the BFS graph kernel development.

4.3 Summary

This chapter reviews the Graph 500 benchmark including the graph generator, the experimental setup and the details of the output performance information. The input parameters for the graph generator are the $SCALE$, $edgefactor$ and the probabilities ($A$, $B$, $C$ and $D$) to distribute the edges along the graph. The
The experimental setup consists of 64 executions of the BFS implementation with different starting vertex. After each execution the output BFS tree is validated. The metric utilized to measure the implementation performance is TEPS. Along the 64 executions metrics such as: number of edges, time and TEPS are collected and output statistics are presented. The harmonic mean is the average used across multiple architectures to present results. Despite the fact that the harmonic mean can be an inaccurate metric because some Graph 500 graphs include unconnected vertices, this metric still is the one used for the results presented in Chapter 5 and 6.
Chapter 5

Top-Down BFS Vectorisation

In Chapter 2, the conventional layer-synchronous BFS parallel algorithm was introduced, which consists of traversing the graph by layers. Processing vertices in each layer to explore the adjacency list of vertices related to them (parent-child relation) is called top-down approach. This is in contrast to the bottom-up approach, which is based on processing non-visited vertices to find their parents in the frontier (child-parent relation). A combination of both approaches top-down and bottom-up, called hybrid, is the state-of-the-art BFS algorithm [BAP12]. This chapter presents the process of vectorising the parallel top-down BFS algorithm, whereas the description of vectorising the hybrid approach is presented in Chapter 6.

Vectorising the top-down BFS algorithm aims to exploit the finer-grain parallelism, located in the adjacency list exploration, by using SIMD instructions to access the vector units of the Xeon Phi architecture. A reminder of the parallel BFS algorithm is provided in Section 5.1 and the data structures used are presented in Section 5.2. Two versions of the BFS were implemented, non-SIMD and SIMD version, to compare the performance benefit of using the vector units. The non-SIMD version is introduced in Section 5.3 and it is the baseline implementation that does not use any SIMD. Section 5.4 presents a full description of the SIMD version including the design of the algorithm and the implementation on the Xeon Phi analyses the code regions where vectorisation is applied Section 5.5. Two sets of experiments were conducted, the comparison between the SIMD and the non-SIMD versions of the top-down BFS algorithm and an exploration of thread affinity. The experimental setup is described in Section 5.6, followed by an analysis of both experiments in Sections 5.7 and 5.8 respectively. For the SIMD
version, a comparison with the state-of-the-art is given based on the results presented by Gao et al. in [TYG13]. Since the source code of their implementation is not available, a detailed comparison cannot be made. Finally, the conclusions are presented in Section 5.9. A summary of this chapter is given in Section 5.10.

5.1 Parallel Top-Down Breadth-First Search

The parallel layer-synchronous BFS algorithm, introduced in Section 2.2.2, consists of exploring the graph by layers. This algorithm shows two levels of parallelism, coarse-grain and fine-grain. The former is in the outer loop, when the vertices in the frontier are explored (the input list). The latter is in the inner loop, when each of those vertices is explored for any non-visited adjacent vertex. The relation between each vertex with those in the adjacency list is a parent-child relation, also known as top-down. This nested loop is presented in Algorithm 4, indicating where the multithread race conditions take place. The concept of multithread race conditions is introduced in Section 3.2.1 and the description of the race conditions in this algorithm is further in Section 5.3.

Algorithm 4 parallel-top-down(in, Adj, vis, out, P)

1: for parallel $u \in$ in do ▷ coarse-grain parallelism
2:     for parallel $v \in$ Adj[$u$] do ▷ fine-grain parallelism
3:       if vis.Test($v$) then
4:         vis.Set($v$)
5:         out.add($v$)
6:         $P[u] = v$
7:       end if
8:     end for
9: end for

The non-SIMD and the SIMD BFS implementations are based on Algorithm 4. The non-SIMD version uses multi-threading for the outer loop and copes with the race conditions by using atomic operations. Similarly, in the SIMD version the coarse-grain parallelism is exploited by multi-threading but not only, the fine-grain is also exploited by using the vector unit of the Intel Xeon Phi architecture introduced in Chapter 3. Both implementations are described in the Sections 5.3 and 5.4.
5.2 Data Structures

The parallel BFS implementation is based on the implementation, `bfs_replicated_csc`, given in the Graph 500 source code [WL]. The graph is efficiently represented by a Compressed Sparse Row (CSR) matrix format, which is composed by two 32-bits integer arrays: `rows` and `colstarts`. The `rows` array contains the adjacency list of every vertex and the `colstarts` stores the `start` and the `end` indexes of every vertex pointing to the `rows` array. The CSR is described in the Graph 500 graph generator in Section 4.1.1. In practice, the use of the CSR is when the adjacency list is explored; see line 2 in Algorithm 4.

Moreover, the data structures used for the frontier, the output queue and the visited arrays are bitmap arrays, and an integer array for the predecessor array. A bitmap is a mapping from a set of items to bits with values either zero or one. By having vertices represented by a bitmap, the working set size can be reduced significantly [APPB10]. For example, an array that holds 1,048,576 ($2^{20}$) vertices represented by 4-byte integers would require 4MB. By using bitmaps this memory storage can be reduced significantly to 131,072 bytes (128 KB).

Figure 5.1 illustrates the mapping between an array of integers to bits. The upper array is an array of integers, where every index corresponds to a vertex in the array. The valid values of each element in this array are either zero or one and the length of the array is the total number of vertices. In this example, only vertex 28 and 30 are set to one. The array at the bottom is the bitmap array and its length is the total number of vertices divided by the number of bits of an integer (32 bits). Every integer in the bitmap array represents the status (0,
1) of 32 vertices. Thus, the same vertices 28 and 30 are set to one but both are located in the first index of the integer array.

### 5.3 The non-SIMD Breadth-First Search

The term *non-SIMD* refers to the simple implementation of the parallel BFS algorithm on the Xeon Phi without using the vector unit. Hence, based on Algorithm 4, the parallelisation will be using multiple threads at the outer loop. However, graphs present data dependencies when the graph is being traversed. This data dependencies cause that multiple threads might end up trying to explore the same vertex. In practice, within the nested loop, two race conditions between threads can be identified.

The first race condition happens when a vertex $v$ if it has been *visited* previously. Multiple threads might test on the same vertex at the same time. Figure 5.2 illustrates this race condition. Two threads, A and B, are trying to update vertex 5, which is a child of both vertex 2 and 3. While this could end up in redundant work when the status of the vertex and the queue are updated, the major impact is in the predecessor list, where the parent of vertex 5 can be set to either 2 or 3. However, this is a *benign race condition* since the correctness of the algorithm is not affected. It means that different correct output BFS trees can be generated.

![Figure 5.2: Example of data benign race condition.](image)

The second race condition is more critical and is related to the memory size that the architecture can guarantee to keep data consistent. In other words, the memory unit that keeps data without being corrupted after its modification (read
and write operations). For instance, Intel 64 and IA-32 architectures guarantee to keep data consistent for any of the following data sizes: 8, 16, 32 or 64-bits during basic memory operations (read and write) [Int09]. As part of the x86 family processors, the Xeon Phi architecture follows that same instruction set, which impacts directly on the bitmap 32-bits arrays utilised to store the visited, the frontier (input list) and output queue (output list). The effect leads into a race condition when different threads try to update multiple bit values in the same word (32-bit). An example of this case is shown in Figure 5.3, where vertices 5 and 9 are updated by different threads but their location in the bit array is in the same word (32 bits integer). As a result, one of them is invalidated by the other, which leads into an incorrect output BFS tree.

\[\text{Algorithm 5: Top-down BFS vectorisation with atomic operations} \]

Despite the first condition being benign, the second race condition leads to incorrect results. By using atomic operations, both of them can be eliminated. The atomic operations are set during the modification of the visited array, and the output queue in the parallel BFS algorithm. Algorithm 5 shows the pseudocode after adding atomic operations to the parallel top-down BFS algorithm. These changes follow the two optimisations proposed by Agarwal et al. [APPB10].

First, the output queue contains all the vertices of the frontier of the next layer, all of them are set as visited. Thus, the output queue can be seen as a subset of the visited array, which holds all the visited vertices in the graph. By having the output queue as a subset of the visited array, it is possible to only update the output queue to further merge them at the end of exploring the layer.
as can be seen in line 10. Hence, by only updating the output queue, the atomic operations are reduced because the visited array will no be longer updated with atomic operations. However, still both bitmaps (the visited and the output) have to be tested because they comprise the total set of visited vertices. This can be seen in the testing condition, line 3, as any vertex that is not set as visited either in the visited array or in the output queue.

The atomic operation is in the function call out.LockedReadSet(v) in line 4. It uses the CPU instruction _sync_fetch_and_or that allows to modify atomically the output queue.

Algorithm 5 atomic parallel top-down-step\((in, Adj, vis, out, P)\)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for parallel (u \in \text{in}) do</td>
</tr>
<tr>
<td>2</td>
<td>for (v \in \text{Adj}[u]) do</td>
</tr>
<tr>
<td>3</td>
<td>if (\text{vis.Test}(v) = 0) AND (\text{out.Test}(v) = 0) then</td>
</tr>
<tr>
<td>4</td>
<td>if (\text{out.LockedReadSet}(v) = 0) then</td>
</tr>
<tr>
<td>5</td>
<td>(P[u] = v)</td>
</tr>
<tr>
<td>6</td>
<td>end if</td>
</tr>
<tr>
<td>7</td>
<td>end if</td>
</tr>
<tr>
<td>8</td>
<td>end for</td>
</tr>
<tr>
<td>9</td>
<td>end for</td>
</tr>
<tr>
<td>10</td>
<td>(vis = vis \cup out)</td>
</tr>
</tbody>
</table>

5.4 The SIMD Breadth-First Search

Vectorisation in the parallel top-down BFS algorithm can be exploited at fine-grain parallelism in the inner loop of Algorithm 4, during the adjacency list exploration. Specifically, the vectorisation of the adjacency list consists of processing multiple vertices at the same time. Despite automatic vectorisation, which is introduced in Section 3.3, being a useful tool that allows the compiler to enable vectorisation, its functionality is limited when non-contiguous memory access and data dependencies are present [Int12e]. Unfortunately, both characteristics are present in the BFS algorithm, which makes it not to be a good candidate for automatic vectorisation.

Algorithm 6 shows the specific points where these two obstacles for automatic vectorisation appear. First, the compiler would try to vectorise the conditional statement in line 2, which involves looping through all the indexes but only executing the ones for which the condition is true. However, having set the
adjacent vertices randomly during the graph generation leads to having non-contiguous memory accesses of the visited array. The second obstacle is the data dependency present in line 5, during the setting of the vertex \( v \) as parent of \( u \) in the predecessor list. Since multiple iterations are executed at the same time, different adjacent vertices can be assigned to the same parent in the predecessor list, which can lead to incorrect results. Both problems make it difficult for the compiler to activate automatic vectorisation and even guiding the compiler by using \textit{pragma} hints would require several changes in the source code. These changes would include transforming the data to a friendly vector way, so that the automatic vectorisation can be activated. However, these transformations require functions to gather data into a contiguous location in memory and to scatter it back after being processed.

\textbf{Algorithm 6} Adjacency List Automatic Vectorisation Adj\((v, in, Adj, out, P)\)

1: \textbf{for parallel} \( v \in Adj[u] \) \textbf{do} \\
2: \hspace{1em} \textbf{if} \ vis.Test(\( v \)) \textbf{then} \hspace{1em} >> \text{irregular data access} \\
3: \hspace{2em} vis.Set(\( v \)) \\
4: \hspace{2em} out.add(\( v \)) \\
5: \hspace{2em} P[u] = \( v \) \\
6: \hspace{1em} \textbf{end if} \\
7: \textbf{end for} \\

Due to the obstacles that the BFS algorithm presents for \textit{automatic} vectorisation, this work investigates the vectorisation using explicit intrinsic functions\(^1\), previously introduced in Section 3.4.2, including a hardware optimisation for the \textit{scatter} and \textit{gather} functions. Intel C/C++ and GCC contain intrinsic functions to map directly SIMD functions. Specifically, the intrinsic functions utilised in this work are part of the Intel AVX-512 instruction set [Rah12]. These intrinsic functions allow control over the vector unit, which leads to having to be aware of different requirements and optimisations including, \textit{alignment}, \textit{peel} and \textit{remainder} loops and \textit{prefetching}. The details of these requirements are presented further in Section 5.4.2.

\(^1\)Intrinsic functions are a set of functions that are handled specially by the compiler
5.4.1 Description of the Algorithm

This section describes the mapping of vectorising the BFS algorithm on the Xeon Phi. Based on Algorithm 6, the vectorisation of the adjacency list consists of exploring multiple vertices at a time. The number of vertices processed in one SIMD instruction depends on the vector unit width and the data type of the element. Since the Xeon Phi vector unit width is 512 bits, it is capable of processing up to 16 (32-bit) elements at a time as mentioned in the description of the architecture in Section 3.3. Hence, the adjacency list exploration, instead of exploring one vertex at a time, it could be possible for one thread, to explore 16 (32-bit integer) vertices simultaneously.

Vectorising the adjacency list exploration involves three main SIMD steps. Firstly, a sequence of vertices in the adjacency list are loaded into the vector unit, which can hold 16 (32-bit) vertices. Secondly, all the loaded vertices are filtered by using the visited array and the output queue bitmaps to find the ones that have not been visited yet either in previous layers (visited array) or in the current layer (output queue). Finally, for the non-visited vertices the result is set back to the predecessor array $P$ and the output queue $out$. Figure 5.4 shows an example of the vectorisation of the adjacency list exploration with a vector registers of width 512 bits after loading the data for a set of vertices.

The specific values in the visited and output queue bitmap arrays are loaded into the vector unit by using the SIMD gather instructions since the vertices in the adjacency list will generally not be contiguous in the bitmap arrays. The scatter and the gather operations of the Intel Xeon Phi are described in Section 3.3. Figure 5.5 illustrates both instructions to update the visited array. Since the visited array is a bitmap array, every vertex has an index, from 1 to the total number of vertices in the graph, that is mapped into a word index and an offset index in the bitmap array. This mapping is handled by two steps, first the word index is calculated by dividing the vertex index $b$, by the word size (32-bit). The result of the division is the word index and the remainder is the offset index. The division can be calculated using the division vector instruction and the remainder by using the vector bit shifting operations to get the bit value of the vertex.

Once the specific values of the visited array and the output queue are loaded into the vector unit, a filtering process is applied. To find all the vertices that have not been either visited previously in upper layers or put into the output queue recently in the current layer, two logical operations (OR and NOT) are used to...
CHAPTER 5. TOP-DOWN BFS VECTORISATION

Figure 5.4: Example of vectorising the adjacency list exploration.

Figure 5.5: Gather and scatter operations, to load non-contiguous visited array.

create a vector mask. The OR operation will find the union of the vertices that have already been visited and the ones that have been put into the output queue. By applying the NOT logical operation, we can find all the vertices that have not been visited or set into the output queue. Afterwards, when the result is
scattered into the predecessor array and the output queue, only those indexes
that have one as bit value in the mask are updated.

During the update of the visited and the output bitmap arrays, the same
race condition explained in Section 5.3 appears. However, it is not possible to
utilise atomic operations during vectorisation, mainly because multiple elements
are processed at the same time [TYG13]. Trying to update the same 32-bits word
at the same time could lead to an incorrect result due to the bit race condition
described in Section 5.3. Following the method presented by Gao et al. [TYG13],
an extra step was added to the algorithm called the restoration process. This
helps to keep the output and the visited bitmaps consistent for processing the
next layer. Algorithm 7 presents the pseudocode of the restoration process, which
is discussed next.

**Restoration Process**

The restoration process of the algorithm consists of finding all the potentially
corrupted words in the output bitmap array that were updated. If a bit race
condition happened, the output array should have at least one bit set in the
word since, if more than one thread writes the same word, one of them will win
the race and update the word. Since the predecessor list is an array of integers,
it does not present the bit race condition and can be used to fix the corrupted
output array as follows. First the vertices that were updated in the predecessor
list are identified by setting their parent to a negative number rather than to the
number of the parent vertex. To identify these updated vertices, the parent is set
to a negative value by subtracting the total number of vertices $V$ from its value.
Second, all the words in the output bitmap array are scanned looking for non
zero words. In those words are located the bits of the vertices that are corrupted
and need to be fixed. Each of the 32 bits in the word are step through to look
for the corresponding vertices, in lines 20 and 21 of Algorithm 7, that have been
set to a negative number in the predecessor list. These vertices are restored by
setting their corresponding bit in the output bitmap and adding back the number
of nodes in the predecessor list. As a result, the visited bitmap array is updated
consistently.

Although the restoration process implies extra work, it solves the bit race
condition without having to use atomic operations while keeping correctness and
even more important allows vectorisation. Algorithm 7 shows the complete pseudocode of the BFS algorithm containing the restoration process to cope with the bitmap race condition in lines 15 to 30. Bitmap operations used by the visited, input and output arrays are: InitBitmap(), SetBit(n), GetBit(n), TestBit(n) and bit2vertex(n), where n is the bit position.

Algorithm 7 Parallel BFS without bit race conditions.

Initialise: in.InitBitmap() out.InitBitmap() vis.InitBitmap()
1: for parallel vertex \( u \in V(G) - s \) do
2: \( P[u] = \infty \)
3: end for
4: in.SetBit(s)
5: vis.SetBit(s)
6: \( P[s] = s \)
7: while \( in \neq 0 \) do
8: for parallel \( u \in in \) do
9: for parallel \( v \in Adj[u] \) do
10: if \( v \notin \) (vis.TestBit(v) OR out.TestBit(v)) then
11: out.SetBit(v)
12: \( P[v] = u - V \) \( \triangleright V \) is the total number of vertices in the graph
13: end if
14: end for
15: for parallel \( w \in out \) do \( \triangleright \) Restoration process
16: if \( w \neq 0 \) then
17: for \( b \in w \) do \( \triangleright \) iterate through every bit in w
18: vertex = bit2vertex(b)
19: if \( P[vertex] < 0 \) then
20: out.SetBit(vertex)
21: vis.SetBit(vertex)
22: \( P[vertex] = P[vertex] + V \)
23: end if
24: end for
25: end if
26: end for
27: end for
28: swap(in, out)
29: \( out \leftarrow 0 \)
30: end while
5.4.2 Xeon Phi Optimisations

The optimisation of the vectorisation of the BFS top-down algorithm involves the understanding of crucial factors such as cache-oriented memory address alignment, loop vectorisation, masking and prefetching [Int12e]. These are described next.

Data alignment The intrinsic function `mm_malloc` [Int12e] was utilised to align the `rows` array implementing the adjacency information. This data alignment allows the vector unit to have more efficient access to the memory and it is described in Section 3.3. However, due to the way the `rows` array is built, it might lead to some less-than-full-vector loop vectorisation inefficiencies, such as the peel and the remainder loops [TSP+13], which are discussed next, caused non-aligned boundary accesses.

Peel and remainder loops As illustrated in Figure 5.6, the peel loop is the sequence of contiguous elements for which the start index in the array does not match with an aligned boundary of the array. At the other end, the remainder loop is the sequence of the last elements, the tail, that do not fit on an aligned boundary. Both cases imply an extra processing step because they cannot be computed as a complete 16 elements vector. There are different approaches to cope with the implications of having less-than-full-vector loops including padding, sequential processing of peel and remainder loops, and the use of vector masks. Figure 5.6 illustrates three approaches.

1. **Padding**: consists of filling the array with meaningless bytes between the end of the last chunk of the adjacency list and the start of the next set. In this way, the start of every chunk of the adjacency list is aligned to a specific byte boundary. Despite this being an effective way to get rid of the peel loops, the remainder loops still exist and padding requires the allocation of more memory space. Therefore, the `rows` array needs to be preprocessed because a copy to the new memory allocation is necessary. This is potentially expensive in both time and memory.

2. **Sequential peel and remainder loops** is another approach to handle the peel and the remainder loops without having to increase the memory allocation size by processing them sequentially. So, instead of consuming time
preparing these elements to be used by the vector unit, they are computed sequentially without using it.

3. **Vector masks**: In this approach, processing the peel and the remainder loops implemented is by using vector masks previously introduced in Section 3.3. In this case, a vector mask is generated to filter what are elements in the peel or remainder elements to be processed within the aligned 64 bytes that contain the valid elements. However, to calculate the vector mask it is necessary to pre-process the rows array to know where the peel and the remainder loops are. Although having vector masks to process the peel and the remainder loops could be more efficient than sequential processing, a preprocessing process is needed to be able to calculate them. The implementation, proposed by this thesis used pre-calculated vector masks for the peel and the remainder loops.

![Diagram](image)

**Figure 5.6**: Illustration of the techniques utilised to cope with less-than-full-vector loops for the 64B data alignment of the Xeon Phi.
Prefetching  Due to the irregular data access patterns that the BFS algorithm shows, software prefetching can help. In addition, working with large graph sizes might result in having large numbers of L2 cache misses, and thus poor performance. This is a technique part of the Xeon Phi architecture, introduced in Section 3.2.1. Prefetch distance is a metric that hints to the compiler the right number of cycles to load data ahead of it use. Finding the right distance is crucial to gain performance [LHSS09]. A recommended distance should be one similar to memory latency [BAYT04]. Rather than calculate a precise distance, and based on the work in [JHL+15], in the adjacency list exploration the rows array for the vertices that will be processed in the next iteration are prefetched with the prefetch distance being set by experimenting. Also, prefetching intrinsic functions were used to load data to the vector unit by setting a hint that indicates which memory will be prefetched to either L1 cache (MM_HINT_T0) or to L2 cache (MM_HINT_T1), see Listing 2 for code examples.

Finally, the vectorisation of the adjacency list exploration consists of splitting the list into chunks of 16 (32 bit) elements. The peel and the remainder loops are considered special cases because vertices need to be filtered according to the pre-calculated mask. Listing 2 shows an example of the source code of the vectorisation of the adjacency list for a full-vector, using optimisations for alignment, vector masks and prefetching.

Specifically, the code follows the three steps described in Section 5.4.1, which are loading the adjacency list, filtering non-visited vertices and setting the values back to memory. Additionally, an intermediate operation is needed due to the discrepancy between the indexes of the 16 input vertex list (32-bit integers) and the bitmap arrays (bits) in the step 2: filtering the unvisited vertices in the visited array and the output queue. This step is implemented by getting the word and the bit offset of each element in the adjacency list. The words vector is used to gather all the words to be updated from the visited array and the output queue. The bit offset vector is used to create a mask to filter the specific bit values by shifting it to the left. The words and the bit offsets are used to generate a vector mask that allows to filter the visited and the output queue bitmap arrays, respectively.

Figure 5.7 shows different results for the BFS top-down algorithm on the Xeon Phi. The experiments involved three implementations, including the SIMD
Listing 2: Source code of the SIMD version of the top-down BFS algorithm.

without optimisations (“SIMD - no opt”), the SIMD BFS algorithm plus align-ment and mask optimisations (“align+ masks”) and the one with prefetching (“Prefetching”). As can be seen performance was increased after applying prefetching because it reduces the number of memory accesses resulting in a maximum number of increment in performance of approximate 300 MTEPS, which is around
Figure 5.7: BFS experimental optimisations results for a graph of SCALE 20 and edgefactor=16.

37% speedup. Moreover, the drop in performance for 240 threads is because despite the maximum number of threads theoretically is 240, one core is fully dedicated for internal management, reducing the number of threads to 236. Hence, in practice the maximum number of threads is 236.

**Thread affinity** Thread affinity is a feature in architectures that enables the pinning of threads to specific logical cores on the physical cores, this feature is introduced in Section 3.4.1. Regardless of the affinity strategy used, if the number of threads is equal to the number of logical cores available, the affinity is equivalent to compact (i.e. four threads per physical core) and resource sharing is at a maximum.

Since the effect of thread affinity on performance is application dependent (for example, sharing cached data between threads may be beneficial but sharing memory bandwidth may not, depending on the application), various experiments were conducted to determine which strategy works best for the new BFS algorithm. By experimenting it was found that balanced affinity was generally better and so it was used for performing the experiments to gather the results shown in Figure 5.7. To illustrate the impact of thread affinity, following the methodology presented in [RNPL15], an experiment related to thread affinity is further described in Section 5.6.
5.5 Which layers are vectorised?

As discussed in Section 4.1.1, large and sparse graphs often present the small-world graph properties of having small diameter and skewed degree distribution [CZF04]. In this thesis, a synthetic graph generator based on recursive matrices (RMAT) is used to create the input small-world large graphs described in Section 4.1.1. The RMAT graph size is defined by two input values: the SCALE and the edgefactor. The total number of vertices in the graph is calculated by $2^{SCALE}$ and the number of edges generated by $2^{SCALE} \times \text{edgefactor}$. The graph structure is crucial because it can help to improve performance by exploiting the architecture resources efficiently, such as the vector unit on the Xeon Phi. Table 5.1 shows the number of input vertices, the number of edges and the traversed vertices by the BFS top-down algorithm per layer, for a graph of 1,048,576 million vertices (RMAT graph with SCALE 20 and edgefactor 16), choosing the starting vertex randomly. This data is typical.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Vertices</th>
<th>Edges</th>
<th>Visited</th>
<th>Visited/Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>12</td>
<td>12</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>21,892</td>
<td>18,122</td>
<td>0.83</td>
</tr>
<tr>
<td>3</td>
<td>18,122</td>
<td>13,547,462</td>
<td>540,575</td>
<td>0.03</td>
</tr>
<tr>
<td>4</td>
<td>540,575</td>
<td>17,626,910</td>
<td>100,874</td>
<td>0.005</td>
</tr>
<tr>
<td>5</td>
<td>100,874</td>
<td>150,698</td>
<td>486</td>
<td>0.003</td>
</tr>
<tr>
<td>6</td>
<td>486</td>
<td>490</td>
<td>4</td>
<td>0.008</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Traversed vertices per layer for a 1,048,576 million vertices graph, SCALE 20 and edgefactor 16.

As can be seen in the table, the number of input vertices per layer increase with the number of layer until the middle layer is reached and then starts to decrease. The diameter of the graph (the minimum number of edges to reach one vertex from another) is 7, which is reflected in having 7 traversed layers. Furthermore, the number of edges per layer varies according to the edgefactor, which is used to distribute the edges per vertex as described in Section 4.1.1. Both graph characteristics, diameter and vertex degree are key to decide what is the best way to increase the vector unit usage. At the beginning of traversing a graph, the total number of vertices minus the starting vertex are non-visited, this implies that the first layers most of the vertices in the frontier are set as visited. Assuming that most of the vertices are traversed in the first layers, the vectorised
SIMD BFS top-down algorithm is used only for the first two layers because this is where the ratio between the number of visited vertices and the number of edges is higher. The rest of the layers are processed using the non-SIMD *top-down* algorithm presented in Section 5.3.

## 5.6 Experimental Setup

This section presents the experimental settings used in the experiments to evaluate and analyse the performance of the vectorisation of the top-down BFS. The experiments are divided in two parts. The first part consists of a comparison between the non-SIMD and the SIMD version of the top-down BFS algorithm. The second one, is related to the exploration of thread affinity. The experimental settings of both experiments are in Section 5.6.1 and 5.6.2 respectively.

**Hardware platform** Section 3.4.3 presents a detailed summary of the hardware and compilation settings for these experiments.

### 5.6.1 Top-Down Settings

The experimental results reported are from a sequence of sets of 64 executions (one for each randomly selected start vertex) in which the following parameters are: the number of threads and the graph SCALE factor (the edgelfactor is fixed at 16). The number of threads were chosen in the range of 1 to 240 as: 1, 2, 8, 16, 32, 40, 64, 100, 180, 200, 210, 228, 232 and 240 (the maximum number of one thread per logical core). In order to compare with the state-of-the-art, Gao et al. [TYG13], the SCALE were set to 18, 19 and 20 because those graph sizes is where there is an interesting gap between the non-SIMD BFS implementation and the SIMD, which can be seen in Figure 9 in [GLZS14].

### 5.6.2 Thread Affinity Settings

The effect of affinity on the BFS algorithm is demonstrated by and experiment running a 48 thread version manually controlling the affinity to achieve one, two, three and four threads per core (1T/core, 2T/core, 3T/core and 4T/core), thus using 48 cores with one thread per core down to 12 cores with four threads
per core. The thread affinity is controlled through the environment variable \texttt{KMP\_AFFINITY}.

### 5.7 Top-Down Vectorisation Analysis

Figure 5.8 shows the experimental results for the non-SIMD version and the SIMD optimised version for three different graphs sizes, SCALE (18, 19 and 20), and edgefactor 16. Both versions show similar scalability but the SIMD version is approximately 50 MTEPS, 150 MTEPS and 300 MTEPS faster than the non-SIMD version for scales 18, 19 and 20 respectively, with a maximum performance speedup of around 29\% for the graph size $SCALE = 20$. In this context speedup refers to an increase of the speed in the performance of the algorithm and not to the speed-up concept of Ambdahl’s law introduced in section ?? . The speedup was calculated taking the non-SIMD version as a baseline and the difference in performance is given in terms of percentage. However, it can be seen that as the number of threads increases, the rate of increase of the number of TEPS decreases. This effect is a result of hardware multithreading and is discussed in Section 5.8. Finally, the variation in performance between the range from 200 to 236 threads might be caused due to workload imbalance during the exploration of the vertices in each layer since, as the number of threads increases, the chances of vertices processed by a thread having an uneven number of adjacent vertices increases because each thread has fewer vertices to process. The results show that the maximum number of TEPS is above 1 GigaTEP. This is higher than the best published result in [GLZS14] (the dashed line in Figure 5.8(c) is based on the results in their paper), which gives approximately 800 MTEPS, achieving an approximate maximum speedup of 37\% over their native vectorised BFS algorithm for the same graph size (SCALE 20) - the highest top-down BFS performance figure on a similar Xeon Phi that we have found in the literature. The Xeon Phi used by [GLZS14] has 57 cores running 1.1 GHz, each core with four hardware threads. A 32 KB L1 cache and 512 KB L2 cache per core. Then, the maximum number of threads in their results is not clear stated but presumably is 228 threads. The model of the Xeon Phi used in this thesis (5110P) has 60 cores and the same characteristics as theirs.
5.8 Thread Affinity Analysis

Table 5.2 shows the result of running with 48 threads but varying the number of threads pinned per physical core manually using thread affinity as described in Section 3.4.1.

<table>
<thead>
<tr>
<th>#Threads</th>
<th>Thread Affinity</th>
<th>Cores</th>
<th>TEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>1T/C</td>
<td>48</td>
<td>4.69E+08</td>
</tr>
<tr>
<td></td>
<td>2T/C</td>
<td>24</td>
<td>2.67E+08</td>
</tr>
<tr>
<td></td>
<td>3T/C</td>
<td>16</td>
<td>1.89E+08</td>
</tr>
<tr>
<td></td>
<td>4T/C</td>
<td>12</td>
<td>1.42E+08</td>
</tr>
</tbody>
</table>

Table 5.2: Performance SIMD version by setting thread affinity for a graph size of $SCALE = 20$ and $edgefactor = 16$.

These results show the detrimental effects of over-populating the cores for the BFS. The TEPS obtained with one and two threads per core are significantly higher than the TEPS with three and four threads per core. This reduction in the TEPS rate as the number of threads per core increase is the key driver to the changes in slope observed in Figure 5.8 occurring around 60, 120 and 180 cores when the number of threads per core has to increase as more threads are used. At these points, each thread’s exclusive access to cache space decreases as does its share of memory bandwidth. Despite these changes in slope, the performance of the both the simd and non-simd BFS *top-down* algorithm continues to scale. So, by using fully populated cores (59), each one with the maximum number of threads (4), the number of TEPS for 236 logical threads is the fastest. Beyond 236 thread, threads are placed on the final core in the Xeon Phi used which is reserved for the operating system on the Phi and is therefore continually busy, resulting in a dramatic fall in performance.

In the future, it might be possible to exploit this behaviour by under-populating cores with threads performing BFS and to make use of so-called *helper* threads running on a core to assist with, for example, prefetching to help hide memory latency [KST11].
5.9 Conclusions and Future Work

The contributions of this chapter are, first, the development of an improved OpenMP parallel, highly vectorised SIMD version of the BFS using vector intrinsics and successfully exploiting data alignment and prefetching. This new implementation achieves a higher number of TEPS than previous published results, Gao et. al. [GLZS14], for the same type of Xeon Phi with an approximate speedup of 37% for a one million vertices graph ($SCALE = 20, edge\_factor = 16$). The second contribution is an investigation of the impact of the thread affinity mapping and simultaneous threading on performance on an underpopulated system, a topic under researched in the literature. The experiments shown that by utilising four threads per core (fully utilised), the performance drops dramatically down for the third and fourth threads. This suggests future possibilities to take advantage of under-populating the cores and using the spare capacity to improve latency hiding through the use of helper threads, for example, to complement prefetching.

5.10 Summary

In this chapter, the vectorisation and performance optimisation issues of the top-down BFS algorithm on the Xeon Phi was presented. In particular, the BFS top-down algorithm without (bit-wise) race conditions was studied to apply vectorisation. Furthermore, the details of the optimisations used for vectorisations were presented including data alignment, prefetching, masking and thread affinity. A comparison between the non-vectorised version and the vectorised version of the top-down BFS algorithm was conducted. The results demonstrate better speedup of 37% over a similar version of the top-down BFS algorithm exploiting vectorisation with the Xeon Phi. Finally, the version of the top-down BFS algorithm presented in this chapter is used in Chapter 6 for the hybrid BFS implementation, where the lessons learned from vectorisation are also applied.
Figure 5.8: BFS non-SIMD and SIMD experimental results for SCALE values of 18, 19, 20 and edge factor = 16. In Figure (c) the dashed line represents the best MTEPS reported in [GLZS14] with a maximum number of threads of 228.
Chapter 6

The Hybrid BFS Vectorisation

This chapter presents the vectorisation of the hybrid BFS algorithm, which is the combination of the top-down algorithm of the BFS, previously introduced in Chapter 5, and the bottom-up approach. The main difference between these approaches lies in the fact that while the top-down uses a parent-child relation, the bottom-up approach traverses the graph following a child-parent relation. Firstly, the bottom-up algorithm is introduced in Section 6.1. Section 6.2 describes the hybrid BFS algorithm which switches between both approaches by using an online heuristic described in Section 6.2. The vectorisation of the hybrid algorithm involves utilising the vectorised version of both approaches, the SIMD top-down and the SIMD bottom-up. The vectorisation of the top-down BFS algorithm has been explained in Chapter 5. Section 6.3 describes the vectorisation process of the bottom-up but the vectorisation of both approaches, top-down and bottom-up are used for the results presented.

Moreover, a methodical analysis of the vectorisation of the bottom-up algorithm was conducted, where three bottom-up implementations were evaluated. The experimental setup is described in Section 6.4, followed by the performance analysis in Section 6.5. Furthermore, to see the benefits of using vectorisation, Section 6.6 presents the results of the comparison between the hybrid BFS using the non-SIMD version of the bottom-up and the one using the vectorised bottom-up. Both versions were compared against the state-of-the-art hybrid BFS introduced by Gao et al. [GLZS14]. Finally, the conclusions and future work are presented in Section 6.7.
6.1 Bottom-Up Breadth-First Search

The bottom-up BFS algorithm was introduced by Beamer et al. [BAP12] and it differs from the top-down in that instead of stepping through the vertices in the frontier, it iterates through the non-visited vertices within the layer. Each vertex looks for the existence of a parent in their adjacency list.

A valid parent is one which is in the frontier of the current layer being processed. Once a parent is found, the current non-visited vertex is set as visited, put into the output queue for processing in the next layer, set in the predecessor list and the process then continues looking for a parent for the next non-visited vertex. Algorithm 8 shows the pseudocode for the bottom-up step in the BFS algorithm.

Algorithm 8 bottom-up-noSIMD(in,out, vis, P)

1: for parallel v ∈ vis do ▷ All the non-visited vertices
2:    for n ∈ Adj[v] do ▷ Iterating the adjacency list
3:       if n ∈ in then ▷ A parent was found
4:          vis.Set(v)
5:          out.Set(v)
6:          P[v] = n
7:       end if
8:    end for
9:  end for

Notice, that the naive parallelisation of the bottom-up algorithm is indicated in line 1 by using parallel in the for loop that iterates through the non-visited vertices. The source code of this algorithm can be seen in Listing 5 of Appendix C.

6.2 Hybrid Breadth-First Search

The bottom-up approach differs from the top-down approach in the direction of processing vertices and which frontier to process. Firstly, the bottom-up, instead of exploring vertices in the frontier looking down for a parent-child relation, searches up for a child-parent relation when the adjacency list of a non-visited vertex is traversed. Secondly, the input list of vertices for each algorithm is
different. The top-down receives as input, the list of vertices to be processed in the layer (the current frontier), while the bottom-up receives the list of the remaining non-visited vertices in the graph. To illustrate the differences, two examples are shown in Figure 6.1. In the top-down traversal, vertex three is being explored. Despite vertex 3 having to iterate through all 16 elements in its adjacency list, only vertices 4, 25 and 32 are marked as visited (highlighted vertices) because the others have already been marked as visited, implying an inefficiency in the process since only 3 edges out of 16 are being processed. In contrast, in the bottom-up traversing, the process of iterating through the adjacency list stops when the parent of the non-visited vertex 3 is found (vertex 32), without having to iterate through all the adjacency list.

Figure 6.1: Example of the bottom-up adjacency list exploration.

To summarise, the efficiency of both approaches relies on two factors:

1. The number of input vertices. In the case of the top-down, the input is the vertices in the frontier of the current layer and the non-visited vertices for the bottom-up.

2. The number of vertices in the adjacency list. The top-down iterates through all the list, whereas the bottom-up stops once a parent is found.

The computational complexity of traversing a graph, with $V$ vertices and $E$ edges using the top-down approach is $O(V + E)$ [CLRS09]. The total lengths of all of the adjacency list is $\Theta(E)$, so the total time spent in scanning the
adjacency list is $O(E)$. Therefore, the computational complexity of the bottom-up algorithm, where $NV$ is the number of non-visited vertices currently in the graph, is $O(NV + E)$. This suggests that the number of vertices in the frontier and the current number of non-visited vertices remaining would determine which approach is more efficient for a particular layer. For instance, in some cases, the bottom-up can be more efficient than the top-down approach when a parent is found before having completed a scan of the adjacency list. This can lead a large reduction in the number of comparisons required. However, in some other cases, for example in early layers the top-down is more efficient because most of the vertices explored in the adjacency list are being set as visited in these layers. To swap between the top-down or the bottom-up approaches, an online heuristic is introduced in Section 6.2.1. This heuristic is based on the comparison between the number of vertices in the layer versus the number of non-visited vertices in the graph.

6.2.1 Online Heuristic

One characteristic of real-world graphs is that they have small diameter [CZF04], as described earlier in Section 4.1. Traversing this type of graph, results in a rapid growth of the number of vertices to be visited during the first layers up to a threshold where the number starts to decrease [BAP12], suggesting that the number of vertices per layer changes. Based on this, and the fact that the efficiency of the two approaches to traverse the graph, top-down and bottom-up, rely on the number of input vertices and the number of edges, analysing the nature of these drastic changes is crucial. For this reason, the number of input vertices and the number of edges per layer are analysed to determine which approach is the best option to use to process each layer.

1. The number of input vertices. This analysis involves the calculation of the number of vertices in the layer and the number of non-visited vertices in the graph. Figure 6.2 shows the number of vertices per layer and the number non-visited vertices in an example typical real-world graph. It can be seen that the number of vertices per layer grows rapidly in the first four layers before dropping after the 6th layer. Additionally, the number of non-visited decreases significantly between layer 3 and 5 from roughly 1000 to 500 (note the logarithmic scale on the y-axis). Therefore, a good
strategy is to swap between top-down and bottom-up approaches depending on whether the number of vertices in the layer or the number of non-visited vertices in the graph is lower. Therefore, in this example, top-down should be applied to the layers that have fewer vertices to explore (layers 1, 2 and 7) and the bottom-up approach should be used for the remainder.

![Figure 6.2: Comparison between number of vertices to be explored in the layer and the number of non-visited vertices in the graph. Notice that to make comparable the two sets of numbers, the number non-visited vertices was divided by $\alpha = 1024$ value, the number of RMAT graph generated by Graph 500 using $SCALE = 20$ and edgefactor = 16.](image)

2. **The number of edges.** Determining which is the best approach for processing each layer is also related to the number of edges explored per layer. When using the top-down approach the number of edges to be checked grows rapidly compared to the number of vertices actually visited. Table 6.1 shows an example which illustrates the large difference between the growth rate of the number of edges and of the number of vertices actually traversed per layer. For example, in layer number four, the top-down approach would check 17,626,910 edges but would only visit 100,874 vertices. This large difference can be exploited in order to perform an efficient exploration. In layers where the difference between the number of edges and the number
of traversed vertices is particularly large, the bottom-up approach can be applied. In the first layers, the top-down approach is more efficient because the majority of the vertices have not yet been visited. However, in the middle layers, where the number of edges grows rapidly compared to the number of explored vertices, the bottom-up approach is likely to be more efficient. Because there is more chance to find a parent for a non-visited vertex within the already visited vertices in the previous layers, reducing the steps for searching a parent.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Vertices</th>
<th>Edges</th>
<th>Traversed vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>21,892</td>
<td>18,122</td>
</tr>
<tr>
<td>3</td>
<td>18,122</td>
<td>13,547,462</td>
<td>540,575</td>
</tr>
<tr>
<td>4</td>
<td>540,575</td>
<td>17,626,910</td>
<td>100,874</td>
</tr>
<tr>
<td>5</td>
<td>100,874</td>
<td>150,698</td>
<td>486</td>
</tr>
<tr>
<td>6</td>
<td>486</td>
<td>490</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Traversed vertices per layer for a 1,048,575 million vertices graph, RMAT graph of SCALE 20 and edgefactor 16.

Based on a similar analysis, Beamer et al. [BAP12] developed a new hybrid BFS algorithm. This hybrid BFS uses an online heuristic that swaps between the top-down and the bottom-up. This heuristic is based on different metrics to switch to the top-down approach for “small” layers and to the bottom-up for “large” layers. Figure 6.3 shows a sketch of the hybrid BFS algorithm, where it can be seen that the while the layers in the graph are growing the top-down is applied to switch to the bottom-up in the middle layers and finally swap back to the top-down when the layers are shrinking. The metrics involved in the online heuristic are described in the following section.

6.2.2 Metrics

Whether a layer is processed using the top-down or bottom-up approach is dictated by an online heuristic based on three metrics: the number of edges to check in the frontier \( m_f \), the number of vertices in the frontier \( n_f \) and the number of edges to check of non-visited vertices \( m_u \). These metrics are implemented using counters which are updated during the traversal of each layer. The \( m_f \) adds up the degree
of every node in the layer, $n_f$ sums the number of vertices added to the layer and $m_u$ counts the number of edges of the non-visited vertices. The heuristic compares these metrics at the end of each layer and the resulting information is used to determine whether the layer should be processed using the top-down or the bottom-up approach.

The switch from top-down to bottom-up occurs when the number of edges in the layer is greater than the number of total non-visited edges in the graph. However, since the number of vertices in the frontier and the number of edges are considerably different in magnitude, to make $m_f$ and $m_u$ comparable, the number of non-visited edges $m_u$ is divided by a tuning parameter $\alpha$. This ratio is also called $C_{TB}$, as shown in Equation 6.1. The switch from bottom-up to top-down occurs when the layer starts to shrink. This is detected when the number of vertices in the layer is less than a proportion of the total number of vertices in the graph ($n$), controlled by the tuning parameter $\beta$, where $n/\beta = C_{BT}$, shown in Equation 6.2.

\[
m_f > \frac{m_u}{\alpha}, \text{ where } \frac{m_u}{\alpha} = C_{TB} \tag{6.1}
\]
\[
n_f < \frac{n}{\beta}, \text{ where } \frac{n}{\beta} = C_{BT} \tag{6.2}
\]
6.2.3 Switching Points

There are four transitions that can occur when switching between the top-down approach and the bottom-up approach. Figure 6.4 illustrates these state transitions.

1. **Top-Down.** The first transition starts with the top-down approach and stays there while the number of edges in the layer is less than or equal to $C_{TB}$.

2. **Top-Down to Bottom-Up.** The second transition is from the top-down to the bottom-up approach and it happens when the number of edges in the frontier is greater than the number of unexplored edges. This transition detects the threshold when the layer is large enough to start processing it with the bottom-up.

3. **Bottom-Up.** The third transition stays in the bottom-up approach processing the middle layers, while the number of edges in the frontier is greater or equal to $C_{BT}$.

4. **Bottom-Up to Top-Down.** The fourth transition is when switching from the bottom-up to the top-down, and it happens when the number of vertices in the layer is less than a fraction of the total number of vertices in the graph, which is when the layers start shrinking.

Figure 6.4: State diagram to switch between top-down and bottom-up approaches in the hybrid BFS algorithm.
An example of the transitions between the top-down and bottom-up approaches of the hybrid BFS algorithm is presented in Table 6.2. The first transition occurs when the number of edges in the layer $m_f$ is greater than the threshold $C_{TB}$. The hybrid BFS switches back to the top-down approach when the number of vertices in the layer $n_f$ is less than $C_{BT}$.

<table>
<thead>
<tr>
<th>Layers</th>
<th>$n_f$</th>
<th>$m_f$</th>
<th>$m_u$</th>
<th>$C_{TB}$</th>
<th>$C_{BT}$</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>554</td>
<td>4,194,304</td>
<td>299,593</td>
<td>10,922</td>
<td>top-down</td>
</tr>
<tr>
<td>2</td>
<td>554</td>
<td><strong>668,380</strong></td>
<td>6,934,258</td>
<td><strong>495,304</strong></td>
<td>10,922</td>
<td>bottom-up</td>
</tr>
<tr>
<td>3</td>
<td>97,725</td>
<td>6,672,448</td>
<td>261,770</td>
<td>18,697</td>
<td>10,922</td>
<td>bottom-up</td>
</tr>
<tr>
<td>4</td>
<td>77,711</td>
<td>260,803</td>
<td>967</td>
<td>69</td>
<td>10,922</td>
<td>bottom-up</td>
</tr>
<tr>
<td>5</td>
<td>868</td>
<td>882</td>
<td>85</td>
<td>6</td>
<td>10,922</td>
<td>top-down</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>80</td>
<td>5</td>
<td>10,922</td>
<td>top-down</td>
</tr>
</tbody>
</table>

Table 6.2: Example of the transition between the top-down and bottom-up approaches of the hybrid BFS for a graph size $SCALE = 18$ and $edgefactor = 16$ using $\alpha = 14$ and $\beta = 24$.

The online heuristic was adapted by Gao et al. [GLZS14] for their implementation BFS hybrid on the Xeon Phi. However, they reduced the overhead of the previously described approach by eliminating the calculation of $m_f$ (number of edges in the frontier) and $m_u$ (number of non-visited edges) metrics. Instead, Gao et al. [GLZS14] only calculate the number of vertices, $n_f$, and the number of non-visited, $n_u$, in the frontier. Therefore, this algorithm uses Equations 6.3 and 6.4.

$$n_f > \frac{n_u}{\alpha}, \text{ where } \frac{n_u}{\alpha} = C_{TB} \tag{6.3}$$

$$n_f < \frac{n}{\beta}, \text{ where } \frac{n}{\beta} = C_{BT} \tag{6.4}$$

Table 6.3 shows, for the same example as used in Table 6.2, the online heuristic implemented by Gao et. al [GLZS14]. The first switch occurs in layer 2 when the number of vertices in the layer ($n_f$) is greater than $C_{TB}$. The second switch occurs when the number of vertices in the layer decreases to the point that is less than $C_{BT}$. The choice for each layer is seen to be the same.

Table 6.4 presents a summary of the values of the two parameters used in the online heuristic, $\alpha$ and $\beta$, used in the two implementations of the BFS hybrid algorithm done by Beamer et al. [BAP12] and Gao et al. [GLZS14].
Layers | $n_f$ | $n_u$ | $C_{TB}$ | $C_{BT}$ | Approach
--- | --- | --- | --- | --- | ---
1 | 1 | 262,143 | 255 | 4,096 | top-down
2 | 554 | 261,589 | 255 | 4,096 | bottom-up
3 | 97,725 | 163,864 | 160 | 4,096 | bottom-up
4 | 77,711 | 86,153 | 84 | 4,096 | bottom-up
5 | 868 | 85,285 | 83 | 4,096 | top-down
6 | 5 | 85,280 | 83 | 4,096 | top-down

| TEPS | 1,009,411,193
---

Table 6.3: Example of the transition between the top-down and bottom-up approaches of the hybrid BFS for a graph size SCALE=18 and edgefactor = 16.

<table>
<thead>
<tr>
<th>Reference</th>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>SCALE</th>
<th>edgefactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beamer et al. [BAP12]</td>
<td>14</td>
<td>24</td>
<td>$&gt;$20</td>
<td>$&gt;$8</td>
</tr>
<tr>
<td>Gao et al. [GLZS14]</td>
<td>1,024</td>
<td>64</td>
<td>18-22</td>
<td>8, 16, 32 and 64</td>
</tr>
</tbody>
</table>

Table 6.4: Parameters used in the online heuristic: $\alpha$ and $\beta$. Developed by Beamer et al. [BAP12] and Gao et al. [GLZS14].

To summarise, the difference between the implementations is that Beamer et al. [BAP12] use metrics based on the number of edges in the layer $m_f$ and the number of edges of non-visited vertices $m_u$ to control the switching between the top-down and the bottom-up. However, Gao et al. [GLZS14] uses metrics based on the number of vertices, $n_f$ and the total number of non-visited vertices in the graph $nv$, to control switching. Despite the fact that the metrics based on the number of edges are more accurate, as they calculate the actual number of vertices that are processed during the adjacency list exploration, they require additional counters to sum the degrees of the vertices involved, adding extra computational overhead. As an example of the difference in performance for the executions presented in Table 6.2 and Table 6.3 can be seen when comparing the number of TEPS between both heuristics: around 800 MTEPS for the former and 1 GTEP for the latter. For this reason, in this work, the online heuristic used is based only on the number of vertices rather than the number of edges. Specifically, the hybrid BFS implementation presented in this chapter uses an online heuristic which swaps between the top-down and bottom-up approach based on the $n_f$ (number of vertices in the frontier) and $n_u$ (number of non-visited vertices) metrics; and $n_f$ and $n$ (total number of vertices in the graph) are used to swap between the bottom-up to the top-down. The values used for the parameters $\alpha$ and $\beta$ are the same as the ones used by Gao et al. [GLZS14] presented in Table 6.4 because these
values showed to make the switch between the two approaches of the algorithm in the key layers.

6.2.4 Description of the Algorithm

The pseudocode of the hybrid BFS is presented in Algorithm 9, showing the switching points to transition between the top-down and the bottom-up algorithms. Particularly, the vectorisation of the hybrid involves the vectorised version of both algorithms. The vectorisation of the top-down algorithm is described and analysed in Chapter 5 and the vectorisation of the bottom-up is described in Section 6.3.

The experiments described in Section 6.4 uses the hybrid BFS version calling the vectorised version of the top-down.

Algorithm 9 hybrid-BFS($G, s$)

1: while $in \neq 0$ do
2:   if $|in| < \frac{n}{2}$ then
3:     top-down($in, vis, out, P)$
4:   else if $|in| > \frac{2n}{\alpha}$ then
5:     bottom-up($in, vis, out, P)$
6:   end if
7:   $n_u \leftarrow$getCounters()
8:   swap($in, out$)
9:   $out \leftarrow 0$
10: end while

6.3 Bottom-Up Vectorisation

The bottom-up approach, introduced in Section 6.1, consists of stepping through each of the non-visited vertices in a layer to find the first vertex in their adjacency list in the frontier. In general, the vectorisation of an algorithm can be tricky in the sense that input data need to be structured in a way that increases vector unit utilisation as discussed in Section 3.4. Specifically, the vectorisation of the bottom-up algorithm does not follow the same principle of vectorising the adjacency list as is used in the top-down, which consists of processing the adjacency list in chunks of 16 elements as described in Chapter 5. This is mainly because while the top-down approach intends to process the largest number of vertices at the same time, the bottom-up aims only to find one parent of each vertex to be processed. For
that reason, the vectorisation of the bottom-up involves an extra step to gather the input vertices in a layout that better utilises the vector unit. Section 6.3.1 explains the process of vectorising the bottom-up, followed by an analysis of which layers the bottom-up is better than top-down in Section 6.3.2.

6.3.1 Algorithm: Setting Multiple Parents

While the parallel version of the bottom-up in Section 6.1 looks to set one parent of each non-visited vertex at a time, the vectorised algorithm looks to set parents for multiple vertices at a time. The algorithm consists of the following four steps to process 16 input vertices at a time aiming to find a parent for each by iterating through their adjacency lists.

1. **Load input vertices.** A sequence of 16 consecutive vertices is loaded to a vector register in a loop over all vertices.

2. **Filtering non-visited vertices.** The input vertices are filtered, so that the only vertices being processed are the non-visited vertices. This filtering can be done by using a bit vector mask formed from the visited bitmap array.

3. **Adjacent vertices iterations.** Each of the adjacent vertices of the non-visited vertices are gathered over a number of iterations. In the first iteration, the first vertex in the adjacency list for each non-visited vertex is gathered. Subsequent iterations gather the next adjacent vertex. The number of iterations is determined by a statically set threshold related with the average number of iterations required before a parent is found. Figure 6.5 illustrates the gathering of the adjacent vertices of the \( n^{th} \) iteration. After the adjacent vertices are gathered, they are tested to verify if they have been visited previously. The adjacent vertices that are in the frontier of the layer will be set as the parents of their respective vertex and they are no longer considered for processing in the next iteration. After the threshold is reached, the vectorised version is no longer efficient and the algorithm swaps to the non-SIMD version.

4. **Execute the non-SIMD version.** After the threshold in the adjacent vertices is achieved, the input vertices that have not yet found a parent are
processed by the using the non-SIMD bottom-up algorithm introduced in Section 6.1.

Algorithm 10 shows the pseudocode of the vectorisation of the bottom-up BFS algorithm. The process starts by loading a sequence of 16 consecutive vertices to be processed, starting with vertex zero up to the total number of vertices in the graph. Thus, the vertices are filtered so only the non-visited vertices are going to be processed. Then, to do so the visited bitmap array is iterated through stepping every word (32-bits integer) as it is shown in line 1. Since each word is 32 bits in length, but only 16 (32-bit) elements can be loaded into the vector register, the word is processed in two halves as shown in lines 3 to 5. The function LoadVertices transforms a single half word into a vector of 16 vertices, with the word parameter specifying whether the upper or lower half word should be processed. Function GetHalf loads the 16 bits of a half word into the bit mask mask_{vis}. This bitmask is used further to filter the visited vertices out.

The process continues by iterating through the adjacent vertices of each vertex in the input vertices. The maximum number of iterations is delimited by a
Algorithm 10 bottom-up-multiple-set\((in, \ vis, \ out, \ P)\)

1: \textbf{for} \(u \in \ vis\) \textbf{do} \hspace{1cm} \triangleright \text{Stepping through every word (32-bits) in \(\vis\).}
2: \hspace{1cm} \text{\(word = 0\)}
3: \hspace{1cm} \textbf{while} \(word < 2\) \textbf{do}
4: \hspace{2cm} \text{vertices} \leftarrow \text{LoadVertices}(u, \ word) \hspace{1cm} \triangleright \text{Loading 16 vertices.}
5: \hspace{2cm} \text{\(mask_{\vis} \leftarrow \text{GetHalf}(u, \ word)\)}
6: \hspace{2cm} \text{\(word = word + 1\)}
7: \hspace{2cm} \text{\(pos = 0\)}
8: \hspace{2cm} \textbf{for} \ pos < MAX\_POS \textbf{do}
9: \hspace{3cm} \text{LookingParents}(in, \ vis, \ out, \ P, \ vertices, \ pos, \ word, \ mask_{\vis}, \ mask) \hspace{1cm} \triangleright \text{Every bit in flag.}
10: \hspace{3cm} \text{\(pos = pos + 1\)}
11: \hspace{2cm} \textbf{end for}
12: \hspace{1cm} \text{\(i = 0\)}
13: \hspace{1cm} \textbf{while} \ mask.getBit(i) == 0 \text{ AND } i < 16 \textbf{do} \hspace{1cm} \triangleright \text{Every bit in flag.}
14: \hspace{2cm} \text{bottom-up-noSIMD()} \hspace{1cm} \triangleright \text{Every bit in flag.}
15: \hspace{2cm} \text{\(i++\)}
16: \hspace{2cm} \textbf{end while}
17: \hspace{1cm} \textbf{end while}
18: \textbf{end for}

threshold called \(MAX\_POS\). This threshold is a constant calculated based on the minimum number of iterations within which a parent is most likely to be found. In each iteration the function \(\text{LookingParents}()\) is called to process the 16 input vertices simultaneously aiming to find their parents. If no parent vertex is found up to the threshold, the search continues using the non-SIMD bottom-up algorithm presented in Section 6.1.

Setting the threshold \(MAX\_POS\)

The threshold that determines the number of iterations to be processed is related to the average number of iterations through the adjacency list before a parent is found. Table 6.5 shows the average number of edges processed per visited vertex per layer during the 64 iterations of the bottom-up BFS algorithm. Notice that the average was calculated by taking into account only the starting vertices that lead to connected subgraphs, unconnected starting vertices are ignored. Also, during the 64 iterations, the number of layers varied between 5 and 6 for different starting vertices, hence the average for the sixth layer was calculated based on only the non-zero values. The results in Table 6.5 show that the middle layers (layers 3, 4 and 5) have smaller averages of 34.0, 1.06 and 2.51, respectively, implying that the bottom-up BFS algorithm is more effective for the middle layers. However,
CHAPTER 6. THE HYBRID BFS VECTORISATION

the average for the third layer is above the average for the fourth and the fifth layers. For this reason, Figure 6.6 shows the variability of the number of edges per visited vertex for the third layer in a boxplot format, which is described in Section 4.1.2. Figure 6.6 (a) shows 6 outliers (4 circles and 2 crosses) that clearly make a significant impact on the resulting average and Figure 6.6 (b) shows the zoom in of the same plot showing only the first two outliers. It can be seen that the number of edges per visited vertex in the vast majority of cases is under 8. For this reason, the value of the MAX_POS threshold used for the further experiments of the bottom-up BFS algorithm is statically set to 8.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Avg Edges/Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,879,193</td>
</tr>
<tr>
<td>2</td>
<td>30,910</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>1.06</td>
</tr>
<tr>
<td>5</td>
<td>2.51</td>
</tr>
<tr>
<td>6</td>
<td>41.06</td>
</tr>
</tbody>
</table>

Table 6.5: Average of the number of edges processed per visited vertex using the the bottom-up BFS algorithm per layer. The graph size is SCALE = 18 and the edge factor = 16.

Algorithm 11 shows the pseudocode of the LookingParents function used for processing the adjacent vertices.

1. Firstly the 16 adjacent vertices of the “input vertices” are loaded into the vector register using the LoadAdj function (line 1). This function gathers the \( n^{th} \) adjacent vertices of the input vertices, only the ones set to zero in the bitmask \( mask_{vis} \) are candidates to be processed. The starts and ends input lists are the starting and the ending indexes in the adjacency list of each vertex and \( pos \) is the \( n^{th} \) location within this range. If \( pos \) exceeds this range, a mask \( (mask_{pos}) \) is set so as to not take into account that vertex for further processing in the next iteration. The return value is a vector that holds a list of 16 adjacent vertices.

2. Secondly, the frontier of the adjacent vertices is gathered by the function Gather (line 2). This function gathers the respective values in the frontier of the current adjacent vertices in the vector \( vadj \), only for the vertices set to zero in the input mask \( mask_{pos} \). The result of this function is a 16-bits
mask, in which each bit indicates whether an adjacent vertex in \( vadj \) is in the frontier or not.

Figure 6.6: Boxplot of the third layer of the 64 bottom-up BFS executions for a graph size of \( SCALE = 18 \) and \( edge \text{factor} = 16 \).
3. Thirdly, the bitmask is tested to verify whether at least one of the adjacent vertices has been found to be in the frontier of the current layer (line 3).

4. Finally, the parents found are scattered back to the predecessor array ($P$) and the visited and output queue bitmap arrays. Additionally, a mask, which is received as input parameter of the function, is updated each time new parents have been found. This mask is used in further iterations to identify the vertices that have already had a parent found to prevent them from being processed in further iterations.

**Algorithm 11** LookingParents($in, vis, out, P, vertices, pos, mask_{vis}, mask_{pos}$)

1: vadj ← LoadAdj(vertices, starts, ends, pos, mask_{pos}, mask_{vis})
2: frontier ← in.Gather(vadj, mask_{pos})
3: if frontier.Test() != 0000 then $\triangleright$ At least there is one parent in the frontier.
4: P.Scatter(vertices, vadj, frontier)
5: vis ← vis $\cup$ frontier
6: out ← out $\cup$ frontier
7: mask ← mask $\cup$ frontier
8: end if

Figure 6.7 shows an example of the vectorisation of the bottom-up BFS algorithm illustrating the four steps of the algorithm. Furthermore, Listing 3 shows the source code of the LookingParents function. Additionally, Appendix C shows the source code of all the functions involved in the vectorisation of the bottom-up BFS algorithm.

### 6.3.2 Which layers are suitable for vectorisation?

The efficiency of the bottom-up using vectorisation relies on the number of non-visited vertices remaining. The more non-visited vertices there are, the better is the usage of the vector unit. Table 6.6 shows an example of the variation of number of non-visited vertices per layer in column $n_u$, for an example of a Graph 500 graph. The highlighted rows in gray are processed by the bottom-up algorithm and it can be seen that the number of non-visited vertices decreases rapidly during the processing of these layers. This raises the question about what is the best option to use the vectorisation version of the bottom-up algorithm along the middle layers. Two solutions, *vectorisation per layers* and *vectorisation per range*, were adapted to the algorithm to decide whether to use the vectorised version or the non-SIMD version.
__inline void SBFS_2QBM_hybrid_SIMD::LookingParents(int vis_mask,
bitmap_t *frontier, bitmap_t *queue, bitmap_t *explored, int pword,
int psegment, int &fend, int &pos, __m512i &vstart, __m512i &vend, __m512i &vvertices){
  __m512i vtmp = _mm512_set1_epi32(pos);
  __m512i vadd = _mm512_add_epi32(vstart, vtmp);
  __mmask16 vcmp = _mm512_cmpgt_epi32_mask(vend, vadd);
  __m512i vneig = _mm512_mask_i32gather_epi32(_mm512_set1_epi32(0), vcmp,
                                          vadd, rows, sizeof(word_t));
  int res = 0;
  //Getting the high part of the word (16-bits)
  vis_mask = (explored->start[pword]>>(psegment*VNELE8))&0xFFFF;
  __mmask16 mask1 = _mm512_kand(_mm512_knot(_mm512_int2mask(vis_mask)),vcmp);
  //2.- filter visited adjacent vertices according to vis and out
  //Getting WORD offset and BIT offset
  __m512i vword = _mm512_srlv_epi32(vneig, _mm512_set1_epi32(5));
  __m512i vbits = _mm512_and_epi32(vneig,_mm512_set1_epi32(0x1F));
  __m512i fron_words = _mm512_mask_i32gather_epi32(_mm512_set1_epi32(0), mask1,
                                           vword, frontier->start, sizeof(word_t));
  //Shifting 1 to the left indexes position in the vneig array
  __m512i bits= _mm512_sllv_epi32(_mm512_set1_epi32(1), vbits);
  //Filtering with the unvisited mask
  __mmask16 mask = _mm512_mask_test_epi32_mask(mask1 ,fron_words, bits);
  res = _mm512_mask2int(mask);
  if(mask != 0x0000 ){  //at least one neighbour is in the frontier
    _mm512_mask_prefetch_i32scatter_ps(bfs_tree, mask, vvertices,sizeof(word_t),
                                         _MM_HINT_T0);
    _mm512_mask_i32scatter_epi32(bfs_tree, mask, vvertices, vneig, sizeof(word_t));
    explored->start[pword] |= res<<(psegment*VNELE8);
    queue->start[pword] |= res<<(psegment*VNELE8);
  }
  //flag used to indicate the end of the processing
  fend |= (res | ((explored->start[pword]>>(psegment*VNELE8))&0xFFFF));
}

Listing 3: Source code of the LookingParents() SIMD function.

The vectorisation per layer is a coarse-grained solution and it means applying
the vectorised techniques described in Section 6.3.1 (ie. the processing of a word-
at-a-time) to every word in a layer. In this solution a layer-based threshold,
based on the fraction of the non-visited vertices in the layer determines whether the entire layers will be processed with vectorisation or not. If not, the layer will be processed using the non-SIMD bottom-up algorithm. The vectorisation per range solution is more fine-grained, thus if the layer is selected to be processed by the bottom-up BFS algorithm, each 32-bit word is tested to see whether it contains enough non-visited vertices to warrant use of the vectorised technique. This requires a new threshold to be set. These two solutions are discussed in detail in the following sections:

<table>
<thead>
<tr>
<th>Layers</th>
<th>( n_f )</th>
<th>( n_u )</th>
<th>Total vertices</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>262,143</td>
<td>262,144</td>
<td>top-down</td>
</tr>
<tr>
<td>2</td>
<td>554</td>
<td>261,589</td>
<td>262,144</td>
<td>bottom-up</td>
</tr>
<tr>
<td>3</td>
<td>97,725</td>
<td>163,864</td>
<td>262,144</td>
<td>bottom-up</td>
</tr>
<tr>
<td>4</td>
<td>77,711</td>
<td>86,153</td>
<td>262,144</td>
<td>bottom-up</td>
</tr>
<tr>
<td>5</td>
<td>868</td>
<td>85,285</td>
<td>262,144</td>
<td>top-down</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>85,280</td>
<td>262,144</td>
<td>top-down</td>
</tr>
</tbody>
</table>

Table 6.6: Example of the execution of the hybrid BFS for a graph size \( SCALE = 18 \), \( edge\text{-}factor = 16 \) and starting node = 2005, showing the number of vertices and the number of non-visited vertices per layer.
Vectorisation per Layer

This solution consists of setting a threshold for the non-visited vertices per layer aiming to only apply vectorisation to those layers containing a large amount of non-visited vertices. The layers that pass this threshold will be processed using the vectorised version of the bottom-up algorithm, the rest of the layers will use the non-SIMD version. The threshold is set as a fraction of the total number of vertices in the graph. The fraction used to set the threshold is high because the aim is to process the layers with a large number of non-visited vertices using vectorisation. For instance by setting the fraction to 0.9 of the total number of vertices, Table 6.7 shows the results where only layer number two passes the threshold and so executes the vectorised version, layers three and four execute the non-SIMD version of the algorithm.

<table>
<thead>
<tr>
<th>Layers</th>
<th>( n_f )</th>
<th>( n_u )</th>
<th>Fraction(0.9)</th>
<th>Total vertices</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>262,143</td>
<td>235,929</td>
<td>262,144</td>
<td>top-down</td>
</tr>
<tr>
<td>2</td>
<td>554</td>
<td>261,589</td>
<td>235,929</td>
<td>262,144</td>
<td>SIMD</td>
</tr>
<tr>
<td>3</td>
<td>97,725</td>
<td>163,864</td>
<td>235,929</td>
<td>262,144</td>
<td>non-SIMD</td>
</tr>
<tr>
<td>4</td>
<td>77,711</td>
<td>86,153</td>
<td>235,929</td>
<td>262,144</td>
<td>non-SIMD</td>
</tr>
<tr>
<td>5</td>
<td>868</td>
<td>85,285</td>
<td>235,929</td>
<td>262,144</td>
<td>top-down</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>85,280</td>
<td>235,929</td>
<td>262,144</td>
<td>top-down</td>
</tr>
</tbody>
</table>

Table 6.7: Example of the execution of the hybrid BFS for a graph size SCALE=18, edgefactor = 16 and starting node = 2005, showing the fraction threshold set to 0.9 to choose between either the vectorised or the non-SIMD version of the bottom-up algorithm.

Vectorisation per Range

Similarly to the vectorisation per layers solution, the vectorisation per range sets a threshold to either call the vectorised version of the bottom-up algorithm or the non-SIMD one. Nevertheless, the main difference is that vectorising per layers only uses the vectorised version for complete layers, whereas vectorising per range is applied to all the middle layers to be processed by the bottom-up.

The threshold is tested during the filtering of the non-visited vertices of the sequence of 16 input vertices in the previously described vectorised bottom-up algorithm in Section 6.3.1. Specifically, Algorithm 12 shows the pseudocode after the threshold has been added. This checks if the number of non-visited vertices...
exceeds the threshold, in which case the vectorised version of the bottom-up algorithm is executed, otherwise it swaps to the non-SIMD version for the (half) word.

Algorithm 12 LookingParents\(\left(\text{in, vis, out, } P, \text{vertices, pos, mask}_{\text{vis}}, \text{mask}_{\text{pos}}\right)\)

\begin{algorithm}
\begin{algorithmic}
\STATE \text{vadj} ← \text{LoadAdj(vertices, starts, ends, pos, mask}_{\text{pos}}, \text{mask}_{\text{vis}})
\IF {\text{CountZeroBits(mask}_{\text{vis}}) > THRESHOLD}}
\STATE \text{parents} ← \text{in.Gather(vadj, mask}_{\text{pos}})
\IF {\text{parents.Test}() \neq 0x0000}
\STATE $\triangle$ At least there is one parent in the layer.
\STATE \text{P.Scatter(vertices, vadj, parents)}
\STATE \text{vis} ← \text{vis} \cup \text{parents}
\STATE \text{out} ← \text{out} \cup \text{parents}
\STATE \text{mask} ← \text{mask} \cup \text{parents}
\ELSE
\STATE $i = 0$
\WHILE {\text{mask.getBit}(i) == 0 \AND i < 16}
\STATE $\triangle$ Every bit in flag.
\STATE \text{bottom-up-noSIMD()}
\STATE $i++$
\ENDWHILE
\ENDIF
\end{algorithmic}
\end{algorithm}

The threshold can vary in the range of the 16 elements to vectorise, so between 1 and 16. However, because it is expected that vectorisation will be better for a high number of non-visited vertices, the threshold needs to be tuned. Table 6.8 shows the results in performance (GTEPS) for a typical graph and start vertex when varying the threshold. The threshold with value 8 is the one with the peak performance and it was selected for the analysis in Section 6.5.

<table>
<thead>
<tr>
<th>Vectorisation Threshold</th>
<th>GTEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1.75</td>
</tr>
<tr>
<td>8</td>
<td>1.76</td>
</tr>
<tr>
<td>6</td>
<td>1.73</td>
</tr>
<tr>
<td>4</td>
<td>1.75</td>
</tr>
</tbody>
</table>

Table 6.8: Example of the execution of the hybrid BFS for a graph size SCALE = 18, edge factor = 16, varying the vectorisation threshold for the non-visited vertices. The number of GTEPS are the average out of 5 iterations.
Since the number of non-visited vertices are spread around the layer, there are some cases where most of the sequence of 16 vertices are non-visited. In those cases, the vectorisation per range decides which of the two algorithms between the vectorised and the non-SIMD version is the better option in terms of performance in a more dynamic fine-grained way. Table 6.9 shows the number of executions (i.e. 16 bit half word) for which the vectorised version was chosen for the middle layers. It can be seen that layer two applied vectorisation all along the layer because the counter (16,384) is equal to the total number of vertices ($2^{18} = 262,144$) divided by 16. Furthermore, in this example, layer 3 applied quite a significant amount of vectorisation in the layer, whereas layer 4 did not use vectorisation at all.

<table>
<thead>
<tr>
<th>Layers</th>
<th>Vectorised executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16,384</td>
</tr>
<tr>
<td>3</td>
<td>12,780</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.9: Number of vectorised executions with the threshold set to 8. This example was executed with the bottom-up BFS using SIMD-range with threshold for the non-visited vertices set to 8. Graph size SCALE=18, edgefactor=16 for starting vertex 2005.

6.4 Experimental Setup

This section presents the experimental settings used in the experiments to evaluate and analyse the performance of the vectorisation of the hybrid BFS. The experiments are divided in two parts. Firstly, three different implementations of the bottom-up are evaluated to compare their performance. Secondly, a complete comparison is conducted comparing the hybrid BFS algorithm using the non-SIMD bottom-up and the versions using vectorisation.

**Hardware platform** Section 3.4.3 presents a detailed summary of the hardware and compilation settings for these experiments.
6.4.1 Bottom-Up Settings

The hybrid BFS algorithm is evaluated using three different implementations of the bottom-up algorithm:

1. **non-SIMD** This implementation refers to the bottom-up algorithm presented in Section 6.1, which does not use vectorisation.

2. **SIMD-layer** This bottom-up implementation applies vectorisation per layers and is introduced in Section 6.3.2.

3. **SIMD-range** The bottom-up using the vectorisation per range as presented in Section 6.3.2.

The evaluation consists of comparing the performance in terms of number of TEPS achieved using the Experimental Developmental Framework described in Chapter 4. This framework allows an easy access to the Graph 500 experimental functions for the three different implementations. The input parameters for the experiments are shown in Table 6.10. Specifically, these parameters were chosen because there is an interesting performance gap between the SIMD and the non-SIMD versions shown later in Section 6.11.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>18</td>
</tr>
<tr>
<td>Edgefactor</td>
<td>32</td>
</tr>
<tr>
<td>Number of threads</td>
<td>228</td>
</tr>
<tr>
<td>Affinity</td>
<td>balanced</td>
</tr>
</tbody>
</table>

Table 6.10: Experimental input parameters.

In addition, the Performance Application Programming Interface library (PAPI) [oT16] was used to get access to the hardware performance counters available on the Xeon Phi providing access to the processor events; more details about the performance counters on Xeon Phi can be seen earlier in Section 3.5. The main events relevant to the analysis of the three implementations are related with instruction counting and cache access, see Table 6.11.

Based on the results of the three bottom-up implementations a thorough performance analysis between them is presented in Section 6.5.
6.4.2 Hybrid Breadth-First Search Settings

These experiments aim to evaluate the hybrid BFS algorithm using the non-SIMD bottom-up against the SIMD-layer version, which according to the further analysis in Section 6.5 is better than the SIMD-range version. The evaluation involves the variation of the graph sizes used while executing on the maximum number of threads on the Xeon Phi architecture. The variation of the graph sizes was chosen based on the results presented by the state-of-the-art implementation of Gao et al. [GLZS14]. Therefore, the graph size varies the SCALE from 18 to 22 and the edgefactor for 16, 32 and 64. The maximum number of threads for the Intel Xeon Phi architecture is 240, but to be able to compare with Gao et al. in Section 6.6, the experiments were set up for 228 threads. In summary, the input parameters used for this experiment are summarised in Table 6.12.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>18-22</td>
</tr>
<tr>
<td>Edgefactor</td>
<td>16, 32, 64</td>
</tr>
<tr>
<td>Number of threads</td>
<td>228</td>
</tr>
<tr>
<td>Affinity</td>
<td>balanced</td>
</tr>
</tbody>
</table>

Table 6.12: Experimental input parameters.

6.5 Bottom-Up Analysis

The performance analysis of the vectorisation of the bottom-up algorithm is one of the main contributions of this thesis and it consists of a systematic comparison of three versions of the bottom-up, the non-SIMD, the SIMD-layer and the SIMD-range. This comparison is based not only in terms of performance (TEPS) but also in terms of the effectiveness of the utilisation of resources of the Xeon Phi
architecture, including cache access and counting instructions. Firstly, to be able to compare the PAPI events for the function calls related to the bottom-up algorithm, a specific starting vertex was selected by using the criterion described in Section 6.5.1. Secondly, the PAPI events for the instructions counters and the cache access are gathered for the three bottom-up implementations and analysed in Section 6.5.2. Finally, a performance analysis in terms of TEPS is presented in Section 6.5.3.

In this section, there are three terms used to identify the version of the BFS algorithm chosen per layer. Table 6.13 shows these terms.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD-SIMD</td>
<td>SIMD top-down</td>
</tr>
<tr>
<td>BU-SIMD</td>
<td>SIMD bottom-up</td>
</tr>
<tr>
<td>BU-noSIMD</td>
<td>no-SIMD bottom-up</td>
</tr>
</tbody>
</table>

Table 6.13: Terminology for different versions of the BFS algorithm.

### 6.5.1 Choosing a Starting Vertex

The boxplot graph presented in Figure 6.8 depicts the performance variability of the 64 BFS executions of the Graph 500 experimental suite described in Chapter 4. The definition of a boxplot graph is described in Section 4.1.2. Each one of the 64 executions computes the BFS algorithm with a randomly chosen starting vertex. Despite the 64 starting vertices being selected by a random process, they remain the same during different executions of the same experiment because they are generated by the same seed [MWBA10]. Figure 6.8 shows TEPS data for 10 separate executions of the same experiment. In the figure can be observed, that the variation, in terms of TEPS, is higher for some starting vertices than others for the 10 independent experiments. The box represents the range in which 50% of the TEPS values lie and the horizontal line represents the median value and the crosses (+) and circles (○) represent the outliers.

As an example, the variations for the fourth and the twelfth BFS executions with starting vertices 12 and 119 respectively are shown in the boxplot representation in Figure 6.9. At first glance, it can be seen that the variation for the twelfth execution is higher than that for the fourth execution. While the difference between the minimum (the first “whisker”) and the maximum (the last “whisker”) value for execution 12 is around 300 MTEPS, the same range
Figure 6.8: Boxplot graph representing the variation in number of TEPS of the 64 BFS executions among 10 experimental iterations for each of the 64 random starting vertices. The implementation uses the non-SIMD version of the bottom-up algorithm on 228 threads. The graph size is $SCALE = 18$ and $edge\ factor = 32$.

for execution 4 is around 100 MTEPS. Moreover, the IQR (interquartile range)$^1$ for execution twelfth is around 200 MTEPS, which is higher than the IQR of execution 4 (around 50 MTEPS).

This variability between iterations is due to the the parallel execution of BFS being non deterministic [LS10], which implies that different executions with the same starting vertex can lead to different valid output BFS trees. Having different options to traverse the graph causes a variation in performance from run-to-run because threads process of vertices non-deterministically leading to vertices being explored in different orders from run-to-run which results in different BFS trees being generated at a different processing time since, for example, unexplored vertices encountered in one ran may have already been explored in another run.

$^1$Also called *middle fifty* and it refers to the difference between the third quartile and the first quartile around the mean [MTL78].
This is in addition to the normally expected variation of execution time from run-to-run due to other processes executing on a computer, which found to be small on the Xeon Phi.

Despite this variation being interesting to analyse, it involves the setting up of a robust infrastructure to be able to capture the different output BFS trees in a deterministic way which would require more development time. Instead, for practical reasons, the following experiments conducted for the bottom-up analysis in Section 6.5.2 use the fourth execution as a basis since it presents low variation from run-to-run minimising the impact of the variation on the results used for the performance analysis.

Another cause of this variation is due to the great variation in performance according to the starting vertex. To show an example of the variation along the 64 iterations, two executions are analysed, the first one uses a “good” starting vertex achieving high number of TEPS and the second one is a “bad” starting vertex, which has lower TEPS compared with maximum number of TEPS, the results of both executions are presented in Tables 6.14 and 6.15. The results show the approach selected per layer, the metrics ($n_f$, $n_u$, $C_{TD}$ and $C_{BU}$) based on the online heuristic presented previously as the time (seconds) and the final number of TEPS of the execution.

First of all, note the difference in performance of a factor of approximately 1.5 between the number of TEPS of the “bad” and the “good” starting vertices. Also, despite both executions are processed in six layers, the approaches to solve each layer are different. For instance, in Table 6.14, layers 2, 3 and 4 are processed by using the bottom-up, whereas the execution in Table 6.15 uses the bottom-up in different layers (3, 4 and 5). This is due to the online heuristic introduced in Section 6.2.1 that selects the approach to use either the top-down or bottom-up BFS algorithms driving the choice of approach according to the metrics calculated by $C_{TB}$ and $C_{BT}$. The main difference is in the number of vertices in the frontier given by $n_f$. As can be seen in Table 6.14 for the “good” vertex, the frontier starts with one vertex and then expands quickly to 554, achieving the maximum number of vertices in the frontier of 97,725 to finally drop down. In contrast, for the “bad” vertex in Table 6.15, despite it also starting with one vertex in the frontier, it continues with 20 and 32,939, achieving the maximum number of vertices in the frontier of 137,716 in layer 4 and finally drops down. Therefore, despite the heuristic swaps properly according to the metrics in both cases, the difference
### Table 6.14: Results of the hybrid BFS with a “good” starting vertex \textbf{20025}, using $\alpha = 1024$ and $\beta = 64$ for a graph size of \textit{SCALE} = 18 and \textit{edgefactor} = 16.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Approach</th>
<th>$n_f$</th>
<th>$n_u$</th>
<th>$C_{TB}$</th>
<th>$C_{BT}$</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD-SIMD</td>
<td>1</td>
<td>262,143</td>
<td>255</td>
<td>4,369</td>
<td>0.0008</td>
</tr>
<tr>
<td>2</td>
<td>BU-noSIMD</td>
<td>554</td>
<td>261,589</td>
<td>255</td>
<td>4,369</td>
<td>0.0010</td>
</tr>
<tr>
<td>3</td>
<td>BU-noSIMD</td>
<td>97,725</td>
<td>163,864</td>
<td>160</td>
<td>4,369</td>
<td>0.0007</td>
</tr>
<tr>
<td>4</td>
<td>BU-noSIMD</td>
<td>77,711</td>
<td>86,153</td>
<td>84</td>
<td>4,369</td>
<td>0.0002</td>
</tr>
<tr>
<td>5</td>
<td>TD-SIMD</td>
<td>868</td>
<td>85,285</td>
<td>83</td>
<td>4,369</td>
<td>0.0001</td>
</tr>
<tr>
<td>6</td>
<td>TD-SIMD</td>
<td>5</td>
<td>85,280</td>
<td>83</td>
<td>4,369</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

**TEPS** 934,797,415

### Table 6.15: Results of the hybrid BFS with a “bad” starting vertex \textbf{70108}, using $\alpha = 1024$ and $\beta = 64$ for a graph size of \textit{SCALE} = 18 and \textit{edgefactor} = 16.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Approach</th>
<th>$n_f$</th>
<th>$n_u$</th>
<th>$C_{TB}$</th>
<th>$C_{BT}$</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD-SIMD</td>
<td>1</td>
<td>262,143</td>
<td>255</td>
<td>4,369</td>
<td>0.0005</td>
</tr>
<tr>
<td>2</td>
<td>TD-SIMD</td>
<td>20</td>
<td>262,123</td>
<td>255</td>
<td>4,369</td>
<td>0.0035</td>
</tr>
<tr>
<td>3</td>
<td>BU-noSIMD</td>
<td>32,939</td>
<td>229,184</td>
<td>223</td>
<td>4,369</td>
<td>0.0009</td>
</tr>
<tr>
<td>4</td>
<td>BU-noSIMD</td>
<td>137,716</td>
<td>91,468</td>
<td>89</td>
<td>4,369</td>
<td>0.0002</td>
</tr>
<tr>
<td>5</td>
<td>BU-noSIMD</td>
<td>6,152</td>
<td>85,316</td>
<td>83</td>
<td>4,369</td>
<td>0.0002</td>
</tr>
<tr>
<td>6</td>
<td>TD-SIMD</td>
<td>36</td>
<td>85,280</td>
<td>83</td>
<td>4,369</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

**TEPS** 592,263,107

in performance implies that in some cases the efficiency of both approaches is determined by the input graph structure. For instance, for starting vertices with high degree the structure of the BFS tree traversal spans quickly in the first layers.

### 6.5.2 PAPI Counters Analysis

The three \textit{bottom-up} implementations: the non-SIMD, the SIMD-layer and the SIMD-range were instrumented to have direct access to some of the hardware events of the Xeon Phi processor using the PAPI library. By monitoring these events, it is possible to use the performance counters to help analyse each of the three implementations in terms of the usage of the resources on the Xeon Phi architecture. PAPI counters are turned on and off around the code segments that are to be analysed. Despite the Xeon Phi having several performance counters listed in Section 3.5, in this analysis five of them were selected to monitor the instructions of the program, cache memory accesses and the vector unit. The counters are divided in two sets, the instructions counters and those related to
Figure 6.9: Boxplot graph representing the variation in number of TEPS of the fourth execution out of the 64 BFS executions with the starting vertices 12, 119 over 10 experiment iterations. The BFS implementation uses the non-SIMD version of the bottom-up algorithm on 228 threads. The graph size is $SCALE = 18$ and $edge\ factor = 32$.

cache access. First, the instructions counters include the total number of cycles and the total number of instructions, and the CPI (cycles per instruction) is calculated based on them. The CPI can be seen as a metric to measure performance consisting of calculating the average number of clock cycles per instruction for a program [HP11]. Second, the cache access counters include the $L1$ data cache misses and the $L2$ load misses. In addition, an extra hardware counter relevant to analyse the vector unit utilisation is the number of vector or SIMD instructions. The analysis of the five counters is presented next. To have a clear analysis of the data, the experiments in the following sections were conducted by executing only one thread.
PAPI Instructions Counters

This section presents the results of the analysis based on the PAPI instruction counters for the hybrid BFS algorithm using the three different bottom-up implementations. The data for non-SIMD version, the SIMD-layer and the SIMD-range versions are presented in Table 6.16, Table 6.17 and Table 6.18, respectively.

The tables list the layers of the hybrid algorithm, followed by the approach used to traverse each layer. Specifically, the highlighted rows in grey are the layers processed by the bottom-up and those are the ones to focus on this analysis. Additionally, the results show the number of non-visited vertices in each layer which is relevant to the bottom-up approach. Finally, the performance counters for cycles and instructions are displayed as well as the CPI calculation.

Based on the results it can be seen whether in all cases the middle layers (3-5) use the non-SIMD or a vectorised/SIMD version of the bottom-up. Various observations can be made:

1. For either of the SIMD versions of the bottom-up (SIMD-layer or SIMD-range) the third layer, marked as BU-SIMD in Tables 6.17 and 6.18, is faster (time column) than the non-SIMD version for the third layer, and for the fourth and fifth layers all three versions (non-SIMD, SIMD-layer and SIMD-range) have approximately the same execution time.

2. Also it can be observed in the third layer in Tables 6.17 and 6.18 that the cycles, instructions and CPI, for both SIMD versions (SIMD-layer and SIMD-range) are approximately the same and both SIMD versions are better than the non-SIMD version. Based on this observation, further comparisons are made between the non-SIMD and only the SIMD-layer version.

3. The cycles count for the non-SIMD and the SIMD-layer are in the same proportion as the time, as expected, since the clock rate of the Xeon Phi is the same.

4. Comparing the number of instructions in the SIMD-layer version is lower than that for the non-SIMD but the CPI for SIMD is higher than that for the non-SIMD, suggesting that the instructions executed for SIMD are not as efficient as for non-SIMD version. This may be caused as a result of the SIMD-layer version using vector instructions.
Table 6.16: Hybrid BFS execution for starting vertex 12, 119 using the bottom-up non-SIMD version. PAPI was used to get the hardware counters for cycles, instructions and CPI(cycles/instructions). \textit{SCALE} = 18 and \textit{edge factor} = 32.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Approach</th>
<th>NV</th>
<th>Time (sec)</th>
<th>Cycles</th>
<th>Inst.</th>
<th>CPI (cyc/inst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD-SIMD</td>
<td>262,143</td>
<td>0.000092</td>
<td>105,945</td>
<td>15,772</td>
<td>6.72</td>
</tr>
<tr>
<td>2</td>
<td>TD-SIMD</td>
<td>262,142</td>
<td>0.000458</td>
<td>471,431</td>
<td>84,169</td>
<td>5.60</td>
</tr>
<tr>
<td>3</td>
<td>BU-noSIMD</td>
<td>259,153</td>
<td>0.000844</td>
<td>903,366</td>
<td>175,435</td>
<td>5.15</td>
</tr>
<tr>
<td>4</td>
<td>BU-noSIMD</td>
<td>96,758</td>
<td>0.000271</td>
<td>295,299</td>
<td>43,854</td>
<td>6.73</td>
</tr>
<tr>
<td>5</td>
<td>BU-noSIMD</td>
<td>62,373</td>
<td>0.000128</td>
<td>140,088</td>
<td>26,673</td>
<td>5.25</td>
</tr>
<tr>
<td>6</td>
<td>TD-SIMD</td>
<td>62,295</td>
<td>0.000086</td>
<td>96,070</td>
<td>14,091</td>
<td>6.82</td>
</tr>
</tbody>
</table>

Table 6.17: Hybrid BFS execution for starting vertex 12, 119 using the bottom-up vectorising per layer (SIMD-layer). PAPI was used to get the hardware counters for cycles, instructions and CPI(cycles/instructions). \textit{SCALE} = 18 and \textit{edge factor} = 32.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Approach</th>
<th>NV</th>
<th>Time (sec)</th>
<th>Cycles</th>
<th>Inst.</th>
<th>CPI (cyc/inst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD-SIMD</td>
<td>262,143</td>
<td>0.000085</td>
<td>96,417</td>
<td>14,269</td>
<td>6.76</td>
</tr>
<tr>
<td>2</td>
<td>TD-SIMD</td>
<td>262,142</td>
<td>0.00045</td>
<td>462,515</td>
<td>81,132</td>
<td>5.70</td>
</tr>
<tr>
<td>3</td>
<td>BU-SIMD</td>
<td>259,153</td>
<td>0.00063</td>
<td>687,282</td>
<td>99,539</td>
<td>6.90</td>
</tr>
<tr>
<td>4</td>
<td>BU-noSIMD</td>
<td>96,758</td>
<td>0.000272</td>
<td>302,230</td>
<td>40,385</td>
<td>7.48</td>
</tr>
<tr>
<td>5</td>
<td>BU-noSIMD</td>
<td>62,373</td>
<td>0.000123</td>
<td>135,862</td>
<td>25,139</td>
<td>5.40</td>
</tr>
<tr>
<td>6</td>
<td>TD-SIMD</td>
<td>62,295</td>
<td>0.000086</td>
<td>94,236</td>
<td>12,818</td>
<td>7.35</td>
</tr>
</tbody>
</table>

Table 6.18: Hybrid BFS execution for starting vertex 12, 119 using the bottom-up vectorising per range (SIMD-range). PAPI to get the hardware counters for cycles, instructions and CPI (cycles/instructions). The graph size is \textit{SCALE} = 18 and \textit{edge factor} = 32.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Approach</th>
<th>NV</th>
<th>Time (sec)</th>
<th>Cycles</th>
<th>Inst.</th>
<th>CPI (cyc/inst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD-SIMD</td>
<td>262,143</td>
<td>0.000088</td>
<td>102,299</td>
<td>14,651</td>
<td>6.98</td>
</tr>
<tr>
<td>2</td>
<td>TD-SIMD</td>
<td>262,142</td>
<td>0.000455</td>
<td>469,204</td>
<td>78,124</td>
<td>6.01</td>
</tr>
<tr>
<td>3</td>
<td>BU-SIMD</td>
<td>259,153</td>
<td>0.000632</td>
<td>688,487</td>
<td>100,131</td>
<td>6.87</td>
</tr>
<tr>
<td>4</td>
<td>BU-SIMD</td>
<td>96,758</td>
<td>0.000277</td>
<td>304,065</td>
<td>49,291</td>
<td>6.17</td>
</tr>
<tr>
<td>5</td>
<td>BU-SIMD</td>
<td>62,373</td>
<td>0.000135</td>
<td>148,529</td>
<td>30,800</td>
<td>4.82</td>
</tr>
<tr>
<td>6</td>
<td>TD-SIMD</td>
<td>62,295</td>
<td>0.000085</td>
<td>95,635</td>
<td>13,793</td>
<td>6.93</td>
</tr>
</tbody>
</table>

5. According to [Int12c] the threshold for the CPI in the Xeon Phi architecture should be investigated if it is higher than 4.0. However, it also points out that applications dominated by accessing to memory can exceed this threshold. The CPI result is definitely a metric that needs further exploration on the BFS algorithm, subject to future investigation.
CHAPTER 6. THE HYBRID BFS VECTORISATION

PAPI Cache Access and Vector Instructions Counters

This section presents the results of the PAPI cache access and vector related counters for the hybrid BFS algorithm for the three different bottom-up implementations. The non-SIMD version, the SIMD-layer and the SIMD-range version are summarised in Tables 6.19, 6.20 and 6.21, respectively.

Similarly to the instructions counters, the highlighted rows in grey are the layers processed by the bottom-up and those are the ones to focus on for the analysis. The tables show the number of non-visited vertices (NV) relevant to the bottom-up approach as well as the performance counters for cache access, including L1 cache misses and L2 data misses. These L1 and L2 data misses counters count the number of times when data was not found in the L1 and L2 caches of the Xeon Phi described in Chapter 3. The vector instructions counter is also included in this analysis. Notice that data shown in Tables 6.19, 6.20 and 6.21 is from a different run of the experiment because the maximum number of the PAPI performance counters per execution is restricted to 2 at a time for the Xeon Phi [Int12b]. Thus, the experiments were executed to get the L1 and L2 misses performance counters and then the vector instructions performance counter. This explains why the time data is slightly different from the previous results in Tables 6.16, 6.17 and 6.18, which also confirms the variability from run-to-run even using the same starting vertices as described earlier in Section 6.5.1.

There are three main observations that can be made:

1. The cache accesses counters (labeled L1 and L2 misses) are generally lower for the non-SIMD version than the vectorised version (BU-SIMD). For instance, for the the third layer of the non-SIMD version in Table 6.19, the L1 and L2 misses are 1,877 and 2,168 respectively, which are lower than the ones from the vectorised version either using the SIMD-layer (2,329 and 2,828) or SIMD-range (2,304 and 2,835) in Tables 6.19 and 6.20.

2. There is a strong correlation between the cache access and the vector instructions counters. Comparing the vector instructions counters between the non-SIMD version in Table 6.19 and the counters for the vectorised version, either SIMD-layer or SIMD-range, in Tables 6.20 and 6.21, the number of vector instructions is much higher using BU-SIMD version and remains constant for the layers using the BU-noSIMD version. For those
Table 6.19: Hybrid BFS execution for starting vertex 12, 119 using the **bottom-up non-SIMD version**. PAPI was used to get the hardware counters for L1 and L2 cache misses and vector instructions. The graph size is $SCALE = 18$ and $edgefactor = 32$.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Approach</th>
<th>NV</th>
<th>Time(sec)</th>
<th>L1 misses</th>
<th>L2 misses</th>
<th>Vec. Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD-SIMD</td>
<td>262,143</td>
<td>0.000088</td>
<td>212</td>
<td>210</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>TD-SIMD</td>
<td>262,142</td>
<td>0.000448</td>
<td>323</td>
<td>189</td>
<td>181</td>
</tr>
<tr>
<td>3</td>
<td>BU-noSIMD</td>
<td>259,153</td>
<td>0.000881</td>
<td>1,877</td>
<td>2,168</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>BU-noSIMD</td>
<td>96,758</td>
<td>0.000261</td>
<td>774</td>
<td>752</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>BU-noSIMD</td>
<td>62,373</td>
<td>0.000126</td>
<td>435</td>
<td>151</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>TD-SIMD</td>
<td>62,295</td>
<td>0.000083</td>
<td>204</td>
<td>118</td>
<td>45</td>
</tr>
</tbody>
</table>

layers using the BU-SIMD version, the number of L1 and L2 misses is higher than the ones using the BU-noSIMD version. For example, as can be seen in layers 4 and 5 of *SIMD-range* in Table 6.21 against the other two versions.
3. The difference between the \textit{SIMD-layer} and the \textit{SIMD-range} is in the number of vector instructions counters per layer. On the one hand, the \textit{SIMD-layer} uses vectorisation for the layers above a threshold. For example, in Table 6.20 only the third layer uses the BU-SIMD. On the other hand, the \textit{SIMD-range} introduces a condition, within the implementation, to decide either to use BU-noSIMD or BU-SIMD in all middle layers. Comparing the vector instructions counters between both implementations in Tables 6.20 and 6.21, it is clear that, while the \textit{SIMD-layer} applies vectorisation only the BU-SIMD for the top layer, the \textit{SIMD-range} applies BU-SIMD for all middle layers but the utilisation of the vector unit decreases in consecutive layers. However, this effect does not make a clear impact in terms of performance (execution time). It also can be observed from Tables 6.20 (SIMD-layer) and Table 6.21 (SIMD-range) that for layers 4 and 5, SIMD-range executed more instructions than SIMD-layer (and non-SIMD) but a slight better CPI (since the cycles for both versions are very similar). A precise analysis of this behavior is left for future work since the current emphasis is on an experimental evaluation of the versions.

6.5.3 Performance Analysis

The boxplot graph in Figure 6.10 presents the difference in performance between the three \textit{bottom-up} implementations: the \textit{non-SIMD}, the \textit{SIMD-layer} and the \textit{SIMD-range} for a series of 10 experimental runs of the Graph 500 BFS (as described in Section 4.1.2). Based on the previous analysis of hardware performance counters, this graph confirms that both, the \textit{SIMD-layer} and the \textit{SIMD-range}, implementations achieve similar performance, despite their different implementations, which is higher than the \textit{non-SIMD} version. The main reason for the similar performance of the two SIMD implementations is that they both exploit SIMD completely in the third layer of the example, as can be seen in Tables 6.17 and 6.18, which is a crucial layer because the amount of \textit{non-visited} vertices is larger than layer 4 and 5.

6.6 Hybrid Breadth-First Search Analysis

The results of the hybrid BFS consists of two sets, the first is a comparison between the \textit{non-SIMD} version and the vectorised \textit{SIMD-range} version of the
bottom-up BFS algorithm described in Section 6.3.2. The *SIMD-range* was selected but as it is explained in Section 6.5.3 both versions, SIMD-layer and SIMD-range, present similar performance. By comparing both sets of results, the *non-SIMD* and the *SIMD-range*, it is possible to see the benefit of using vectorisation for the bottom-up approach. The second set of results include a comparison between the hybrid BFS algorithm using vectorisation against the state-of-the-art hybrid BFS algorithm on the Xeon Phi, presented by Gao *et al.* [GLZS14].

To clarify, the results of both versions non-SIMD and the SIMD version are only for the bottom-up BFS algorithm by all means that despite the results shown the hybrid BFS version including the top-down and the bottom-up algorithms only the vectorisation of the bottom-up version is analysed in this section, the analysis of the vectorisation of the top-down has been previously discussed in Chapter 5.
No-SIMD versus SIMD  Figure 6.11 shows the results of the hybrid BFS algorithm using the non-SIMD version and the SIMD-range bottom-up versions for different graph sizes varying the SCALE from 14 up to 20 and for edge factor 16, 32 and 64 (a, b and c in the figure).

1. For the three cases in Figure 6.11 (a, b and c), the performance of the SIMD version (red line) is higher than the non-SIMD (blue line), as was expected as a result of exploiting the vector unit. The speedup is approximated around 250 MTEPS (7.14%), 500 MTEPS (9.09%) and 790 MTEPS (8.98%) respectively. Moreover, the higher the edge factor, the wider is the gap between both versions, keeping a similar shape as long as the graph size increases. The reason for this is that the vectorisation for the bottom-up algorithm does not apply the same strategy for vectorising the adjacency list as the top-down does, which is affected by the edge factor. Instead, the bottom-up sets up a threshold, introduced in Section 6.3.1, to only iterate through certain number of adjacent vertices. Thus, even though the edge factor might be bigger, which means that the number of adjacent vertices is larger, the bottom-up cuts off iterating through all the adjacent vertices due to this static threshold.

2. Along the three plots, there is a “hump” in SCALE 17, 17 and 18 for the graphs in Figure 6.11 a, b and c respectively. In particular the same “hump” can be observed in the implementation of the state-of-the-art Gao et al. [GLZS14] in Figure 6.12. This hump might be caused as an artifact of the Graph 500 during the graph generation, its analysis will part of the future work of this thesis.

SIMD versus the State-of-the-Art  Figure 6.12 shows the results of the hybrid BFS algorithm using the SIMD-range bottom-up version and the results from the state-of-the-art vectorised hybrid BFS algorithm Gao et al. [GLZS14] for different graph sizes varying the SCALE from 14 up to 20 and for edge factor 16, 32 and 64 (a, b and c in the figure).

1. For the three cases in Figure 6.12 (a, b and c), the performance of the SIMD version of this thesis (red line) is higher than the state-of-the-art SIMD hybrid BFS algorithm (green line). The state-of-the-art BFS algorithm is published by Gao et al. [GLZS14] and the results shown in Figure 6.12 (a, b
Figure 6.11: Hybrid BFS performance results comparing the non-SIMD versus the SIMD-range bottom-up version for different RMAT graph sizes, using SCALE from 14 up to 20 and edgefactor of 16, 32 and 64, generated by Graph 500.
and c) are the approximate values manually taken from the data presented in the paper. The approximate maximum speedup in each of the figures (a, b, c) is around 950 MTEPS (33%), 500 MTEPS (6.90%) and 1.50 GTEPS (22%).

Moreover, since the source code of the Gao et al. implementation is not available, it is not possible to make a thorough analysis of the comparison between the results in this thesis and theirs. However, there are some insights that might explain the reasons for achieving better performance.

- Again, the difference in the results is not only related to the results of the vectorisation of the bottom-up, but also to the vectorisation of the top-down BFS algorithm. Then, one of the key factors for outperforming the Gao et al. implementation is due to the fact that the vectorisation for the top-down version has been proven to be faster than the version of Gao et al. [GLZS14] in Section 5.7. However, what is not yet clear is the impact of the vectorisation of the bottom-up version over the complete hybrid BFS algorithm because, as was explained previously in the non-SIMD and SIMD comparison of the bottom-up, despite the fact that the SIMD version is faster, the difference in terms of performance is small.

- Another factor for achieving better performance than Gao et al. [GLZS14] might be simply because of the way the bottom-up algorithm is implemented, the source code can be seen in Listing 9 of Appendix C. This implementation consists of stepping through each word in the visited bitmap array, which means that 32 bits (an integer), representing 32 vertices of the graph, are processed by one thread. That way each thread accesses chunks of memory that are independent of each other, eliminating the usage of atomic operations and the cost of synchronisation.

- Gao et al. [GLZS14] presents results for graph sizes of scale = 22 and edgefactor = 64. However, in this thesis the maximum graph size is for scale = 22 for edgefactor = 16 and scale = 21 for edgefactor = 32. The results presented in Table 6.11 are only for SCALE = 20 and edgefactor of 16, 32 and 64. The reason for not having the same results for different graphs sizes up to scale = 22 is mainly because...
the vectorisation of the *top-down* algorithm uses extra auxiliary data structures for the use of precalculated masks, which requires more allocation of memory as described in Section 5.4.2.

2. The *Graph 500* web-based list keeps an updated record of the performance of parallel computers. The performance is measured by running the BFS in different parallel architectures and according to the performance (number of TEPS) the computers are ranked. According to the most recently published list (June 2016) [gra], an implementation of the hybrid BFS algorithm created by Golovina *et al.* [GSF14] is executed on the same model of the Intel Xeon Phi used in this thesis (5110P), which is ranked in the 147th place in the list. As it is described in Section 2.3.3, this implementation is based on automatic vectorisation with two main optimisations: *loop unrolling* and *prefetching*, resulting in a performance of 1.80143 GTEPS for a graph size of $scale = 23^2$, using 60 cores. Despite the fact that in this thesis it is not possible to get the results for that graph size, due to the increment that the data structures utilised for vectorisation cause, the TEPS achieved for a graph of $scale = 22$ and $edgefactor = 16$ is approximately 3.7 GTEPS, which is far higher than the results from Golovina *et al.* for the $scale = 23$. Even though these results are not directly comparable because Golovina *et al.* do not present the results for a graph size of $scale = 22$, they are indicative of the continuity of the scaling. Though, the difference is that Golovina *et al.* used automatic vectorisation whereas in this work vectorisation is handled manually used by using intrinsic functions.

### 6.7 Conclusions and Future Work

In this chapter, the vectorisation and the analysis of the performance of the *bottom-up* approach of the hybrid BFS algorithm on the Xeon Phi was presented. In particular, three implementations of the BFS *bottom-up*, using different levels of vectorisation, were analysed based on their performance. Furthermore, a comparison between the SIMD hybrid BFS algorithm presented in this thesis and the state-of-the-art was conducted. Therefore, a first contribution of this work

---

2The figure reported in the Graph 500 list does not contain the edgefactor. However, according to the Graph 500 specifications it should be 16 and the results are reporting the harmonic mean
Figure 6.12: Hybrid BFS performance results for the SIMD-range version and the state-of-the-art results presented by Gao et al. [GLZS14] for different graph sizes, using SCALE from 14 up to 20 and edgefactor of 16, 32 and 64, generated by Graph 500.
1. The performance analysis of the vectorised version of the \textit{bottom-up} BFS algorithm conducted in Section 6.5.2, using PAPI library, lead in finding different performance counters that need a thorough understanding. For instance, the CPI (cycles/instructions) that according to [Int12c] the threshold for the CPI in the Xeon Phi architecture should be investigated if it is higher than 4.0. This could help to understand more about the Xeon Phi architecture and the usage of its features crucial for graph traversals including vector unit and cache memory.

2. An investigation of the variability from run-to-run experiments, mentioned in Section 6.5.1, due to the non-deterministic nature of the BFS could help to understand what is the impact of variation over the division of the workload among threads.

3. Finally, the study to find overlaps between different BFS trees in graph traversals, starting with different starting vertices, aiming to find information that can be useful to precalculate the following BFS output trees within the 64 iterations of the Graph 500 experiments. Previous studies of the BFS algorithm have not dealt with the idea of finding BFS trees overlaps which can be potentially used by the centrality graph analysis algorithm, so future work over this subject is needed to confirm this finding.

\footnote{Gao et al. [GLZS14] results are not listed in the Graph 500 list.}
6.8 Summary

In this chapter the concept of the bottom-up approach was first introduced which, combined with top-down approach, results in what is known as the hybrid BFS algorithm based on the parallel layer-synchronous BFS algorithm presented in Section 2.2. This algorithm has been demonstrated to show better performance than the conventional top-down BFS algorithm. For this reason, in this chapter the process of vectorising the hybrid BFS algorithm was presented, consisting of the combination not only of the SIMD version of the top-down, described in Chapter 5, but also of the bottom-up approaches. An online heuristic to switch between both approaches was analysed in Section 6.2.1. As a result of the analysis of this heuristic, some future directions for research, including the variability in performance from run-to-run, are summarised in Section 7.2. Furthermore, in addition to presenting a systematic analysis of the bottom-up algorithm to better understand the impact of the utilisation of the vector unit in the algorithm, this chapter also shows a comparison with the state-of-the-art hybrid BFS algorithm in Section 6.6, showing better performance.
Chapter 7

Conclusions and Future Work

This chapter presents a summary of each one of the chapters presented in this thesis in Section 7.1. Furthermore, Section 7.2 introduces the conclusions drawn from the results obtained for the vectorisation of the BFS algorithm. Moreover, the directions for future research are presented in Section 7.3.

7.1 Summary

Chapter 1 presented the fundamentals of graph theory, including the formal definition of a graph, its representation in a computer and the common graph analysis algorithms. Furthermore, an overview of the evolution of parallel computers and its programming challenges was introduced, starting with the earlier parallel vector processors, continuing with advanced vector processors and ending with the latest trend of heterogeneous architectures. This thesis has been focused on the parallelisation of the BFS using advanced vector processors of the Intel Xeon Phi.

Chapter 2 introduced the parallel BFS algorithm, the challenges behind its parallelisation, and the parallel architectures that have been used for its acceleration. Particularly, studies that have been focused on the parallelisation of the BFS using vectorisation with the Xeon Phi are identified and put in the context of this thesis. The work presented by Gao et al. [GLZS14] is the state-of-the-art but their source code is not available for a comparison.

Chapter 3 described the architecture of the Intel Xeon Phi used in this thesis. This is categorised as a manycore architecture with up to 60 cores, each of them with a vector processor unit that uses the Intel AVX-512 instruction set. The programming model of the Xeon Phi is based on a shared-memory system, which
was also introduced to define the multithreading terminology used during this thesis. Moreover, to analyse the performance of programs, hardware performance counters allow the capture events for hardware specific resources. For instance, performance counters are used for the analysis of the vector instructions in the bottom-up BFS implementation in Chapter 6.

Chapter 4 introduced the Graph 500 benchmark, including the graph generator and the experimental suite for running the BFS experiments. Furthermore, a framework to facilitate the development of multiple implementations was presented and used during the development of the vectorisation of the top-down BFS algorithm in Chapter 5 and the vectorisation of the hybrid BFS algorithm in Chapter 6.

Chapter 5 presented the process of vectorising the conventional top-down BFS algorithm. This algorithm was vectorised using manual vectorisation based on intrinsic functions, part of the AVX-512 instruction set, and optimisations related to the Xeon Phi architecture were applied including data alignment, vector masks and prefetching. This new implementation achieved a higher number of TEPS than previous published results for the same type of Xeon Phi. Furthermore, the impact of thread affinity mapping and hyperthreading on the performance on an underpopulated system was investigated, a topic under researched in the literature. This supported the exploration of the trade-off between the maximum number of threads and their performance degradation in hardware simultaneous threading for the BFS.

Chapter 6 described the bottom-up BFS algorithm, which combined with the top-down results in the state-of-the-art hybrid BFS algorithm. Therefore, vectorising the hybrid BFS algorithm consists of the vectorisation of both the top-down and the bottom-up approaches. The top-down BFS vectorisation was introduced in Chapter 5, whereas the vectorisation of the bottom-up is explained in Chapter 6. A systematic analysis of the bottom-up BFS vectorisation, using hardware performance counters, was conducted using three different implementations: the no-simd, the simd-layer and the simd-range. Moreover, a comparison between the vectorised version of the hybrid BFS algorithm against the state-of-the-art was conducted, demonstrating that the results in this thesis produced better performance.
7.2 Conclusions

Due to the surge of graph algorithms for the analysis of large datasets, these algorithms have become interesting algorithms to optimise. The BFS is an important graph algorithm which is used as a building block for other graph analysis algorithms. The parallelisation of this algorithm has been shown to be challenging due to the irregular memory access patterns, data dependencies and workload imbalance; these are characteristics that limit its scalability. For this reason, different parallel architectures and programming models have been investigated aiming at the acceleration of this important graph algorithm, including recent manycore architectures using advanced vector processing with a high degree of parallelism. The ultimate motivation for this thesis is to optimise the utilisation of advanced vector processors for solving the BFS graph analysis algorithm.

7.2.1 Top-Down Breadth-First Search Vectorisation

The results related to the vectorisation of the conventional top-down algorithm presented in Chapter 5 lead to the following conclusions:

1. The vectorisation of the top-down BFS algorithm using the Intel Xeon Phi vector optimisations: intrinsic functions, data alignment, vector masks and prefetching achieved a higher number of TEPS than the previous published results for the same type of Xeon Phi [GLZS14]. The maximum speedup obtained was approximately 37% (1.37x) for a graph size of one million vertices ($SCALE = 20$ and $edgefactor = 16$). This speedup is mostly due to the vector optimisations mentioned (intrinsic functions, data alignment, vector masks, peel and remainder loops) but nearly half (18%) of this improvement is caused by using software prefetching as described in Section 5.4.2.

2. The investigation of thread affinity presented in Section 5.4.2 shows that by fully occupying each core of the Xeon Phi architecture (4 threads per core) for solving the breadth-first search algorithm, the performance of each thread is degraded by just over three quarters (from 470 MTEPS to 140 MTEPS) relative to that of one thread per core. Similarly, on two threads the performance of each is just over a half of that for the one thread per core
This implies that there is a trade-off between the maximum number of threads and performance degradation caused by hardware multithreading.

### 7.2.2 Hybrid Breadth-First Search Vectorisation

A summary of the main conclusions of the vectorisation of the hybrid BFS algorithm presented in Chapter 6 is listed below:

1. The analysis over the vectorisation of the bottom-up BFS algorithm presented in Section 6.5.3 shown that the maximum performance increase by using the bottom-up BFS algorithm (SIMD-layer) is roughly 9% over the no-SIMD version.

2. A comparison between the vectorised version of the hybrid BFS algorithm against the state-of-the-art [GLZS14] was conducted, demonstrating that the results in this thesis show better performance. An approximate maximum speedup of 33% for graph size of one million vertices ($SCALE = 20$ and $edge\text{factor} = 16$). Furthermore, a comparison with the results of Golovina et al. [GSF14], published in the Graph 500 web list [gra] was conducted. Golovina et al., ranked as the 147th place in the list, achieves up to 1.8 GTEPS for a graph size of $SCALE = 23$, whereas in this thesis the maximum performance is 3.7 GTEPS for a graph size $SCALE = 22$ which is over a 100% increase. However, in this thesis it was not possible to produce results for $SCALE = 23$ due to the utilisation of extra auxiliary data structures for handling pre-calculated masks, restricting the memory space.

### 7.3 Future Work

#### 7.3.1 Towards the Heterogeneous Breadth-First Search

In the future, the plan is to extend the BFS algorithm to explore the benefit of vectorisation techniques beyond their use in native mode on the Xeon Phi, including targeting offload mode. Furthermore, to improve the memory access pattern there are three bitmap arrays, the visited, the output and the frontier [APPB10]. While this data structure is beneficial to reduce the memory size and the number of hits to cache, it also has its limitation. Stepping through each bit
in the bitmap array can be inefficient. An optimisation could be done by reducing the number of bits to be tested by using de Bruijn sequences and reducing the number of conditionals to evaluate [Kan13].

### 7.3.2 Towards Automatic Vectorisation

Despite parallelisation using automatic vectorisation sounding promising, this technique has its limitations when trying to parallelise non-contiguous data and data dependencies, features inherent in the characteristics of the BFS algorithm. In this thesis, automatic vectorisation was explored at first, aiming to let the compiler generate the calls to access the vector unit. However, the implications of using automatic vectorisation for the BFS algorithm were firstly, having to use temporary data structures to assure that the data were stored contiguously in memory. Secondly, for data dependencies, the OpenMP library, using the *pragma simd* directives, was used to force the compiler to vectorise specific segments of code. Nevertheless, this technique violates the compiler safety restriction over data dependencies, which does lead to getting incorrect results, requiring the implementation of a restoration process. For these two reasons, in this thesis manual vectorisation, using intrinsic functions, was utilised. However, intrinsic functions are architecture specific, which make them not portable across different vector architectures. Therefore, studying how to cope with irregular memory access applications, like the BFS, is interesting because it can lead not only to the design of new methods for improving automatic vectorisation but also to enlarge the portability between different architectures.

### 7.3.3 Vectorisation for Performance

This thesis has explored the vectorisation of the state-of-the art BFS algorithm using the Intel Xeon Phi provided with advanced vector processors. In the first place, despite the advantage of using the vector unit for solving the BFS leading to improve performance, roughly around 37% for the *top-down* and 9% for the *bottom-up* BFS algorithms, this improvement is limited due to the irregular memory patterns and data dependencies of the algorithm\(^1\), which limits the usage of the vector unit. Equally important, the improvement in performance of the

\(^{1}\)These are effect at the vector level. At the higher level of thread parallelism there are also effects such as load imbalance
hybrid BFS algorithm is mostly due to the swapping between the top-down and the bottom-up approaches exploiting the previous knowledge of the structure of real-world graphs. Consequently, it is important for the parallel BFS optimisation to investigate not only the design of new data structures so they can be more cache memory friendly, but also to understand the structure and the properties of the input graphs.

### 7.3.4 Helper Threads

One of the possibilities with the Xeon Phi, introduced in Section 3.4, is thread affinity, which allows to pin threads to logical cores following a specific layout. The Xeon Phi provides the capability of having up to 4 hardware multithreading, all of them sharing the resources of one core (cache and memory bandwidth, vector unit, etc.). If a core shares the resources between the four threads then it is fully occupied. Therefore, despite the fact that SMT in the Xeon Phi allows to increase the number of threads, up to 240, the performance per thread might be degraded because the resources per core are shared between four threads. In this thesis, thread affinity was shown to find the trade-off between having a greater total number of threads and threads performance degradation per core in Chapter 5.

The results of the experiments conducted for thread affinity in Section 5.4.2 show that by using more than two threads per core (3 or 4 threads) leads to performance degradation of the BFS. Based on that premise, these two threads can be used for helping the processing of the BFS in other ways. For instance, instead of using the third and fourth thread of each core for solving the BFS, it might be beneficial if those threads (third and fourth) are used to improve the performance of the other two threads per core. Prefetching, a technique to fetch data before it is actually required, was shown to help to improve the performance of the BFS by reducing memory latency in Section 5.4.2. Hence, the third and the fourth threads could be helper threads by prefetching data for the other two threads. Although, having only two threads per core, could limit the maximum number of threads by up to 180 threads, each of those threads will benefit from having two extra helper threads for prefetching data, leading to a research area that could be explored in the future.
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.3.5 SIMD Wrapper

In this thesis, the technique for exploiting the vector unit of the Intel Xeon Phi is manual vectorisation by using intrinsic functions as described in Chapter 5 because of the limitations that automatic vectorisation presented during the BFS vectorisation. However, intrinsic functions are architecture specific, which makes them not portable across different vector architectures. For instance, the ARM Neon parallel architecture [ARM09] contains a 128-bit vector unit with their own instruction set.

A way to cope with the portability of vector instructions between different architectures could be approached by implementing a “SIMD wrapper”. This would be a library to abstract the vector functions required by the BFS algorithm including the scatter and the gather functions used along the vectorisation of the adjacency list explained in Chapter 5. Hence, the main purpose of the “SIMD wrapper” is to hide the details of the each architecture instructions, so the access to them is transparent from the programmer perspective, but more important they would be portable across multiple architectures.

7.3.6 Betweenness Centrality

Based on the variability of the BFS algorithm from run-to run, introduced in Section 6.5.1, when the graph is being traversed from a specific starting vertex, all the connected vertices to that starting vertex are explored, creating a subset of the graph (a strongly connected component in graph terminology), in the case where not all vertices in the graph are connected (in the Graph 500 graphs there are unconnected vertices). Hence, in an experiment, all the starting vertices (the selected random 64) that are connected to the main subset of the graph will be traversing the same subset but since the starting vertex is different, every output bfs tree will be different. According to the similarity in terms of structure between two output bfs trees, the subsets that overlaps in the graph could be used to calculate others output bfs trees by using this information for the rest of the 64 iterations, aiming to reduce their processing time. The analysis of using information to generate other bfs trees with different starting vertices is an interesting, promising and as yet as explored line of the research for graph traversals and this is part of the future work of this thesis (where its possible application in the computation of the centrality graph analysis algorithm.
Initially this research can be built with the Graph 500 benchmark, so some of the 64 iterations might be able to improve performance based on the previous BFS trees that have been found to have overlapping segments. However, finding overlapping segments in the structure of the BFS output tree might also be beneficial for solving the *betweenness centrality* problem. This is a graph analysis algorithm used to determined the *centrality* index of each vertex in the graph, which refers to the role of that vertex in the structure of the graph [Bra01]. This algorithm is based on counting the shortest paths using graph traversal algorithms including the BFS. Therefore, the BFS is called many times to find the shortest paths. During the several executions of the BFS, different overlapping segments can be found in advance, so future calls might be able to use that information aiming to reduce the time of finding a valid BFS output tree. No previous study has addressed the question about the overlapping segments between different BFS output trees of different graph traversals, with different starting vertices. Exploring the impact that overlapping segments can make on performance is interesting, especially because it can be useful to optimise other graph analysis algorithms such as the betweenness centrality, which is used during the graph analysis of large datasets.
Appendix A

Graph 500 Harmonic Mean

Tables A.1 and A.2 present an example of how Graph 500 calculates the harmonic mean of the number of TEPS for a graph size of SCALE 18 and edgefactor 16. Due to the existence of zero number of TEPS (17), the harmonic mean ends up in a bigger number (1,196,221,199) than the actual maximum number of TEPS (1,193,649,977).
### Appendix A. Graph 500 Harmonic Mean

#### Table A.1: Example of the Graph 500 calculation of the harmonic mean for a graph size of SCALE = 18 and edgefactor = 16.

<table>
<thead>
<tr>
<th>#Sample</th>
<th>Starting vertex</th>
<th>TEPS</th>
<th>( \frac{1}{\text{TEPS}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5277</td>
<td>423,746,281</td>
<td>0.00000000235990271476</td>
</tr>
<tr>
<td>2</td>
<td>13138</td>
<td>918,243,389</td>
<td>0.00000000108903588329</td>
</tr>
<tr>
<td>3</td>
<td>15762</td>
<td>970,956,253</td>
<td>0.00000000102991251862</td>
</tr>
<tr>
<td>4</td>
<td>15929</td>
<td>875,044,647</td>
<td>0.00000000114279883193</td>
</tr>
<tr>
<td>5</td>
<td>20025</td>
<td>1,193,649,977</td>
<td>0.00000000083776653067</td>
</tr>
<tr>
<td>6</td>
<td>22585</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>26461</td>
<td>978,150,190</td>
<td>0.0000000010223788813</td>
</tr>
<tr>
<td>8</td>
<td>26953</td>
<td>815,714,567</td>
<td>0.0000000012591901702</td>
</tr>
<tr>
<td>9</td>
<td>30126</td>
<td>952,348,991</td>
<td>0.00000000105003523860</td>
</tr>
<tr>
<td>10</td>
<td>30413</td>
<td>877,144,729</td>
<td>0.00000000114006271422</td>
</tr>
<tr>
<td>11</td>
<td>32901</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>34446</td>
<td>966,402,463</td>
<td>0.00000000103476557508</td>
</tr>
<tr>
<td>13</td>
<td>38546</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>44995</td>
<td>884,216,651</td>
<td>0.00000000113094454712</td>
</tr>
<tr>
<td>15</td>
<td>47004</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>51654</td>
<td>913,655,612</td>
<td>0.0000000010945043097</td>
</tr>
<tr>
<td>17</td>
<td>51899</td>
<td>1,112,346,561</td>
<td>0.00000000089900039673</td>
</tr>
<tr>
<td>18</td>
<td>55611</td>
<td>916,487,944</td>
<td>0.00000000109112182733</td>
</tr>
<tr>
<td>19</td>
<td>55996</td>
<td>856,510,273</td>
<td>0.00000000116752831963</td>
</tr>
<tr>
<td>20</td>
<td>57315</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>58247</td>
<td>1,130,876,263</td>
<td>0.00000000088427004118</td>
</tr>
<tr>
<td>22</td>
<td>59622</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>59668</td>
<td>962,524,492</td>
<td>0.00000000103893460211</td>
</tr>
<tr>
<td>24</td>
<td>62454</td>
<td>1,035,630,969</td>
<td>0.00000000096559491725</td>
</tr>
<tr>
<td>25</td>
<td>62875</td>
<td>933,913,419</td>
<td>0.00000000107076307071</td>
</tr>
<tr>
<td>26</td>
<td>68396</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>70108</td>
<td>644,707,216</td>
<td>0.00000000155109168140</td>
</tr>
<tr>
<td>28</td>
<td>72273</td>
<td>830,697,839</td>
<td>0.00000000120380715186</td>
</tr>
<tr>
<td>29</td>
<td>74631</td>
<td>897,011,050</td>
<td>0.0000000011481346906</td>
</tr>
<tr>
<td>30</td>
<td>76484</td>
<td>965,757,820</td>
<td>0.00000000103545628029</td>
</tr>
<tr>
<td>31</td>
<td>78473</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>79771</td>
<td>888,832,001</td>
<td>0.00000000112507200310</td>
</tr>
<tr>
<td>33</td>
<td>80851</td>
<td>1,086,223,748</td>
<td>0.00000000092062063809</td>
</tr>
<tr>
<td>34</td>
<td>81171</td>
<td>722,091,475</td>
<td>0.00000000138486609331</td>
</tr>
<tr>
<td>35</td>
<td>81513</td>
<td>632,536,670</td>
<td>0.00000000158093601166</td>
</tr>
<tr>
<td>36</td>
<td>82424</td>
<td>790,799,698</td>
<td>0.00000000126454271834</td>
</tr>
<tr>
<td>37</td>
<td>88777</td>
<td>1,118,944,236</td>
<td>0.00000000089369958591</td>
</tr>
<tr>
<td>38</td>
<td>91358</td>
<td>681,773,651</td>
<td>0.00000000146676246417</td>
</tr>
<tr>
<td>39</td>
<td>92870</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>103315</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.1: Example of the Graph 500 calculation of the harmonic mean for a graph size of SCALE = 18 and edgefactor = 16.
<table>
<thead>
<tr>
<th>Sample Number</th>
<th>Starting vertex</th>
<th>TEPS</th>
<th>$\overset{1}{TEPS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>105804</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>106797</td>
<td>1,068,442,775</td>
<td>0.000000000039594156210</td>
</tr>
<tr>
<td>43</td>
<td>110559</td>
<td>972,030,505</td>
<td>0.000000000102877429747</td>
</tr>
<tr>
<td>44</td>
<td>116299</td>
<td>1,037,861,753</td>
<td>0.00000000096351946372</td>
</tr>
<tr>
<td>45</td>
<td>119903</td>
<td>939,073,756</td>
<td>0.000000000106487908248</td>
</tr>
<tr>
<td>46</td>
<td>122030</td>
<td>1,026,934,649</td>
<td>0.00000000097377179882</td>
</tr>
<tr>
<td>47</td>
<td>122113</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>125227</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>125847</td>
<td>752,334,755</td>
<td>0.00000000013291953943</td>
</tr>
<tr>
<td>50</td>
<td>128709</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>135134</td>
<td>1,038,151,010</td>
<td>0.000000000096325100184</td>
</tr>
<tr>
<td>52</td>
<td>138967</td>
<td>1,081,457,447</td>
<td>0.00000000092467808416</td>
</tr>
<tr>
<td>53</td>
<td>144001</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>144505</td>
<td>810,569,438</td>
<td>0.00000000123370059682</td>
</tr>
<tr>
<td>55</td>
<td>144955</td>
<td>849,313,282</td>
<td>0.00000000117742183134</td>
</tr>
<tr>
<td>56</td>
<td>147917</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>149208</td>
<td>1,094,231,851</td>
<td>0.00000000091388310321</td>
</tr>
<tr>
<td>58</td>
<td>151942</td>
<td>840,770,092</td>
<td>0.00000000118938578974</td>
</tr>
<tr>
<td>59</td>
<td>158917</td>
<td>915,703,790</td>
<td>0.00000000109205619866</td>
</tr>
<tr>
<td>60</td>
<td>160962</td>
<td>1,047,136,920</td>
<td>0.00000000095498495088</td>
</tr>
<tr>
<td>61</td>
<td>166326</td>
<td>652,625,876</td>
<td>0.00000000153227145454</td>
</tr>
<tr>
<td>62</td>
<td>167943</td>
<td>730,357,595</td>
<td>0.00000000136919230645</td>
</tr>
<tr>
<td>63</td>
<td>168463</td>
<td>963,357,289</td>
<td>0.00000000103803647076</td>
</tr>
<tr>
<td>64</td>
<td>174072</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Zero TEPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max_TEPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harmonic mean</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Example of the Graph 500 calculation of the harmonic mean for a graph size of SCALE = 18 and edgefactor = 16. Continuation of Table A.1.
Appendix B

Top-Down SIMD Source Code

This Appendix shows the source code of the vectorization of the top-down BFS algorithm described in Chapter 5.
APPENDIX B. TOP-DOWN SIMD SOURCE CODE

Listing 4: Source code of the SIMD version of the top-down BFS algorithm.

```c
/* 1.- Load adjacency list to the register */
__m512i vneig = _mm512_load_epi32(&rows[index * 16]);

/* Getting word and bit offset */
__m512i vword = _mm512_div_epi32(vneig, _mm512_set1_epi32(BITS_PER_WORD));
__m512i vbits = _mm512_rem_epi32(vneig, _mm512_set1_epi32(BITS_PER_WORD));

/* Gathering words from visited bitmap array */
_mm512_prefetch_i32gather_ps(vword, explored->start, sizeof(word_t), _MM_HINT_T0);
_mm512_prefetch_i32gather_ps(vword, queue->start, sizeof(word_t), _MM_HINT_T0);

__m512i vis_words = _mm512_i32gather_epi32(vword, explored->start, sizeof(word_t));
__m512i out_words = _mm512_i32gather_epi32(vword, queue->start, sizeof(word_t));

/* Shifting 1 to the left indexes position in the vneig array */
__m512i bits = _mm512_sllv_epi32(_mm512_set1_epi32(1), vbits);

__mask16 mask = _mm512_knot(_mm512_kor(_mm512_test_epi32_mask(vis_words, bits), _mm512_test_epi32_mask(out_words, bits)));

/*3.- Scattering P (bfs_tree) and output queue */
_mm512_mask_prefetch_i32scatter_ps(bfs_tree, mask, vneig, sizeof(word_t), _MM_HINT_T0);

_mm512_mask_i32scatter_epi32(bfs_tree, mask, vneig, _mm512_set1_epi32(vertex - nodes), sizeof(word_t));

/* Setting the output queue */
//Adding to the output queue word the new bit values depending on the mask.
__m512i new_values = _mm512_mask_or_epi32(_mm512_set1_epi32(0), mask, out_words, bits);

_mm512_mask_prefetch_i32scatter_ps(queue->start, mask, vword, sizeof(word_t), _MM_HINT_T0);

_mm512_mask_i32scatter_epi32(queue->start, mask, vword, new_values, sizeof(word_t));
```
Appendix C

The Bottom-Up BFS Source Code

This Appendix presents the source code of the bottom-up BFS implementations introduced in Chapter 6.

```c
void SBFS_2QBM_hybrid_simd::bottom-up-nosimd-adj(int v, int start, int end, bitmap_t *frontier, bitmap_t *queue, bitmap_t *explored){

    for(int i=start ; i<end ; i++) { //Only connected vertices
        int n = rows[i];
        if(bm_get_bit(frontier, n)) { //is in frontier
            bfs_tree[v] = n;
            bm_set_bit(queue, v, 0);
            bm_set_bit(explored, v, 0);
            break;
        }
    }
}
```

Listing 5: Source code of the bottom-up no-SIMD function.
if (topdown){
    start_papi();
    start_time = omp_get_wtime();
    top_down(&queue_bitmap1, &queue_bitmap2, &explored, queue_nwords, 0,
             ini_vertex, layer);
    restoration_opt(&queue_bitmap2, &explored);
    end_time = omp_get_wtime();
    stop_papi();
    printf_papi(level, end_time - start_time);
} else {  // bottom-up
    if((nodes*THRESHOLD) < (nodes - nv)){ // bottom-up SIMD layer
        start_papi();
        start_time = omp_get_wtime();
        bottom_up_simd(&queue_bitmap1, &queue_bitmap2, &explored, queue_nwords);
        end_time = omp_get_wtime();
        stop_papi();
        printf_papi(level, end_time - start_time);
    } else { // bottom-up no SIMD
        start_papi();
        start_time = omp_get_wtime();
        bottom_up_nonsimd(&queue_bitmap1, &queue_bitmap2, &explored, queue_nwords);
        end_time = omp_get_wtime();
        stop_papi();
        printf_papi(level, end_time - start_time);
    }
}

Listing 6: Program control for the hybrid BFS algorithm activated with PAPI counters.
void SBFS_2QBM_hybrid_simd::bottom-up-no-simd(bitmap_t *frontier, bitmap_t *queue, bitmap_t *explored, int nwords){
    #pragma omp parallel for schedule (static, 8)
    for(int i=0 ; i < nwords ; i++){
        word_t val = bm_get_word(explored, i);
        int vertex;
        if(val == 0xFFFFFFFF) continue; //all are visited

        //Scan the bits in the word.
        for(vertex = 0 ; vertex < ULONG_BITS ; vertex++){
            int bit = (int)((val >> vertex) & 1);
            if(bit) continue; // Skip any that is set.
            int vertex_real = i*ULONG_BITS + vertex;
            word_t start = gstart[vertex_real];
            word_t end = gend[vertex_real];
            for(int i=start ; i<end ; i++){
                //Adjacency list
                int n = rows[i];
                if(bm_get_bit(frontier, n)){ //is in frontier
                    bfs_tree[vertex_real] = n;
                    bm_set_bit(queue, vertex_real);
                    bm_set_bit(explored, vertex_real);
                    break;
                }
            }
        }
    }
}

Listing 7: Source code of the bottom-up BFS algorithm for the non-SIMD version.
void SBFS_QBM_hybrid_simd::bottom-up-simd(bitmap_t *frontier, bitmap_t *queue, bitmap_t *explored, int nwords) {
    int segment = 0;
    #pragma omp parallel for schedule (static, 8) private(segment, vertices)
    for(int i = 0; i < nwords; ++i) {
        word_t val = bm_get_word(explored, i);
        if(val == 0xFFFFFFFF) continue; // all are visited
        int fend = 0;
        // Processing the two words
        for(int m = 0; m < 2; m++) {
            int pos = 0;
            segment = (val>>(m*VNELE8))&0xFFFF; // low 16 bits part
            if(segment == 0xFFFF)
                continue;
            fend = segment;
            int vertex = (i*32)+(m*16);
            // The number of visited is small
            if(__builtin_popcount(fend) < T2_SIMD) { // SIMD range
                fend = 0;
                pos = 0;
                // Processing the 16 bits of the visited
                for(int k = 0; k<16; k++) {
                    vertices[k] = vertex + k;
                }
                // Initializing vectors
                __m512i vstart = _mm512_load_epi32(&gstart[vertices[0]]);
                // Loading all the vertices array to the vector register
                __m512i vend = _mm512_load_epi32(&gend[vertices[0]]);
                __m512i vvertices = _mm512_load_epi32(vertices);
                for(pos = 0; pos<T1_SIMD; pos++) {
                    // Gather neighbors to vector register
                    explorv_bu_simd16(segment, frontier, queue, explored, i, m, fend, pos, vstart, vend, vvertices);
                }
                if(fend != 0xFFFFFFFF){
                    for(int b = 0; b < 16 ; b++) {
                        if(fend&(1<<b)) continue;
                    }
                } else {
                    int unvisited = 0;
                    for(int b = 0 ; b < 16 ; b++) {
                        if(fend&(1<<b)) continue;
                        unvisited = vertex + b;
                        bottom-up-nosimd-adj(unvisited, getRngStart(unvisited) + pos, getRngEnd(unvisited), frontier, queue, explored);
                    }
                }
            } else {
                int pos = 0;
                segment = (val>>((n+VNELE8))&0xFFFF); // low 16 bits part
                if(segment == 0xFFFF)
                    continue;
                fend = segment;
                int vertex = (i*32)+(m*16);
                // The number of visited is small
                if(__builtin_popcount(fend) < T2_SIMD) { // SIMD range
                    fend = 0;
                    pos = 0;
                    // Processing the 16 bits of the visited
                    for(int k = 0; k<16; k++) {
                        vertices[k] = vertex + k;
                    }
                    // Initializing vectors
                    __m512i vstart = _mm512_load_epi32(&gstart[vertices[0]]);
                    // Loading all the vertices array to the vector register
                    __m512i vend = _mm512_load_epi32(&gend[vertices[0]]);
                    __m512i vvertices = _mm512_load_epi32(vertices);
                    for(pos = 0; pos<T1_SIMD; pos++) {
                        // Gather neighbors to vector register
                        explorv_bu_simd16(segment, frontier, queue, explored, i, m, fend, pos, vstart, vend, vvertices);
                    }
                    if(fend != 0xFFFFFFFF){
                        for(int b = 0; b < 16 ; b++) {
                            if(fend&(1<<b)) continue;
                        }
                    } else {
                        int unvisited = 0;
                        for(int b = 0 ; b < 16 ; b++) {
                            if(fend&(1<<b)) continue;
                            unvisited = vertex + b;
                            bottom-up-nosimd-adj(unvisited, getRngStart(unvisited) + pos, getRngEnd(unvisited), frontier, queue, explored);
                        }
                    }
                }
            }
        }
    }
}

Listing 8: Source code of the bottom-up SIMD version.
APPENDIX C. THE BOTTOM-UP BFS SOURCE CODE

Listing 9: Source code of the bottom-up SIMD function.

```c
inline void SBFS_QBM_hybrid_simd::LookingParents(int vis_mask, bitmap_t *frontier, bitmap_t *queue, bitmap_t *explored, int pword, int psegment, int &fend, int pos, __m512i vstart, __m512i vend, __m512i vvertices)
{
    __m512i vtmp = _mm512_set1_epi32(pos);
    __m512i vadd = _mm512_add_epi32(vstart, vtmp);
    __mmask16 vcmp = _mm512_cmpgt_epi32_mask(vend, vadd);
    __m512i vneig = _mm512_mask_i32gather_epi32(_mm512_set1_epi32(0), vcmp, vadd, rows, sizeof(word_t));
    int res = 0;
    vis_mask = (explored->start[pword]>>(psegment*VNELE8))&0xFFFF;
    // high 16 bits part
    __mmask16 mask1 = _mm512_kand(_mm512_knot(_mm512_int2mask(vis_mask)),vcmp);
    // filter visited adjacent vertices according to vis and out
    // Getting WORD offset and BIT offset
    __m512i vword = _mm512_srlv_epi32(vneig, _mm512_set1_epi32(5));
    __m512i vbits = _mm512_and_epi32(vneig,_mm512_set1_epi32(0x1F));
    __m512i fron_words = _mm512_mask_i32gather_epi32(_mm512_set1_epi32(0), mask1, vword, frontier->start, sizeof(word_t));
    // Shifting 1 to the left indexes position in the vneig array
    __m512i bits= _mm512_sllv_epi32(_mm512_set1_epi32(1), vbits);
    // Filtering with the unvisited mask
    __mmask16 mask = _mm512_mask_test_epi32_mask(mask1 ,fron_words, bits);
    res = _mm512_mask2int(mask);
    if(mask != 0x0000 ) { // at least one neighbour is in the frontier
        __mmask16 mask1 = _mm512_mask_kand(_mm512_knot(_mm512_int2mask(vis_mask)),vcmp);
        // prefetch neighbouring position of the BFS tree
        _mm512_mask_i32scatter_epi32(bfs_tree, mask, vvertices,sizeof(word_t), _MM_HINT_T0);
        explored->start[pword] |= res<<(psegment*VNELE8);
        queue->start[pword] |= res<<(psegment*VNELE8);
    }
    // flag used to indicate the end of the processing
    fend |= (res | ((explored->start[pword])<<(psegment*VNELE8))&0xFFFF));
}
```
Appendix D

SIMD Intrinsic Functions

This Appendix lists the SIMD intrinsic functions used in the vectorisation of the top-down and the hybrid BFS algorithms in Chapters 5 and 6. The concept of intrinsic functions and vector masks are introduced in Section 3.4.2. A full description of these intrinsic functions can be found in the Intel website [Int]. All functions listed work for 16 32-bit elements.

- _mm512_load_epi32_ Loads 32-bit integers to the vector register.
- _mm512_set1_epi32_ Broadcast 32-bit integer to the elements of a vector register.
- _mm512_add_epi32_ Adds 32-bit integer vectors.
- _mm512_div_epi32_ Divide 16 32-bit integers by a 32-bit integer, returning the result in a vector register.
- _mm512_rem_epi32_ Gets the remainder resulted by the division of two sets of 32-bit integers into a vector register.
- _mm512_cmpgt_epi32_mask_ Performs a comparison between 32-bit integer vectors.
- _mm512_int2mask_ Converts an integer value to a vector mask.
- _mm512_i32gather_epi32_ Gathers 32-bit elements from memory using 32-bit indices to a vector register.
- _mm512_prefetch_i32gather_ps_ Gathers prefetch 32-bit integer vector with 32-bit integer indices.
APPENDIX D. SIMD INTRINSIC FUNCTIONS

_mm512_mask_i32scatter_epi32  Scatter 32-bit integer vector with 32-bit integer indices.

_mm512_mask_prefetch_i32scatter_ps  Scatter prefetch 32-bit integer vector with 32-bit integer indices.

_mm512_sllv_epi32  Takes two int32 vectors and performs a bitwise left shift on the first parameter.

_mm512_srlv_epi32  Performs a logical right shift operation of the 32-bit integer vector.

_mm512_kand  Performs a bitwise AND operation between two vector masks.

_mm512_kor  Performs a bitwise OR operation between two vector masks.

_mm512_knot  Performs a bitwise NOT operation on a vector mask.

_mm512_and_epi32  Bitwise AND operation between 32-bit integer vectors.

_mm512_mask_or_epi32  Bitwise OR operation between 32-bit integer vectors.

_mm512_test_epi32_mask  Performs bitwise AND operation between 32-bit integer vectors.
Appendix E

Acronyms

The list of abbreviations in this thesis.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycles per Instructions</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>EDF</td>
<td>Experimental Development Framework</td>
</tr>
<tr>
<td>FIFO</td>
<td>First Input First Output</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>IQR</td>
<td>Interquartile Range</td>
</tr>
<tr>
<td>MIC</td>
<td>Manycore Integrated Circuit</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
<tr>
<td>MMX</td>
<td>Multimedia Extensions</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple Instruction Single Data</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>PAPI</td>
<td>Performance Application Programming Interface</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RMAT</td>
<td>Recursive Matrix</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction Single Data</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multithreading</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD Extensions</td>
</tr>
<tr>
<td>TEPS</td>
<td>Traversed Edges per Second</td>
</tr>
<tr>
<td>VPU</td>
<td>Vector Processing Unit</td>
</tr>
</tbody>
</table>
Bibliography


11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*.


[WL] Jeremiah Willcock and Andrew Lumsdaine. Graph 500 project on Gitorious.


BIBLIOGRAPHY

