# High Performance Graph Analysis on Parallel Architectures

Athanasios K. Grivas

School of Electical & Electronic Engineering

Newcastle University

A thesis submitted for the degree of

*Doctor of Philosophy*

September 2015

# Abstract

Over the last decade pharmacology has been developing computational methods to enhance drug development and testing. A computational method called network pharmacology uses graph analysis tools to determine protein target sets that can lead on better targeted drugs for diseases as Cancer. One promising area of network-based pharmacology is the detection of protein groups that can produce better effects if they are targeted together by drugs. However, the efficient prediction of such protein combinations is still a bottleneck in the area of computational biology.

The computational burden of the algorithms used by such protein prediction strategies to characterise the importance of such proteins consists an additional challenge for the field of network pharmacology. Such computationally expensive graph algorithms as the all pairs shortest path (APSP) computation can affect the overall drug discovery process as needed network analysis results cannot be given on time. An ideal solution for these highly intensive computations could be the use of super-computing. However, graph algorithms have data-driven computation dictated by the structure of the graph and this can lead to low compute capacity utilisation with execution times dominated by memory latency.

Therefore, this thesis seeks optimised solutions for the real-world graph problems of critical node detection and effectiveness characterisation emerged from the collaboration with a pioneer company in the field of network pharmacology as part of a Knowledge Transfer Partnership (KTP) / Secondment (KTS). In particular, we examine how genetic algorithms could benefit the prediction of protein complexes where their removal could produce a more effective 'druggable' impact. Furthermore, we investigate how the problem of all pairs shortest path (APSP) computation can be benefited by the use of emerging

parallel hardware architectures as GPU- and FPGA- desktop-based accelerators.

In particular, we address the problem of critical node detection with the development of a heuristic search method. It is based on a genetic algorithm that computes optimised node combinations where their removal causes greater impact than common impact analysis strategies. Furthermore, we design a general pattern for parallel network analysis on multi-core architectures that considers graph's embedded properties. It is a divide and conquer approach that decomposes a graph into smaller subgraphs based on its strongly connected components and computes the all pairs shortest paths concurrently on GPU. Furthermore, we use linear algebra to design an APSP approach based on the BFS algorithm. We use algebraic expressions to transform the problem of path computation to multiple independent matrix-vector multiplications that are executed concurrently on FPGA. Finally, we analyse how the optimised solutions of perturbation analysis and parallel graph processing provided in this thesis will impact the drug discovery process.

# Publications

Part of this work has been published in:

- **A.K. Grivas**, T. Mak, A. Yakovlev, J. Wray, "Novel Multi-Layer Network Decomposition boosting acceleration of multi-core algorithms ", in Application-Specific Systems, Architectures and Processors (ASAP), 24th International IEEE Conference, 2013, Washington
- **A.K. Grivas**, T. Mak, A. Yakovlev, J. Wray, A.Mokhov, "Tailoring Graph Analysis algorithms over GPUs", in IEEE Transactions on Computers (submitted September 2015)

# Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Alex Yakovlev. His ability to convey his enthusiasm and passion for research kept me motivated during the hard times. I am also grateful to Dr. Terrence Mak, who was the one that introduced me in the world of research and guided me in the first steps of my PhD.

My sincere thanks also goes to my co-advisor, Dr. Andrey Mokhov, who has been always there to give me a good piece of advice. I would also like to thank Dr. Jonny Wray for his practical advices and insightful discussions that we had during my PhD. Finally, I gratefully acknowledge the encouragement of my family and the patience of my friends throughout this journey.

This research was part of a Knowledge Transfer Partnership (KTP) and Knowledge Transfer Secondment (KTS) between e-therapeutics PLC and Newcastle University. It was supported as a collaborative project by e-therapeutics PLC and Technology Strategy board (Partnership Reference No: 8736).

# Contents

# Nomenclature

$\bar{d}$ Average connected distance

$\Delta$ Double Layer data structure

$\Delta_{Down}$ Lower layer of $\Delta$ data structure

$\Delta_{normal}$ Normalised $\Delta$ data structure

$\Delta_{Up}$ Upper layer of $\Delta$ data structure

$\mu$ Layers of network decomposition

$APSP$ All Pairs Shortest Paths

$BFS$ Breadth First Search

$C$ Condensed strongly connected component to a single node

$C_{num}(G)$ Number of components in $G$

$CN_{betweeness}$ Betweeness centrality

$CN_{closeness}$ Closeness centrality

$CN_{degree}$ Degree centrality

$CSR$ Compressed Sparse Row

$D[v]$ List that stores the distance of a shortest path from $s$ to $v$

$D^i_{sp}[]$ Shortest Paths in $\Delta_{Up}$

$E^{SCC}$ A set of edges of $G^{SCC}$

$FPGA$ Field Programmable Gate Array

$G$ Graph with $V$ nodes and $E$ edges

$G^{SCC}$ Component Network

$G_{cal}$ Compact Adjacency List of $G$

$G^{scc}_{cal}$ Compact Adjacency List of $G^{SCC}$

$GPU$ Graphic Processing Unit

$HPC$ High Performance Computing

$N_B(k)$ Number of blocks of threads

$N_t(k)$ Number of threads

$P[v]$ List that stores the vertex parent of $v$

$PPI$ Protein Protein Interactions

$S_i$ Subnetwork in $\Delta_{Down}$

$S_{pi}$ Union of all $S_{sp}$ results

$S_{sp}$ SP in each subnetwork $S$

$SCC$ A set of strongly connected components

$SP$ Shortest Paths

$SSSP$ Single Source Shortest Paths

$u_i$ Condensed component represented as single node in $\Delta_{Up}$

$V^{SCC}$ A set of nodes of $G^{SCC}$

# Chapter 1

# Introduction

## 1.1 Motivation

Various natural and scientific phenomena can be modelled as graphs. The representation of complex data as graphs can provide a sophisticated way for computational analysis as it reveals significant and non-obvious systemic properties. Over the last decade, the use of graphs was expanded for the representation of various sized complex systems ranging from the World-Wide Web [1] till brain connectivity [2]. Network pharmacology consists an emerging field that combines both experiments and computation for the development of new drugs by using protein-protein interactions represented as graphs [3]. This thesis provides optimised solutions to real-world graph problems emerged from the collaboration of e-Therapeutics PLC [4], a pioneer company in the field of Network Pharmacology, and Newcastle University in the framework of a Knowledge Transfer Partnership (KTP) / Secondment (KTS).

Over the last century, the pharmaceutical industry developed drugs by identifying 'drugable' proteins that can be used for the development of compounds with desired actions against such proteins [5, 6, 7]. A great percentage of drugs function by binding to particular proteins in order modify their biochemical and biophysical operations, however, they cause side effects on a variety of other non-targeted functions [8]. In contrast, network pharmacology uses a more effective, targeted and systematic drug discovery approach. It employs network analysis to

identify the most critical group of proteins in any disease and chemical biology to determine molecules that can target that set of proteins [4].

Proteins are functioning as components of highly interconnected cellular networks known as interactome or Protein Protein Interaction (PPI) networks [9, 10]. Nowadays, the size of PPIs is not limited only to few vertices and edges. The amount of data generated from the analysis of real-world biological phenomena as Protein-protein interactions is getting larger [11]. Accordingly, the size and complexity of graphs representing such data is also getting increased. Their overall behaviour cannot be easily understood or predicted only by examining individual nodes or links [12]. Graph analysis, through the computation of topological features as network centrality, plays a decisive role in the overall comprehension of the properties governing real-world graphs.

In particular, the analysis of PPIs allows the detection of specific nodes that could be used as effective targets for drug intervention [13]. Instead of focusing on particular druggable targets, it was shown that is more advantageous to target a set of proteins [14]. A promising area in network-based pharmacology is the ability to compute combinations of protein complexes, which will produce better synergistic effects when targeted together [15]. However, computing efficient protein combinations is still a bottleneck in computational biology despite many already developed computational techniques based on network centrality features [16]. Apart from optimised graph analysis tools that can lead on better targeted drugs, computational biology needs tools of higher performance in order to accelerate further the drug discovery process. However, this premises the existence of efficient tools that are not restricted by the high computational burden of graph algorithms as all-pairs shortest path (APSP) computation which consists a major component for the computation of most network centrality measures.

As the scale and complexity of graph problems is getting increased due to exponential generation of molecular data [17, 18], common processing units as CPUs cannot cope with the need for high memory and computing resources. Parallel graph processing seems to be an ideal solution that can overcome the limitations of single-core processors. However, the parallelization of graph algorithms is more challenging in comparison to common scientific applications [19]. Graph data are characterised by non-uniform distribution of structural properties that can lead

on poor performance. On the other hand, the graph algorithms developed during the last decade by the computer science community cannot satisfy the new additional design factors. Both the complex structure and size of graphs as well as the computer architectures used for the solution of such graph problems can affect their performance.

Despite the great significance of graph analysis, research on optimised perturbation analysis and parallel graph processing is still in early stages. We are facing daunting challenges due to the irregular structure of the graphs and the fact that processors are getting increasingly parallel [19, 20]. Many graph algorithms are still inherently sequential. In order to fully exploit the emerging parallel architectures we need to design new parallel graph algorithms. The most prominent high performance architectures nowadays are the Graphic Processing Units (GPUs) and Field programmable gateway arrays (FPGAs). While GPUs and FPGAs are ideal HPC accelerators for many applications [21], their exploitation for parallel graph processing makes it more challenging as we need to deal with both the complexity of a parallel architecture and the abnormal nature of graph applications that is not ideal for parallel processors.

## 1.2 Research goal and main contributions

This thesis targets to resolve real-life problems related with the optimisation and acceleration of the drug discovery process. Network pharmacology uses graph analysis tools to develop better targeted drugs. Computing efficiently combinations of proteins that will create a better 'drugable' impact is still a bottleneck for computational biology. Additionally, graph algorithms as APSP is a major component for the computation of most network centrality measures, however, is highly computationally expensive. The research question of this thesis is *how evolutionary and high performance computing can optimise and accelerate graph analysis?* This can be decomposed into a number of sub-problems:

- How genetic algorithms can optimise graph analysis and eventually help to design better targeted drugs?
- How the properties of real-world graphs can favour the development of

better parallel graph algorithms?

- How linear algebraic approaches for parallel graph algorithms can lead on better performance in comparison to classical sequential graph algorithms?

In the course of responding the research questions of our thesis, in each chapter we make contributions in the field of perturbation analysis and parallel graph processing through novel algorithmic implementations. We develop novel approaches for optimised and high performance graph analysis with on-desktop parallel accelerators as GPU and FPGA. In particular, we focus in the design of a genetic algorithm that provides a heuristic search capable to detect nodes of critical importance within a graph. We target to accelerate the APSP computation by a factor of more than two times. Such acceleration will affect positively the performance of our genetic algorithm that uses the APSP algorithm as a basis for its fitness operator. It will help to reduce the overall execution time from several hours to minutes. The main contributions of this thesis are the following:

- I developed a genetic algorithm (GA) that searches for highly optimised node removals that can achieve higher impact than random and targeted attacks. It uses a population of boolean strings that represent different node removal patterns. Over time the survival of the fittest candidates favours better combinations of node removals. As this process is repeated over hundred times, it eventually converges in a well estimated combination of nodes where their removal produce a more efficient impact in graph's robustness.

- I developed a general pattern for graph analysis on multi-core GPUs by exploiting the properties of the analysed graphs. The MLND algorithm has a multi-functional character that is novel in the area of parallel network processing. It uses a data structure that can be used to control the balance between the number of the cores that a multi-core processor contains and the number of components that are going to be analyzed in parallel. At the same time this approach acts as a compressor while is able to decompose a network into smaller modules without losing information regarding its

initial state and process each component concurrently in order to compute the all pairs shortest path.

- I developed a linear based graph algorithm that computes the APSP on FPGA based on a sparse matrix vector (SpMV) multiplication approach that is highly concurrent. The all pairs shortest path computation is implemented through a breadth first search (BFS) algorithm based on linear algebra. New nodes are discovered though consecutive multiplications of the transposed adjacency matrix of the graph multiplied with a vector that its values denote the source and the discovered nodes. This approach provides us with an embarrassingly parallel problem where graph data dependencies are no more existed and can be easily implemented on dataflow computing.

## 1.3 Thesis Outline

The thesis consists of six chapters:

Chapter 2 (*Background*) introduces the basic terminology, concepts and latest research in two main areas involved in this work: perturbation analysis and parallel APSP implementations on GPU- and FPGA-based architectures.

Chapter 3 (*Optimised perturbation analysis*) presents a genetic algorithm for the computation of optimised node combinations that their removal causes more effective impact in graph's robustness than common strategies.

Chapter 4 (*Tailoring graph algorithms over GPUs: Multi-Layer Graph Decomposition*) presents a divide and conquer approach that its design was based on complex networks' embedded properties and computes APSP concurrently on GPU.

Chapter 5 (*Linear algebra approach for parallel graph exploration on FPGAs: case study of APSP*), introduces a linear algebra approach based on sparse matrix multiplication for concurrent APSP computation on FPGA.

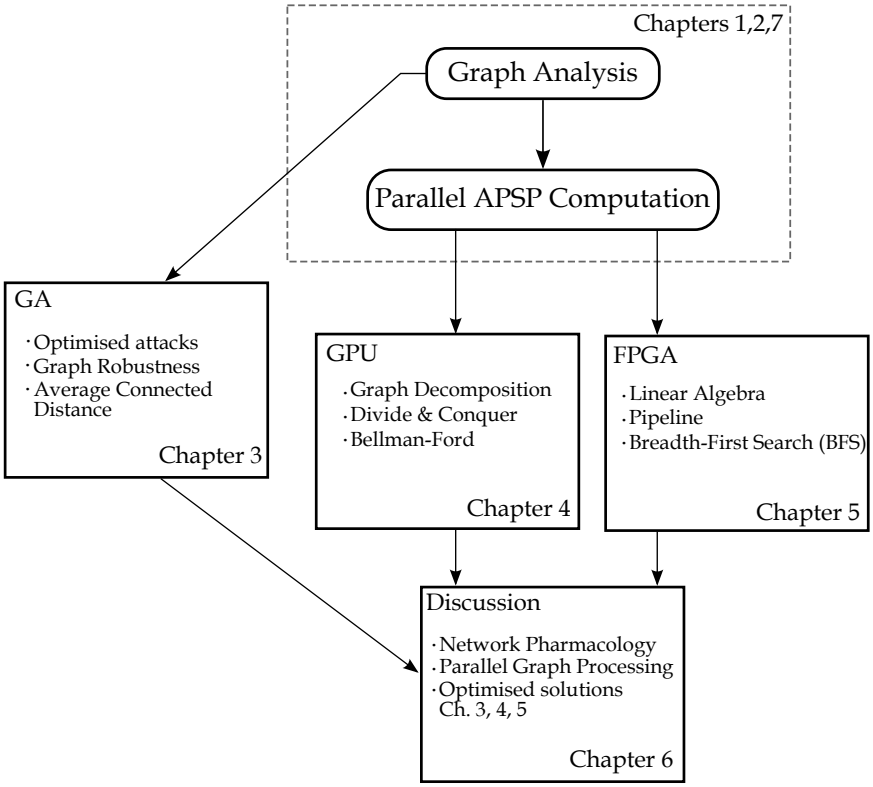



Figure 1.1: A visual description of the thesis structure.

Chapter 6 (*Discussion*) presents the nature of real-world graph problems in the domain of network pharmacology and parallel graph processing, design decisions and potential impact of our optimised algorithmic approaches.

Chapter 7 (*Conclusion*) summarises the contributions of this thesis and discusses directions for future research.

Appendix A (*Appendix*) includes information about algorithms' execution times.

# Chapter 2

# Background

## 2.1 Graph Analysis in Bioinformatics

Proteins are one of the basic constituents of life in our body. Since their functionality is very complicated and not yet completely understood, a graph approach can help to gain insights in this phenomenon [22]. Protein Protein Interaction (PPI) networks describe how proteins interact with each other. The vertices of a PPI graph represent different proteins while edges the physical interactions between them. Most real-world networks, as PPI networks, are not randomly structured but they are governed by some universal laws [23]. For more than 40 years, scientists believed that most vertices possess approximately a similar number of links [24]. However, in 1999 it was discovered that most nodes have very few links while few nodes, called hubs, have many connections [25]. Graphs governed by these properties are known as scale-free networks.

Given such networks, scientific community is interested to answer questions that are related with the importance of proteins, or graph vertices in general, within a network [26]. This can determine if a protein is essential for the survival of an organism or cell [27]. The identification of critical proteins for any disease can lead in the design of drugs that can affect positively the life of many people. This reveals the importance and necessity for tools that can efficiently analyse such graphs.

## Network Centrality

Questions related with the inner working properties of a graph can be answered with graph analysis tools that analyse the structure of a network by computing certain topological features as network centrality [28]. Network centrality can be considered as a function $CN()$ that assigns on each node $u$ a numerical value related with its importance within the analysed graph. Degree centrality $CN_{degree}$, one of the basic local-based network centrality measures, is defined as the number of the incident edges of a node (Fig. 2.1, a). It is a local-based measure as only the nearest neighbours are considered for the node characterisation. In case of a directed network, there are two types of degree centrality: the in-degree and the out-degree as there are in- and out-going edges. Nodes with higher degree centrality are considered to be more important than others as they are better linked with other vertices in the graph [29].

However, questions may be raised not only for the local-based graph properties as the degree centrality but also for the global-based properties. For instance, the detection of a node within a graph from which the transmission of a signal could reach all the other nodes in minimal time. Or, vice versa to detect the vertices that can be reached in the smallest amount of time from any other node within the graph. Such questions cannot be answered with local-based measures as the degree centrality but with global ones as closeness and betweenness centrality that are not restricted only on the nearest neighbours. Closeness centrality $CN_{closeness}$ measures how close and quickly a node can communicate with other nodes in the network [30]. It is the inverse of the average shortest distance from a node $u$ [31] (Fig. 2.1, c) and tends to be smaller for nodes that have larger average distance to other nodes in the graph. In contrast, betweenness centrality $CN_{betweeness}$ (Fig. 2.1, d) is defined as the number of shortest paths between all pairs of vertices that pass through a vertex $u$ [32]. Henceforth, the term 'shortest path' $d(u, v)$ will refer to the minimum length between two nodes $(u, v)$ (Fig. 2.1, b). A formal definition of the shortest path $d(u, v)$ will be given in Section 2.1.1.

Furthermore, scientists are interested for graph analysis tools that can measure the differentiation of graph robustness when single or multiple nodes are removed from a graph due to failures or attacks. The resilience of a graph in

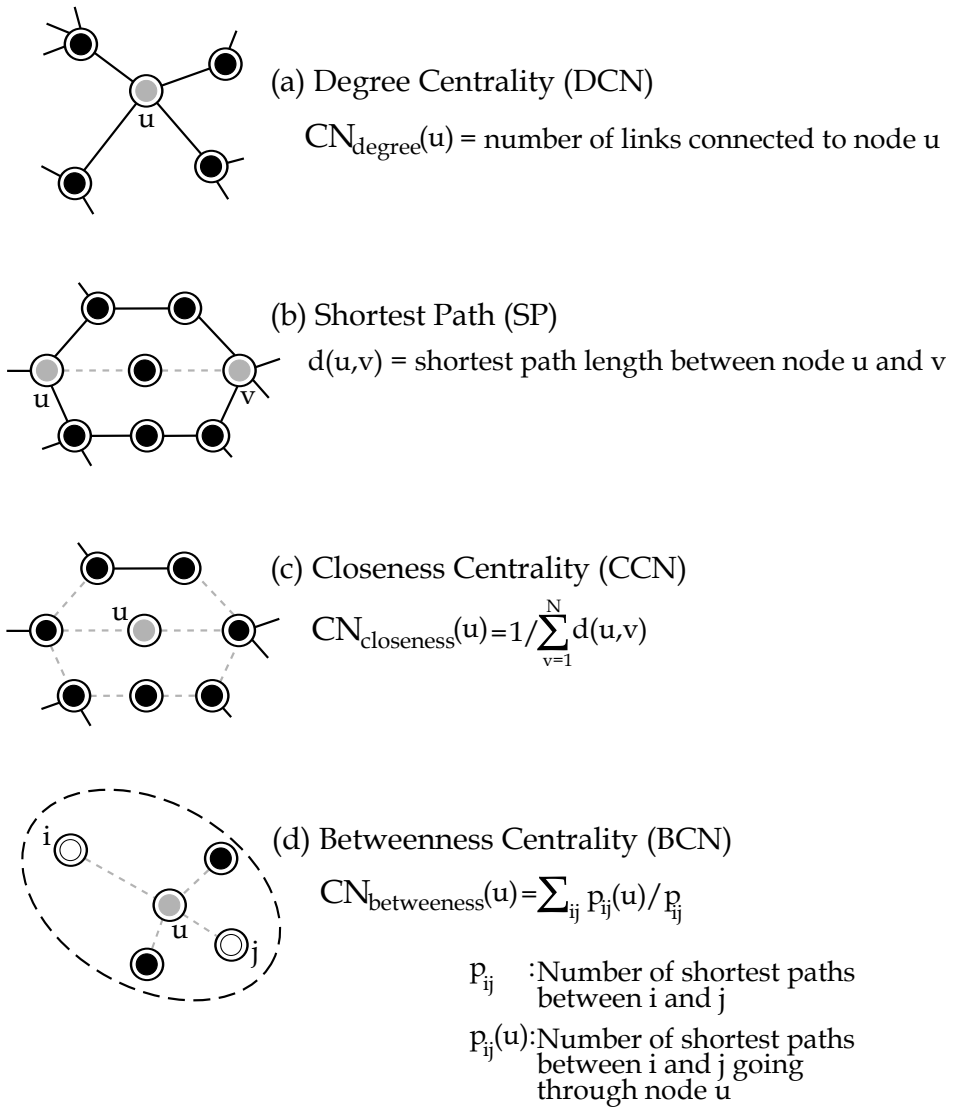



Figure 2.1: Topological features formulas: (a) Degree Centrality, (b) Shortest path, (c) Closeness centrality and (d) Betweenness Centrality.

node attacks or failures is defined as graph robustness. A higher impact in the overall graph robustness can be achieved if we remove node combinations that are more significant for the structure of the network. The ability to compute such high impact node combinations can help researchers in scientific domains as network pharmacology to produce better targeted drugs for complex diseases as cancer and diabetes [33]. Network pharmacology, based on graph analysis, seeks the most critical proteins for any disease and produces multi-target drugs of better efficiency and lower toxicity. Common graph analysis tools as Cytoscape [34], [35] and Nexcade [16] are used to simulate targeted attacks, on individual vertices or groups of nodes, based on graph measures as network centrality. However, tools that compute high impact combinations are absent from the research community. Such high-impact combinations are unlikely to be found unless they are specifically searched for [36].

In this context, we propose an optimised implementation based on a genetic algorithm that computes high impact node combinations where their removal is more effective comparable to the strategies of random and targeted impact analysis. We provide further details related with this novel genetic approach in Chapter 3. Apart from the challenging computation of high impact combinations, the measurement of graph robustness in the context of a genetic algorithm can be very challenging and can lead in performance deterioration due to high computational workloads. Our GA uses the metric of graph robustness in order to assess the effectiveness of its population of solutions. However, as the evaluation of graph's robustness is based on computationally expensive algorithms this may affect GA's overall performance.

## Measuring Graph Robustness

The robustness of a graph is strongly related with the scale-free property [37] as it turns out that targeted attacks on hubs can cause structural failure on the connectivity and topology of a graph. The 'robust, yet fragile' nature of graphs can be used for the development of applications that can either predict or provoke terrorist attacks or failures in the power grid [38]. The removal of highly connected proteins in PPI networks achieves much better perturbations

rather than removing proteins with small number of links [39]. Graph metrics as closeness and betweenness centrality are used for targeted attacks on nodes and can achieve more effective results than the degree centrality [28]. The cascading disturbances created in the global graph topology and connectivity, as a result of the loss of a node or group of nodes, are quantitatively measured with topological features as the average connected distance.

The average connected distance $\bar{d}$ can be computed by multiplying the fraction $\frac{2}{n(n-1)}$ with the sum of the lengths of the shortest paths [40]:

$$\bar{d} = \frac{2}{n(n-1)} \sum_{u=1}^{n} \sum_{v=u+1}^{n} d(u,v) \tag{2.1}$$

where $n$ equals with the number of nodes in the graph. If $\bar{d} = 1$, then all the nodes are directly linked to each other. Consequently, if the average distance is getting smaller this means that also the graph is getting robuster as paths are getting shorter. However, a prerequisite for the calculation of $\bar{d}$ is the computation of the shortest path costs between all pairs of nodes. The cost paths are defined as the sum of edge-weights composing a path while their calculation is based on algorithms that belong in the category of the all pairs shortest path (APSP) computation.

Such algorithms if computed repetitively can lead in high execution time and performance deterioration as their computational complexity is getting higher due to repetition. While nowadays the emerging parallel architectures provide enhanced processing resources, the sparse nature and irregular structure of the analysed graphs makes the parallelisation of the APSP problem very challenging. Therefore, we design two novel parallel APSP approaches presented in Chapters 4 and 5 where we target in the acceleration of the APSP calculation over graphs with real-world properties. Before providing any information related with our APSP approaches, we firstly provide the needed background information for the parallel APSP computation. Furthermore, we present the mostly used sequential APSP algorithms as the Bellman-Ford and Breadth-first search algorithm. We provide the definition of parallel computing, the common strategies for parallel algorithm design and the landscape of how scientific community moved from single-core processors to parallel hardware architectures. Moreover, we present

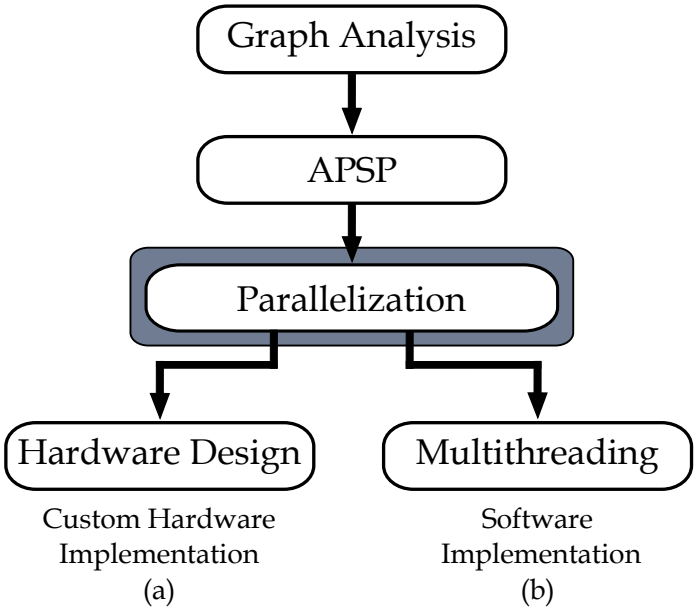



Figure 2.2: Parallelisation of the APSP problem: Use of customised hardware (FPGA) (a) and multi-threaded platform (GPU) (b).

the state-of-art for the parallelization of the APSP problem (Fig. 2.2) in emerging on-desktop hardware architectures as GPU and FPGA. Also, we provide further details about the architecture and the programming model of both GPU and FPGA platforms (Fig. 2.2, a, b). Finally, we present the common graph representation used in parallel architectures and conclude by providing the speedup definition.

### 2.1.1 All-pairs Shortest Path (APSP) Computation

The all-pairs shortest path (APSP) computation is one of the most fundamental problems in computer science. It is defined as the computation of all the minimum length paths between all pairs of vertices. The APSP computation consists a significant component of many practical applications in a variety of fields as bioinformatics [41], data mining [42] and computer aided design [43]. As we discussed in the previous Section, the calculation of average connected distance is based on the APSP computation. However, the APSP problem has a cubic complexity and consists the most expensive component of many graph measures

[44].

In this Section, we provide the definition of the APSP problem and present common sequential APSP algorithms. We provide further details about the Bellman-Ford and Breadth First Search algorithm as they will be used as basic components for our parallel implementations in Chapters 4 and 5.

## Notation and Definition

Let $G$ be a graph defined as $G = (V, E)$ where $V$ is a set of vertices and $E$ a set of edges. If $u$ and $v$ are vertices of $G$ then the tuple $(u, v)$ that connects the two nodes is called edge. Graphs can be either undirected or directed. In a directed graph all edges can be traversed in one direction. On the contrary, an edge $(u, v)$ of an undirected graph can be traversed from both directions. Also, in weighted graphs, each edge is related with a scalar value that represents the strength of the inter-connection between two nodes.

The length or distance $d(P)$ of a path $P =< s, \ldots, u, \ldots, v, \ldots, t >$ from a source node $s \in V$ to a node $t \in V$ is defined as the sum of edge-weights for edges in $P$. The smallest value of $d(P)$ for all paths P from $s$ to $t$, known as shortest path, is denoted as $d(s, t)$. Let $u, v, z$ be three vertices in graph $G$ with non-negative edge weights where the weight function is $w : E \longrightarrow R$. Then the distance function $d : V \times V \longrightarrow R$ is given by [45]:

$$d(s,t) = \begin{cases} \infty, & \textit{if there are no paths from } s \textit{ to } t \\ min\{w(W) \mid W \textit{ is a } s\textit{-}t \textit{ walk }\}, & \textit{otherwise.} \end{cases}$$

where $d(s, t)$ is the shortest path from a source node $s$ to a targeted node $t$.

## Sequential APSP Algorithms

The computation of shortest paths from a source node to all other nodes is known as the Single Source Shortest Path (SSSP) problem while the computation of all shortest paths between all pairs of nodes as All-Pairs Shortest Path (APSP) problem. While the computation of SSSP can be efficiently done in linear time,

the APSP problem is more computationally expensive.

In 1962, Floyd Warshal proposed an algorithm that solves the APSP problem in $O(V^3)$ where $V$ is the number of nodes in the graph [46]. Also, Donald Johnson presented an APSP algorithm in 1977 that its time complexity is $O(|V|^2 \log |V| + |V||E|)$ [47]. However, a solution for the APSP problem can derive from the iterative computation of the SSSP for each source vertex. Dijkstra's algorithm is an ideal SSSP solution for sparse graphs. An accelerated version of Dijkstra's algorithm, with the use of a Fibonnacci heap, can provide APSP in $O(|V||E| + |V|^2 \log|V|)$, where $E$ represents the number of edges in graph [48]. Furthermore, a graph exploration algorithm as Breadth-first search (BFS) can be used for the APSP computation in $O(|V|(|E| + |V|))$ [49]. Finally, an SSSP algorithm as Bellman-Ford solves the APSP problem for graphs with negative weighted edges in $O(|V|(|V||E|))$ [49].

Further details about the sequential Bellman-Ford and Breadth-first Search algorithm are given in the next Sections 2.1.2 and 2.1.3. We focus on these two graph algorithms as their structure favours more parallelism to be exposed and essentially more vertices to be processed in parallel. We give more details about the current state-of-the-art parallel APSP algorithms in Section 2.2.5. Being aware of the basic mechanics of these sequential APSP algorithms will help to better understand the GPU and FPGA based parallel graph implementations in Chapters 4 and 5.

## Graph Representation

The graph data, used as input by graph algorithms, can be represented with various ways. Suppose that $u_1, \ldots, u_n$ are the vertices of a directed or undirected graph $G$. The most common graph representation uses a $|V| \times |V|$ adjacency matrix $A = (a_{ij})$ such that [50]:

$$a_{ij} = \begin{cases} 1, & \text{if}(u_i u_j) \in E \\ 0, & otherwise. \end{cases}$$

If $G$ is a weighted graph then the weight $w(u, v)$ of each edge $(u, v) \in E$ is stored as an entry in the adjacency matrix $A$ (Fig. 2.3, b). In contrast, the non-existence

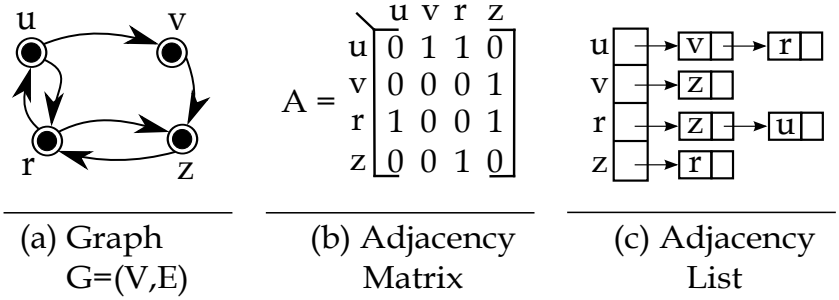



Figure 2.3: (a) Directed graph $G = (V, E)$ represented as (b) adjacency matrix A and (c) adjacency list.

of an edge between two nodes is represented with 0 or $\infty$.

The adjacency matrix representation is ideal for dense graphs. However, the level of sparsity in real-world graphs tends to be high [51]. Consequently, the representation of a sparse graph with an adjacency matrix is inefficient as allocated memory is not fully exploited. In contrast, the adjacency list presentation can minimize graph's memory footprint by storing only the non-zero values. Let $adj[]$ be an array with $|V|$ lists corresponding on each node in $V$ (Fig. 2.3, c). Then each adjacency list $adj[u]$ stores only the nodes that are connected with $u$ in $G$. While the adjacency list representation is commonly used for sequential graph algorithms, although is not preferred for parallel graph algorithms for reasons that will be further explained in Section 2.3.3.

### 2.1.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm computes the SSSP problem in a directed or undirected graph where edges are weighted and can be negative values. It belongs in the algorithmic category of relaxation algorithms where relaxation is the process of assigning a cheaper distance to a vertex $v$ by using the shortest path $s, u$ and the edge $(u, v)$ (Fig. 2.4). If the distance $d(v)$ of node $v$ is larger than the sum of distance $d(u)$ of node $u$ plus the weight of edge $u, v$ then this sum is assigned on the distance $d(v)$ of node $v$ (Algorithm 1). Furthermore, Bellman-Ford algorithm is able to detect negative-weight cycles. This is a very crucial feature as its absence can lead in non-meaningful shortest path results. For brevity, Bellman-ford algorithm depicted in Algorithm 1 does not contain the function of negative cycle

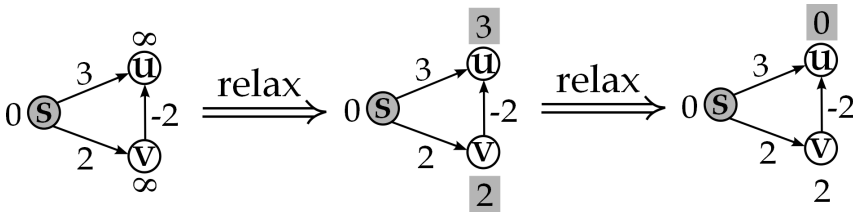



Figure 2.4: Relax edges by assigning correct distances.

detection.

Bellman-Ford algorithm maintains, for each node $u$, two arrays where stores its distance $D[v]$ and predecessor $P[v]$. Initially, the distance array of the source node $D[s]$ is initialised with zero and the distance of the node $v$ with infinity ($D[v] = \infty$ for every $v \neq s$) while $P[v]$ is undefined. The algorithm performs a sequence of relaxation steps. In each iteration the arrays $D[v]$ and $P[v]$ are updated based on the relaxation technique. After $n - 1$ relaxations, $D[v]$ will contain all the correct distances. Once this happens, each $P[v]$ points to the predecessor of $v$ on a valid shortest path from $s$ to $v$.

**Algorithm 1** A general template of Bellman-Ford algorithm [52]

**Input:** An undirected or directed graph $G = (V, E)$ that is weighted and has no self-loops. A vertex $s \in V$ from which to start the search.
**Output:** A list $D$ of distances such that $D[u]$ is the distance of a shortest path from $s$ to $u$. A list $P$ of vertex parents such that $P[u]$ is the parent of $u$.

1: $D \leftarrow [\infty, \infty, \ldots, \infty]$ ▷ n copies of $\infty$
2: $D[s] \leftarrow 0$
3: $P \leftarrow []$
4: **for** $i \leftarrow 1, 2, \cdots, n-1$ **do**
5:  **for** each edge $uv \in E$ **do**
6:  **if** $D[v] > D[u] + w(uv)$ **then**
7:  $D[v] \leftarrow D[u] + w(uv)$
8:  $P[v] \leftarrow u$
9: **return** $(D, P)$

### 2.1.3 Breadth-first Search (BFS)

Given a graph $G$, breadth-first search algorithm traverses the edges of a directed or undirected graph $G$ in order to discover every node that is reachable from a source node $s$. BFS explores all the nodes in level $k$ before moving to nodes in

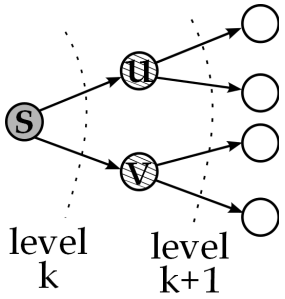



Figure 2.5: Graph exploration with Breadth-first search (BFS).

level $k+1$ (Fig. 2.5). The search maintains a frontier on each level $k$ that it stores all the unvisited nodes. The breadth-first search template depicted in Algorithm 2 uses a queue structure $Q$ to store the frontier. The outcome of a BFS search is a tree $T$ with root $s$ that contains all vertices reachable from $s$.

In general, BFS is used for graph exploration and not for shortest path computation. However, BFS can be used to compute the SSSP while discovering new nodes. When nodes are discovered on a certain BFS level then this level can be used to compute the distance of a node from a source node and store its distance in array $D[]$. If this strategy is applied for all the nodes of a graph $G$ then we can compute the APSP. This modified version of BFS that computes APSP is called as all-pairs breadth-first-search (AP-BFS) [53].

**Algorithm 2** A general breadth-first search template [52]

**Input:** An undirected or directed graph $G = (V, E)$. A vertex $s$ from which to start the search.
**Output:** A list $D$ of distances of all vertices from $s$. A tree $T$ rooted at $s$.

```
1:  Q ← [s]                                   ▷ queue of nodes to visit
2:  D ← [∞, ∞, ..., ∞]                        ▷ n copies of ∞
3:  D[s] ← 0
4:  T ← []
5:  while length(Q) > 0 do
6:      u ← dequeue(Q)
7:      for each w ∈ adj[u] do
8:          if D[w] = ∞ then
9:              D[w] ← D[u] + 1
10:             enqueue(Q, w)
11:             append(T, uw)
12: return (D, T)
```

## 2.2 Parallel Graph Processing

Graph problems as APSP computation consist the most expensive part for graph analysis tools as outlined in Section 2.1. As the scale of graph problems is growing larger this has as result a greater demand for more computational power. Conventional Central Processing Units (CPUs) cannot cope with this data deluge. Therefore, efficient parallel graph processing is becoming important as computational and memory requirements are getting increased. Parallel computing has proven to be an ideal solution when requiring more processing power. The evolution of hardware architectures towards a larger amount of processing units made feasible the concept of 'parallelism' which is the main component for the synthesis of parallel algorithms.

The term 'Parallel graph processing' implies the use of high performance computing and parallel algorithmic techniques in order to accelerate computationally expensive graph problems as APSP. Graph problems have some inherent properties that pose many challenges in their efficient parallel processing. In this Section, we provide a brief description on how hardware evolved from single-core processors to multi-core and parallel architectures. Furthermore, we provide the definition of parallel computation and present common strategies for parallel algorithm design. We analyse the challenges of parallel graph processing and examine techniques used in literature that attempted to accelerate APSP computation in GPU and FPGA platforms.

### 2.2.1 High Performance Computing Landscape

#### From Single to Multi-core Architectures

In 1945, mathematician John von Neuman, suggested the stored-program model of computation [54]. A von Neumann machine executes a single instruction at a time and each instruction operates on few pieces of data [55]. Its performance is related with the time spent to execute a given task and expressed as the multiplication of Instructions executed Per clock (IPC) with processor's frequency [56]. In an effort to enhance CPU's performance, computer architects increased the amount of instructions executed on a single clock cycle through instruction-level

parallelization (ILP) techniques. However, as software evolved, applications become more capable to run multiple tasks simultaneously with the use of threading. A thread is a sequence of instructions that is executed independently from other sequences. The need for parallel execution and the limited physical resources on a single core processor led in the use of simultaneous multi-threading (SMT). A physical processor appears as multiple logical processors and applications can schedule multiple threads on them. However, single-core processors are not able to achieve simultaneous execution of instructions streams but only to interleave them.

In 1965, Gordon Moore stated that chip's number of transistors will double roughly every two years [57]. For the last decades, designers were based on Moore's Law and enhanced microprocessor's performance by creating complex architectures that can operate on higher frequencies. However, this led on power hungry designs that were totally unsustainable and even higher performance was needed [58]. This performance restriction was removed with the development of multi-core architectures that fitted more execution cores on a single die. Threads are not restricted by physical resources as they can independently run on separate cores. While slower frequencies are used in multi-core architectures, better performance and slower growth in power consumption is achieved [59].

### Many-core Architectures

A different architectural design path was the use of many-core architectures where their cores exceed the number of multi-core architectures. A current example is the Graphical Processing Units or GPUs that have been used in computer graphics [60]. GPUs have almost a double number of cores in comparison to many-core CPUs. The performance gap between GPUs and CPUs is related with the different design strategies that are build on. CPU is a latency-oriented processor where it uses complex control logic that allows parallel instruction execution of a single thread while maintaining the appearance of sequential execution. In contrast, GPUs are based on a throughput-oriented architecture. Their hardware takes advantage of the large number of threads, to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations

[61]. This throughput-oriented design saves chip area and allows designers to have more arithmetic units on chip and thus increase the total execution throughput in GPUs (Fig. 2.6). In this thesis, as we utilised NVIDIA's GPU platform [62], we provide further details about its specific architecture and programming model in Section 2.3.1.

### Dataflow Architectures

Conventional processors as CPUs and GPUs, require large number of data transfers between processor and memory in order execute a single instruction. This performance bottleneck can be resolved with the transformation of a complex kernel to a dataflow graph where each node is mapped on a separate functional unit. The concept of dataflow computing was introduced in the 1970s, nevertheless the absence of an architecture with unlimited number of functional units without any memory constraints made it technically infeasible. However, modern Field-programmable gate arrays (FPGAs) helped to overcome this overhead. Processing is being done by forwarding intermediate results from node to node [63]. Consequently, the cyclic memory access phenomenon is eliminated as dataflow computation is inherently local [64]. In order to express the dataflow programmability in FPGAs we used a framework provided by Maxeler [65] and we present further details in Section 2.3.2.

### Parallel Computing Classification

While CPUs cannot keep following Moore's law, FPGAs are able to continue on this track as each new FPGA generation provides more logic and memory resources. Their real strength lies in their capability to deliver application specific designs. In contrast, GPU is not a customised accelerator but an architecture that provides high floating point performance [66].

In general, all parallel architectures can be classified based on Flyn's taxonomy [67]. It categorises them based on their ability to process streams of data and instructions. On the one hand, processors as GPU belong in the single-program-multiple-data (SPMD) category while Multi/Many-core CPUs in both SPMD and multiple-instruction-multiple-data (MIMD) category (Fig. 2.6). On the other

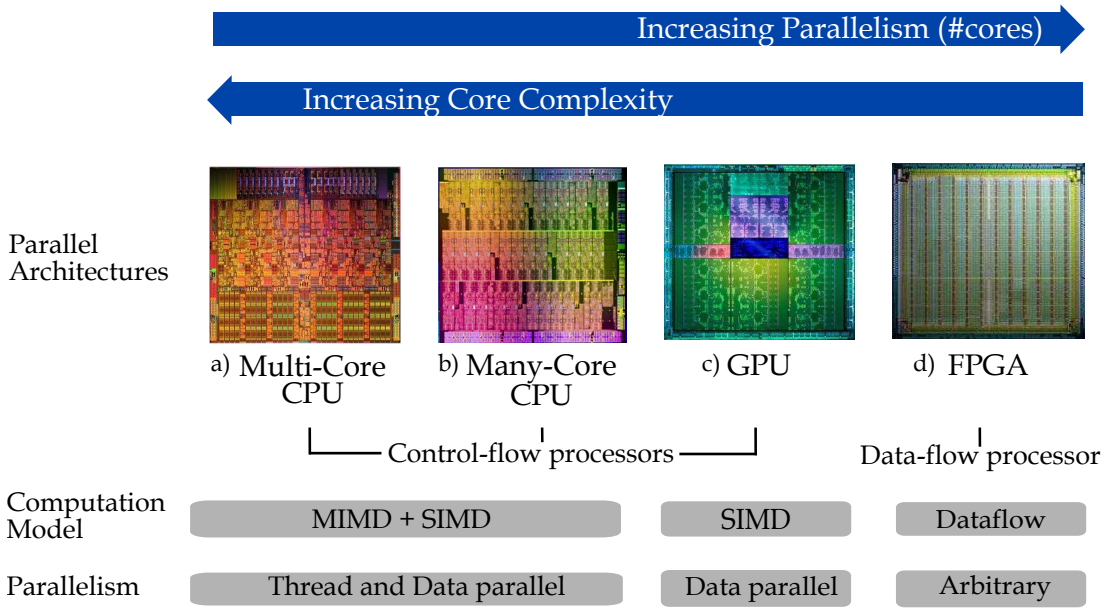



Figure 2.6: Spectrum of current parallel computing architectures.

hand, reconfigurable architectures as FPGAs are flexible to implement anything from a complex circuit to a CPU core. Consequently, this versatility doesn't allow them to be categorised though Flyns's taxonomy as they can implement all possible levels of parallelism.

Both Many- and Multi-core architectures belong in the category of control flow processors. Such processors are equipped with multiple general processing cores that work for everything and we can say that operate in a similar way as workers in a factory. The difference between Multi- and Many-core architectures lies in the fact that the first architecture uses a small number of highly sophisticated cores while the later uses a greater number of less complicated cores that work on slower frequencies. In contrast, a dataflow architecture as FPGA processes data on a dataflow mode that resembles to factory workers that are arranged in an assembly line and each of them is responsible for only one operation on each product.

### 2.2.2 Parallel Computing

Parallel hardware architectures as GPU and FPGA provide the needed technology for parallel computation. However, in order to harness such architectures we need a strategy that maps a problem on these parallel execution resources. Parallel computing is the action of decomposing the domain of a problem with size $n$ in smaller parts that are simultaneously solved in $p$ processing units. Let $N_Z$ be a problem with domain $Z$. If $N_Z$ can be parallelized, then $Z$ can be partitioned into $m$ sub-problems with $m \geq 2, m \in \mathbb{N}$ [68]:

$$Z = z_1 + z_2 + \dots z_m = \sum_{i=1}^{m} z_i \tag{2.2}$$

Parallelism is achieved by either decomposing data or tasks in smaller modules and assigning them on different processing units. If $N_Z$ is a data-parallel problem then $Z$ is composed of data elements. The problem is solved by applying on the whole domain $Z$ the same kernel function $K(\dots)$:

$$K(Z) = K(z_1) + K(z_2) + \dots K(z_m) = \sum_{i=1}^{m} K(z_i) \tag{2.3}$$

On the contrary, if $N_Z$ is a task-parallel problem then each task $z_i$ is applied on a common data stream $D$:

$$Z(D) = z_1(D) + z_2(D) + \dots z_m(D) = \sum_{i=1}^{m} z_i(D) \tag{2.4}$$

Parallel computation is expressed through the design of parallel algorithms that regulate how a problem will be efficiently partitioned and executed on a parallel architecture. In general, parallel algorithm design is not a simple process but it can be considered as an art where the designer needs to be equipped with creativity, discipline and abilities that cannot be easily classified [69]. The introduction of parallelism, as a concept, creates new degrees of freedom that are not existed

in sequential algorithmic design and increases the complexity of developing new parallel algorithms. While designing a new parallel algorithm we need to consider both algorithmic requirements and parallel machine capabilities in order to achieve maximum performance.

### 2.2.3 Designing Parallel Algorithms

None golden rule can lead in the development of the ideal parallel algorithm [68]. In 1989, Cole proposed some high level strategies known as parallel algorithmic patterns that provide an abstraction for the development of parallel algorithmic designs [70]. The parallelism patterns allow the designer to target on the algorithmic strategy that will help him to implement a parallel algorithm without having to think details related with the used parallel programming model. However, it is very critical to identify and use the right pattern on a different problem domain. Some of the most significant algorithmic skeletons in parallel computing are the following [71], [72]:

- **Task-level Parallelism Pattern:** A problem is broken down to a group of tasks (Eq. 2.4) that are executed independently. These category of problems are known as embarrassingly parallel problems as there no dependencies between them.
- **Data parallelism Pattern:** Data are decomposed on smaller chunks and processed in parallel by the same algorithmic function as outlined in Equation 2.3.
- **Divide and Conquer pattern:** A problem is divided on subproblems that are solved concurrently. Their solutions are combined together to provide the final solution (Fig. 2.7).
- **Pipeline Pattern:** The computation is divided in a sequence of stages where data flow within these stages like an assembly line. Each thread is responsible to process each stage on the same time.

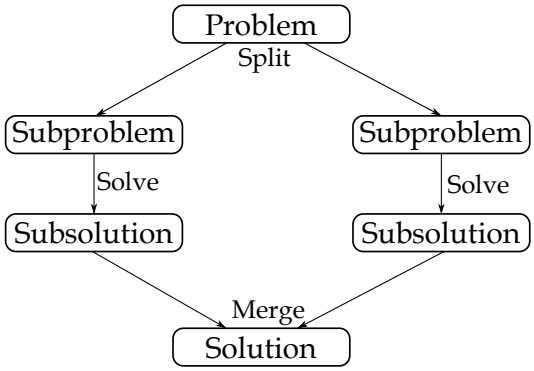



Figure 2.7: The divide and conquer strategy.

### 2.2.4 Parallel Graph Processing Challenges

Parallel computing has commonly used to accelerate both compute and data intensive applications [73]. It is highly optimised for the acceleration of regular numerical calculations in fields spanning from computational physics [74] till biomedical applications [75]. However, mapping combinatorial problems as graph algorithms on parallel architectures poses many challenges [76]. Graphs are ruled by some inherent properties that don't allow them to adapt current computational problem-solving strategies. Above all, the following characteristics of graph problems pose the most considerable challenges in parallel graph processing [20]:

- **Data-driven computations**: In general, graph computations are driven from the structure of the graph that is related on how edges and nodes are connected. Therefore, parallelism cannot be easily expressed by partitioning computation, as the structure of the computations is not known a priori.

- **Unstructured problems**: Data used in graph problems is unstructured and highly irregular. Their irregular nature makes it difficult to partition them and extract the needed parallelism. This can guide on low scalability due to unbalanced computational loads caused from the not so well partitioned data.

- **Poor locality**: Both the irregular and unstructured nature of graph data lead in low memory access locality. Consequently, graph algorithms cannot

achieve high performance on conventional cache-based processors that rely on spatial and temporal memory access locality.

- **High data access to computation ratio**: Graph algorithms tend to explore the structure of the graph without performing large amount of computations. Hence, there is a higher ratio of data access comparatively to other scientific applications. Moreover, these memory accesses characterised by poor locality guide on execution times that are dominated by memory latency.

### 2.2.5 Parallel APSP Algorithms

A parallel graph algorithm can derive either by transforming a sequential algorithm to a parallel version or developing an entirely new algorithm that fits on the targeted parallel architecture [77]. It can also derive from an existing parallel algorithm. In this thesis, we examine the graph problem of all pairs shortest path (APSP) computation, in particular the problem of costs path computation between all pairs of nodes. We seek efficient parallel graph algorithmic techniques implemented on emerging hardware architectures as GPU and FPGA that can accelerate the APSP computation.

Most of the real-word graphs as PPIs tend to be sparse and may have negative weights that indicate antagonism between proteins [78]. Consequently, an efficient parallel APSP algorithm needs to take into consideration all these embedded properties. As we discussed earlier in Section 2.1.1, the APSP computation can derive from two main families of sequential algorithms. One of them targets in SSSP computation as Dijkstra ($O(|E|+|V|\log|V|)$) and Bellman-Ford ($O(|V||E|)$) approach where if they repeated for all nodes of $G$ then they can provide the APSP. In contrast, the other family derives from the Floyd-Warshall approach where its computational complexity is $O(|V|^3)$ [79] regardless the density of the input graph (Table 2.1). The Floyd-Warshall algorithm can process negative edge weights, however, is inefficient for graphs of greater size due to its cubic complexity. In contrast, the APSP can be computed by Dijkstra's algorithm in $O(|V| \times |E| + |V|^2 \log|V|)$ and become $O(|V|^3)$ if the graph is complete. If $|E| = O(|V|)$ then the complexity goes down to $O(|V|^2 \log|V|)$, so Dijkstra's

algorithm is faster from Floyd-Warshall for sparse graphs.

Although, Dijkstra's algorithm is efficient for sparse graphs, however, is inefficient for parallelization due to its inherent sequential properties. Parallel architectures as GPUs and FPGAs need parallel graph algorithms that can exploit their massive parallel processing resources. In order to expose more parallelism, we consider algorithms that can process more vertices simultaneously. The Bellman-Ford algorithm is such a method as it repeatedly processes graph's all interconnections and updates its vertices continuously. Even if Bellman-Ford algorithm is a classically sequential algorithm, is better suited for parallel execution than Dijkstra's algorithm. Additionally, Bellman-Ford algorithm may be slower ($O(|V||E|)$) than Dijkstra's algorithm however is more versatile as it can handle graphs with negative weights (Table 2.1).

While designing parallel graph algorithms, we need to maintain a balance between parallelism and efficiency. While Dijkstra's algorithm doesn't expose any parallelism, in contrast, Bellman-Ford's algorithm that expresses more parallelism is more expensive. This example reveals the need of delivering good parallel efficiency while adding the smallest amount of additional work. Many researchers tried to accelerate the APSP problem with various accelerators as GPUs [80], FPGAs [81] and clusters of processors [82]. In the next subsections, we examine the parallel APSP computation on parallel systems as multi-threaded GPU platforms and customised FPGA accelerators.

## APSP computation in GPUs

In 2004, Micikevicius et al. introduced one of the earliest implementations of APSP computation in NVIDIA's GPUs [83]. They proposed a parallel implementation of Floyd-Warshall algorithm, tested on graph with 2048 vertices and achieved a 3x acceleration comparable to a CPU implementation (Table 2.2). In 2007, Harish and Narayan proposed a parallel APSP approach based on Dikjstra's algorithm, implemented on a single GPU and tested with graphs ranging till 5k vertices [84]. Also, they proposed a Floyd-Warshall implementation based on the CREW PRAM parallelization model [85] and their tests were restricted on

Table 2.1: Sequential APSP algorithms: Comparison of their characteristics and computational complexities.

| Algorithm | Complexity | Characteristics |
|---|---|---|
| Dijkstra | $O(\|V\| \times \|E\| + \|V\|^2 \log \|V\|)$<br>$O(\|V\|^3)$, if graph complete<br>$O(\|V\|^2 \log \|V\|)$, if $\|E\| = O(\|V\|)$ | Efficient for sparse graphs<br>Inefficient for parallelisation |
| Bell-Man Ford | $O(\|V\|\|E\|)$ | Supports negative edges<br>Efficient for sparse graphs<br>Efficient for parallelisation |
| Floyd-Warshall | $O(\|V\|^3)$ | Supports negative Edges<br>Inefficient for sparse graphs |
| AP-BFS | $O(\|V\|(\|E\| + \|V\|))$<br>$O(\|V\|^2)$, if $E = O(\|V\|)$ | Efficient for sparse graphs<br>Supports unweighted graphs |

graphs with only 5k vertices due to the use of adjacency matrix representation. Buluc et al. proposed an APSP approach based on the blocked-recursive Floyd-Warshall algorithm implemented on a single GPU [86]. Their implementation required the whole graph to be stored on GPU's memory and for that reason they reported execution times for graphs with only 8k vertices. Okuyama et al. proposed an improvement in Harish's and Narayan's APSP approach by using a task parallel scheme and reducing data transfers through on-chip memory exploitation [87]. Matsumoto et al. proposed a blocked Floyd-Warshall approach on a hybrid CPU-GPU system that handled graphs with 8k vertices [88].

However, the APSP calculation can also derive from the repetitive computation of the SSSP problem. Davidson et al. proposed a series of work-saving methods that let them achieve better speedups on Bellman-ford and Dijkstra's algorithm [89]. Also, a specialised library for irregular applications as graphs called LonestarGPU [90], consists the state-of-art for parallel graph processing and acceleration of algorithms as Bellman-Ford. It provides vertex parallelisa-

tion where each thread is responsible for one vertex and updates the distances of adjacent vertices. Race conditions are avoided with the use of atomics. Furthernore, Djidjev et al. proposed in 2014 a divide and conquer approach for APSP computation by using multi-node GPU clusters [91] and divided their graph via the Parmetis [92] platform on smaller components. They firstly computed shortest paths inside the partitioned subgraphs and then calculated paths between components. However, they divided their graph on several components without taking into consideration the embedded properties of complex networks.

Taking parallel algorithm design decisions based on the nature of the graph is an emerging area of research. Only Banerjee et al. proposed in 2015 a GPU implementation based on a graph pruning technique and used bi-connected components (BCC) in order to compute APSP [93] (Table 2.2). However, the success of such approach is based on finding good partitions and mainly experimented with planar graphs to ensure good partitions [91]. Additionally, the APSP approach in [93] is restricted only in the use of BCC components that are not revealing in detail the embedded graph properties of complex networks. Therefore we propose a divide and conquer approach that partitions graph in Strongly Connected Components (SCCs) and shortest paths are concurrently computed over these components. Further details about the implementation based on SCCs can be found in Chapter 4.

### BFS exploration in FPGAs

The APSP computation can also derive from graph exploration algorithms as BFS where its computational complexity is only $O(|V|+|E|)$. The modified version of BFS that computes APSP is known as the all-pairs breadth-first-search (AP-BFS) [53] and its complexity is $O(|V|(|E| + |V|))$. For sparse graphs with $E = O(|V|)$, the complexity of AP-BFS is $O(|V|^2)$ (Table 2.1). Many attempts in the literature aimed to accelerate graph exploration through innovative programming, including usage of multi-core systems, Graphic processing units (GPUs) and Field programmable gate Arrays (FPGAs). For multicore processors, Agarwal et al. proposed a BFS implementation that optimize cache utilization and intersocket

Table 2.2: Previous GPU-based parallel APSP approaches.

| Algorithm | GPU Parallel APSP |
|---|---|
| Dijkstra | Harish and Narayan [84], 5K vertices - 2x acceleration<br>Davidson et al. [89], 1M vertices - 1.3x acceleration |
| Bell-Man Ford | Davidson et al. [89], 1M vertices - 2x acceleration |
| Floyd-Warshall | Micikevicius et al. [83], 2048 vertices - 3x acceleration<br>Harish and Narayan [85], 5K - 1.5x acceleration<br>Buluc et al. [86], 8K vertices - 0.7x acceleration<br>Okuyama et al. [87], 32K vertices - 2x acceleration<br>Matsumoto et al. [88], 8K vertices - 0.8x acceleration<br>Djidjev et al. [91], 100K vertices - 7x acceleration<br>Banerjee et al. [93], 400K vertices - 2x acceleration |

communication [94]. Hong et al. designed a hybrid method of level-synchronous BFS that dynamically chooses the best execution method for each BFS-level that can be either multi-core CPU or GPU execution [95]. Luo et al. proposed a GPU targeted approach where each thread is mapped to each frontier vertex of the current BFS level [96]. Instead of an architecture specific implementation Zhong et al. developed a GPU programming framework for graph processing [97].

In the area of FPGAs, previous work mostly focused on memory access optimization [98]. In 2010, Wang et al. proposed a multi-softcore architecture on FPGA for BFS and exploited the on-chip memory to store graphs [99]. In 2012, Betkaoui et al. provided a scalable BFS implementation [81] for a specific FPGA architecture known as Convey HC-1 platform [100] (Table 2.3). Such platforms are known for their high memory bandwidth and their exploitation let them to achieve 2x improvement comparable to the state-of-art GPU and CPU implementations. An optimization on Betkaoui's work was presented in 2014 by Attia et al. [101] where they provided higher memory utilization based on a Convey HC-1 platform and achieved 5x relative speedup. Buluc et al. used a parallel BFS approach based on linear algebra to develop a scalable implementation for distributing memory systems [102]. In this context, we propose an efficient BFS

approach implemented based on linear algebra, independent of any data dependencies and able to provide acceleration in common FPGA architectures. Further details of the new implementation are provided in Chapter 5.

Table 2.3: Previous FPGA-based parallel APSP approaches based on BFS.

| Algorithm | FPGA Parallel APSP |
|---|---|
| AP-BFS | Betkaoui et al. [81], 40K vertices - 2.7x acceleration<br>Attia et al. [101], 1M vertices - 1.3x acceleration |

## 2.3 Parallel Hardware Architectures

The transformation of a parallel algorithm to an efficient program that entirely exploits the characteristics of the targeted parallel architecture is known as parallel programming [103]. The parallel programmer needs to choose the right parallel architecture, understand their parallel programming model and express the needed parallelism. Parallel programming with GPUs and FPGAs differs a lot as in the one we need to operate within a multi-threading environment while in the other we have to express a behavioural description that leads in a customised hardware design. Further details about GPU's and FPGA's architecture and programming model are given in Sections 2.3.1 and 2.3.2.

### 2.3.1 GPGPU Computing

GPUs are general purpose platforms with several SIMD cores. They are used as co-processors in order to enhance applications demand for more processing power. In 2006, NVIDIA introduced a GPU abstract programming model called Compute Unified Device Architecture (CUDA) [68]. It is an extension of C language and defines its own thread hierarchy for GPU platforms (Fig. 2.8, a). The bottom level contains individual threads that communicate through local memory. The intermediate level classifies threads on groups that are known as blocks. Threads within a block can communicate through shared memory (SM) while

their synchronisation can be forced with the use of a synchronisation barrier by using the specific statement of *syncthreads();* [104]. Finally, all blocks form the top level called as grid.

GPU's micro-architecture contains several multiprocessors (MPs) each containing multiple stream processors (SPs) (Fig. 2.8, b). Data are transferred from CPU to global memory (GM) that is implemented as DRAM while shared memory as SRAM. Shared memory is characterised by low latency and high bandwidth therefore is faster than global memory. Blocks of threads use global memory in order to share data between them. Threads in GPU differ from threads in common operating systems as they are much lighter. The programmer designs kernel functions while having in mind only one thread but then specifies the number of block threads and blocks per grid needed for the certain kernel. Each thread processes only one data element each time and each block of threads is independently assigned and executed on each SP.

Multiple threads are executed concurrently in GPU. From hardware perspective, threads are organised in groups of 32 called warps. The number of parallel executed blocks is restricted from the available resources in GPU as registers and shared memory. In reality, each SP executes a warp at a time and schedules the rest on a time sharing mode. Warp execution applies the same instruction on 32 threads and as concept is not exposed on the programmer. In case that each SP contains 8 functional units then it will spend 4 cycles to execute a warp of 32 threads. However, as 32 threads access shared memory simultaneously then a bank conflict may be created. This will have as result shared memory access to take much more cycles [105]. When the SM executes a memory instruction then it switches on a different warp until data are fetched from memory. The ideal scenario would be to combine all memory requests needed by a warp to only one instruction. All memory fetches can be combined together to a coalesced memory transaction if memory addresses are sequential [106]. If not, then a non-coalesced memory transaction will force warp to wait for all memory transactions and this can lead on performance degradation.

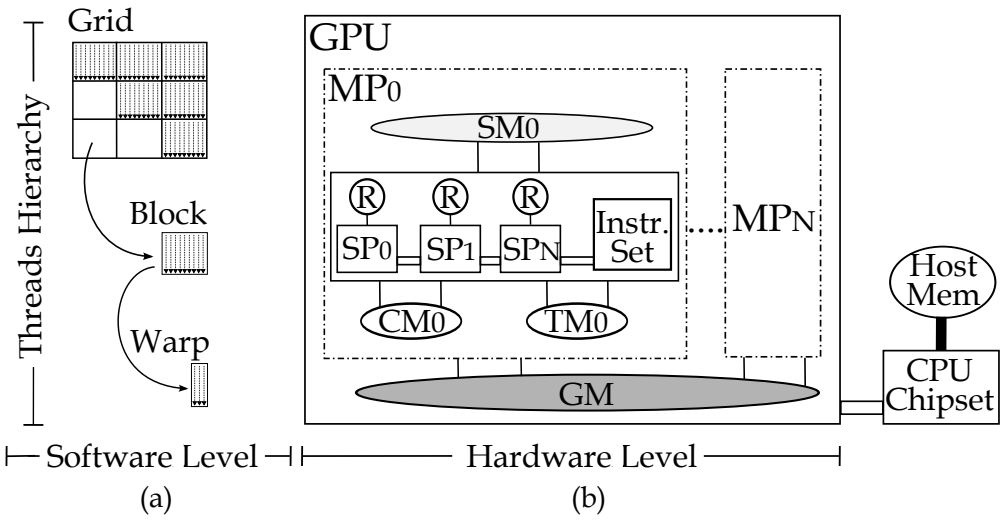


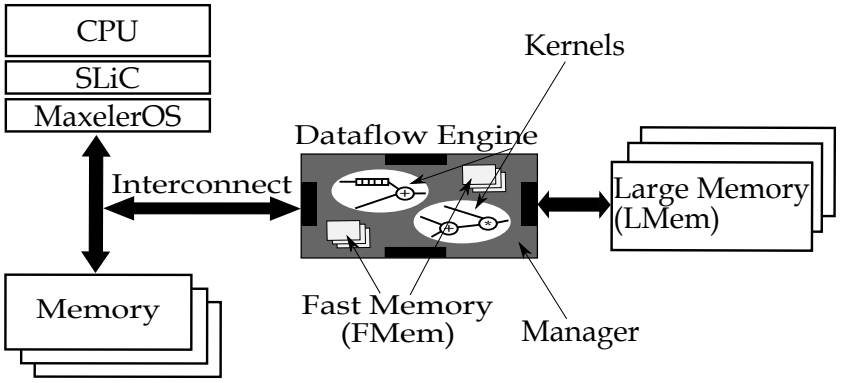

Figure 2.8: (a) CUDA Thread Hierarchy - (b) GPU Architecture.

### 2.3.2 Dataflow Computing

Dataflow computation, as discussed in Section 2.2.1, became feasible due to the introduction of the FPGA technology. The first commercially successful FPGA was developed by Xilinx in 1985 [107]. Reconfigurable devices as FPGA are composed by multiple configurable logic blocks (CLBs). The principle of re-programmability characterises both the functionality and the routing resources between CLBs. FPGA programming differs from GPU programming as the latter provides a higher level of abstraction . Hardware programmer needs to provide a behavioural description of the needed design while having in mind lower hardware details. The procedure of converting an algorithmic description to a low level RTL (register-transfer level) design is known as High Level Synthesis (HLS). The algorithm described with a High Level Language (HLL) is compiled to a structural description and mapped on FPGA through logic and layout synthesis [108].

In 2003, Maxeler introduced a compiler for dataflow programmability in FPGAs [109]. The Maxeler framework describes the physical FPGA device as a Dataflow Engine (DFE). Each DFE can implement multiple kernels that describe the needed computation. Two types of memory are defined by the Maxeler framework: the Fast Memory (FMem) that is implemented as BRAM and the Large Memory (LMem) as DRAM (Fig. 2.9, a). The FMem can store on-chip several

(a) Maxeler Platform




(b) Dataflow Graph (DFG)

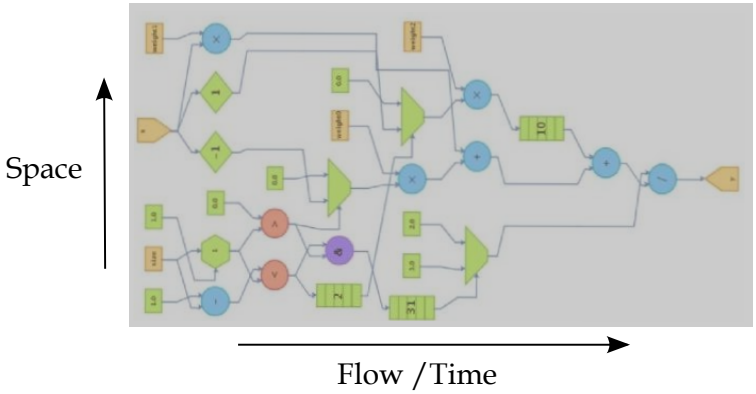



Figure 2.9: (a) Maxeler platform (b) Dataflow graph (DFG).

megabytes of data while LMem several gigabytes off-chip. A dataflow program contains at least one kernel and a manager that organises the data movement in the DFE. The code that runs on DFE is described by a Java library. Dataflow code compiles with MaxCompiler and links with CPU code to create an accelerated application executable. The Dataflow implementations are called from the CPU though the Simple Live CPU (SLiC) interface. While in CUDA devices synchronisation can be implemented with specific statements, FPGAs are more flexible and can implement various types of barriers as counter and tree based approaches that are directly implemented in the hardware [104].

In essence, kernels describe graph structures of pipelined arithmetic units (Fig. 2.9, b). If there are not any loops in the dataflow graph then data just flow from inputs to outputs. In the case of a dataflow graph with loops then data simply flow in a physical loop inside the DFE. As long as there are more data than the stages in the pipeline, the execution of the computation is extremely efficient. The combination of deep-pipelining and exploitation of both inter- and intra-kernel parallelism can enhance performance.

### 2.3.3 Graph Representation on HPC

A wrong choice of the data representation can affect the performance of the parallel graph algorithm. While hardware architectures are characterised by different memory hierarchies they also need a different data structure in order to exploit the maximum performance. It is very significant graph data to be represented in memory in such way that will not waste the available memory bandwidth.

In particular, in GPUs this can be done by minimizing the non-coalesced memory transfers [110] as mentioned in Section 2.3.1. However, the adjacency lists may not be the most efficient data structure for the GPGPU model [84]. As CUDA allows arrays with arbitrary lengths then adjacency lists can be compressed to one large array. This approach is named as Compressed Sparse Row (CSR) representation [84] and also known as Compact Adjacency List (CAL) [111]. It consists also an ideal graph representation for FPGA platforms [112]. CSR include an array *ind* that contains all the adjacency lists that define a graph. An array *ptr* with $|V|+1$ positions is used to determine which adjacencies belong

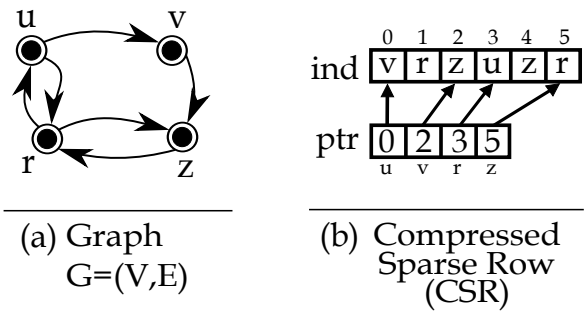



Figure 2.10: The representation of graph $G = (V, E)$ (a) can be compressed to a Compressed Sparse Row (c) where only non zero values are stored.

to which vertex. The elements stored in *ptr* are pointers to *ind* that indicate the beginning and the end of the adjacencies (Fig. 2.10,b). The size of *ind* is equal with the number of non-zero (nnz) values. If the graph is weighted then weights are stored on extra array named *val*. For unweighted graphs this array is not needed.

### 2.3.4 Performance Measures

#### Speedup

By using parallel computing we target to speedup computationally expensive algorithms. Parallel computing, as defined in Section 2.2.2, is the partition of a problem with size $n$ in smaller sub-problems that are simultaneously executed in $p$ processing units. The achieved speedup, due to parallel computation, is measured by comparing how much faster a parallel algorithm runs in contrast to the best sequential one. It is defined as the fraction of the serial $T_{serial}$ and parallel run-time $T_{parallel}$:

$$Speedup = \frac{T_{serial}}{T_{parallel}} \tag{2.5}$$

Speedup will be increased linearly if it grows as function of $p$. In such case, we call this linear speedup and assume that additional overheads of the algorithm are in the same percentage with its execution time. If $T_{parallel} = T_{serial}/p$ then we can talk about perfect linear speedup (Fig. 2.11). From a theoretical perspective

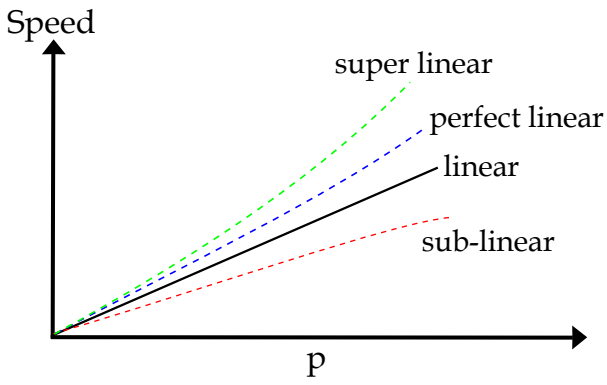



Figure 2.11: The four possible speedups.

this is the maximum feasible speedup in case that problem $n$ has a fixed size.

However, it is very unlikely to reach a linear or perfect linear speedup. Overheads, caused from threads forced by mutexes to be executed serially due to critical sections or memory bottlenecks, make it hard to achieve such a speedup. In reality algorithms manage to achieve sub-linear speedups (Fig. 2.11). However, Gustafson [113] considered that super-linear speedup would be possible if speedup is considered as ratio of speeds (speed=work/time) and not as ratio of times. The assumptions of constant work and fixed-size speedup are the major reasons making impossible the super-linear speedup [114].

Speedup can be classified based on three models. The fixed-size model assumes that the size of the problem $n$ is fixed and the number of processing units $p$ is variable. In contrast, the scaled speedup considers a variable $n$ and $p$ with constant problem size on each processing unit while the fixed-time speedup considers constant work per processing unit. In this thesis, we consider fixed-size speedup by default.

# Chapter 3

# Optimised perturbation analysis

## 3.1 Introduction

Real-world systems ranging from social networks till internet communication can be conceptualised as graphs [115], [116]. There is a great interest to understand how such systems react on external threats and be able to identify nodes that are critical for their robustness. Human diseases can be viewed as perturbations of highly interconnected cellular networks [117] while terrorist attacks can affect the international air traffic [118]. This reveals the importance and the need for tools that can simulate how such perturbations, created by node removals, can influence real-world networks.

Perturbation analysis is becoming a routine strategy for data interpretation in domains ranging from bioinformatics till sociology. It helps to assess their resilience on targeted or random node removals (Fig. 3.1) that can disturb their 'robust, yet fragile' nature [119]. Most real-world networks are extremely resilient on random failures and very sensitive on targeted attacks, as mentioned in Section 2.1. Common graph analysis tools as Cytoscape [34], [35] and Nexcade [16] provide an automated mechanism for perturbation analysis on Protein to Protein Interaction (PPI) networks, by using only targeted attacks based on centrality measures as outlined in 2.1. However, they can't predict what would be the best combination of nodes to remove from a graph in order to cause a higher impact. Such high-impact combinations are unlikely to be found unless they are

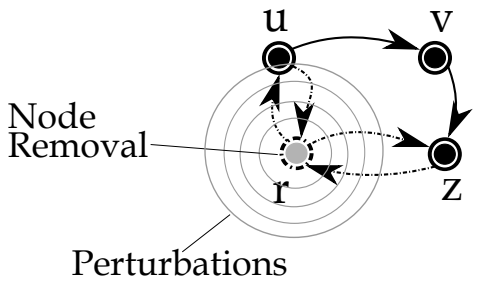



Figure 3.1: Perturbations created in graph $G$ after removing node $r$.

specifically optimised or searched for [36].

As the real-world graphs are characterised by properties that are not yet well understood, similarly the search space in such graph problems is not so well understood and relatively unstructured. In such irregular environments, heuristic strategies as genetic algorithms (GAs) can provide a powerful heuristic search for large complex problems as the computation of high-impact combinations. They may not guarantee the discovery of an optimal solution however they will often provide a good solution if one exists. Their simplicity as algorithms as well as their power to discover good solutions rapidly for difficult search problems makes them appealing for the detection of high impact node removals.

In this Chapter, we provide a genetic algorithm that searches for optimised node removals that can achieve higher impact than random and targeted attacks. Our GA uses a population of boolean strings that represent different node removal patterns. Over time the survival of the fittest candidates favours better combinations of node removals. As this process is repeated over hundred times, it eventually converges in a well estimated combination of nodes where their removal produce higher impact. In this context, we evaluate our GA with real-world Protein Protein Interaction networks and show that the effectiveness of our genetic algorithm is much higher than the common perturbation strategies of random and targeted node attacks.

The remainder of the Chapter is structured as follows: In Section 3.2, we give a brief introduction about genetic algorithms. Section 3.3 introduces the overall methodology used for the design of our optimised perturbation analyser. In Section 3.4, we present the implementation of our genetic algorithm. Section 3.5 shows our experimental results. Finally, we conclude with Section 3.6 that

draws conclusions.

## 3.2 Background

### Perturbation Analysis

Perturbations in real-world systems can have catastrophic impact in their overall functionality. Diseases or terrorist attacks can arise from perturbations within the intermolecular communication or the international air traffic respectively [120]. Such phenomena reveal the importance and necessity to study and monitor how perturbations affect the robustness of a graph.

However, perturbations can only be studied with the help of simulations. Both single node removals and paired perturbations can reveal crucial information for the robustness of a graph. The simulation of sequential perturbations is a method used to monitor the behaviour of a graph in cascading failures. Different groups of nodes are sequentially removed from a graph and its behaviour is monitored continuously based on certain topological properties. Such simulations of sequential perturbations are commonly used by the research community as a standard technique to study the behaviour of graphs over sequential failures [16]. A simulated perturbation is performed either as a random or targeted loss of a node or a group of nodes. The targeted perturbation involves the removal of nodes that are ranked based on a topological property while the random approach corresponds in the random removal of nodes. Both random and targeted perturbation approaches provide inefficient node removal predictions. Therefore, the use of a heuristic search strategy based on evolutionary algorithms can improve the prediction of nodes that can affect more the robustness of a graph.

### Genetic Algorithms

The evolutionary algorithms are inspired by the principle of evolution in nature. They thrive in environments with large amount of candidate solutions where they apply a trial-and-error problem solving strategy in order to find the fittest solutions [121]. Genetic algorithms belong in the larger class of evolutionary

algorithms. Their main advantage is their ability not to be trapped in suboptimal local maximum or minimum. They exploit historical information to guide their search in regions with better performance and avoid a local maximum. Also, they try to gradually improve a group of solution candidates by utilising different variation operators as mutation and crossover.

In general, genetic algorithms work by creating initially a randomly generated population of several candidate solutions. Each candidate is evaluated based on a specific fitness function that measures how well they perform at a given task. Candidate solutions are also known as 'chromosomes' while each member of a chromosome is called as 'gene'. Based on a selection strategy the fittest candidates are selected as parents for a new generation. Their breeding process where new fitter offspring are generated is known as crossover. Such process helps to discard not well-performed chromosomes and keep only the best individuals. However, the crossover process can lead on a population with similar candidate solutions. The low diversity among chromosomes is encountered by using the mutation process that randomly alters their genes. Finally, a genetic algorithm is terminated when a suitable solution is found or fixed number of generations is reached.

## 3.3 Methodology

### 3.3.1 Optimised Node Attacks

The design of our genetic algorithm has been mainly based on the needs of sequential perturbations. As mentioned in Section 3.2, the model of sequential perturbations dictates the sequential removal of a certain number of nodes per step. As a result instead of letting the GA to disable a minimum number of nodes we use the needed constraints in order to follow the model of sequential perturbations. Each removal of nodes is totally independent from the previous one and only a predefined number of nodes is removed. As each step corresponds in sequentially increasing number of nodes, our GA uses this rule and searches for an optimised group of nodes to remove in each step.

Initially, our genetic algorithm creates a randomly generated population of

boolean strings representing different protein target sets (Fig. 3.2, a). The removed nodes contained in a boolean string, for the particular case of PPI networks, represent the deletion of certain proteins. The length of these strings is equal with the total number of nodes $n$ contained in the analysed graph $G$. Boolean string's true values represent the existent nodes in the graph while false values the removed ones. In essence, boolean strings represent multiple versions of the same graph $G$ but with different node combinations removed each time. Each string is evaluated based on a fitness function corresponding on the average connected distance $\bar{d}$ of graph $G$. Based on Equation 2.1, $\bar{d}$ is defined as:

$$\bar{d} = \frac{2}{n(n-1)} \sum_{u=1}^{n} \sum_{v=u+1}^{n} d(u,v) \tag{3.1}$$

where $n$ equals with the number of nodes in the graph. By computing the average connected distance, we quantitatively measure the alternation of graph robustness caused from the removal of the certain set of $c$ nodes. A fitness score $Fit_n$, expressing the impact created by the removed nodes, is assigned on each boolean string (Fig. 3.2, b).

The boolean strings with the highest scores are chosen as parents and contribute in the generation of new strings that contain better combinations of removed nodes. They are selected based on the tournament selection strategy. A tournament between two randomly chosen candidates is organised and the fittest one is chosen. Parents give birth to new children through the crossover process. In particular, two boolean strings belonging to different parents are divided in the middle and a new children is generated by combining each half of each parent to a new boolean string (Fig. 3.2, c). As it is hard to detect from the beginning which crossover type may provide the most efficient results we decided to use the simplest form of a single crossover point and eventually to alternate this process if the results are not efficient enough.

Furthermore, in order to keep population's diversity in high level our algorithm applies mutations on its offsprings. Boolean string's values are randomly flipped based on a certain probability (Fig. 3.2, d). However, the process of mutation could faulty increase the predefined number of removed nodes. For this reason,

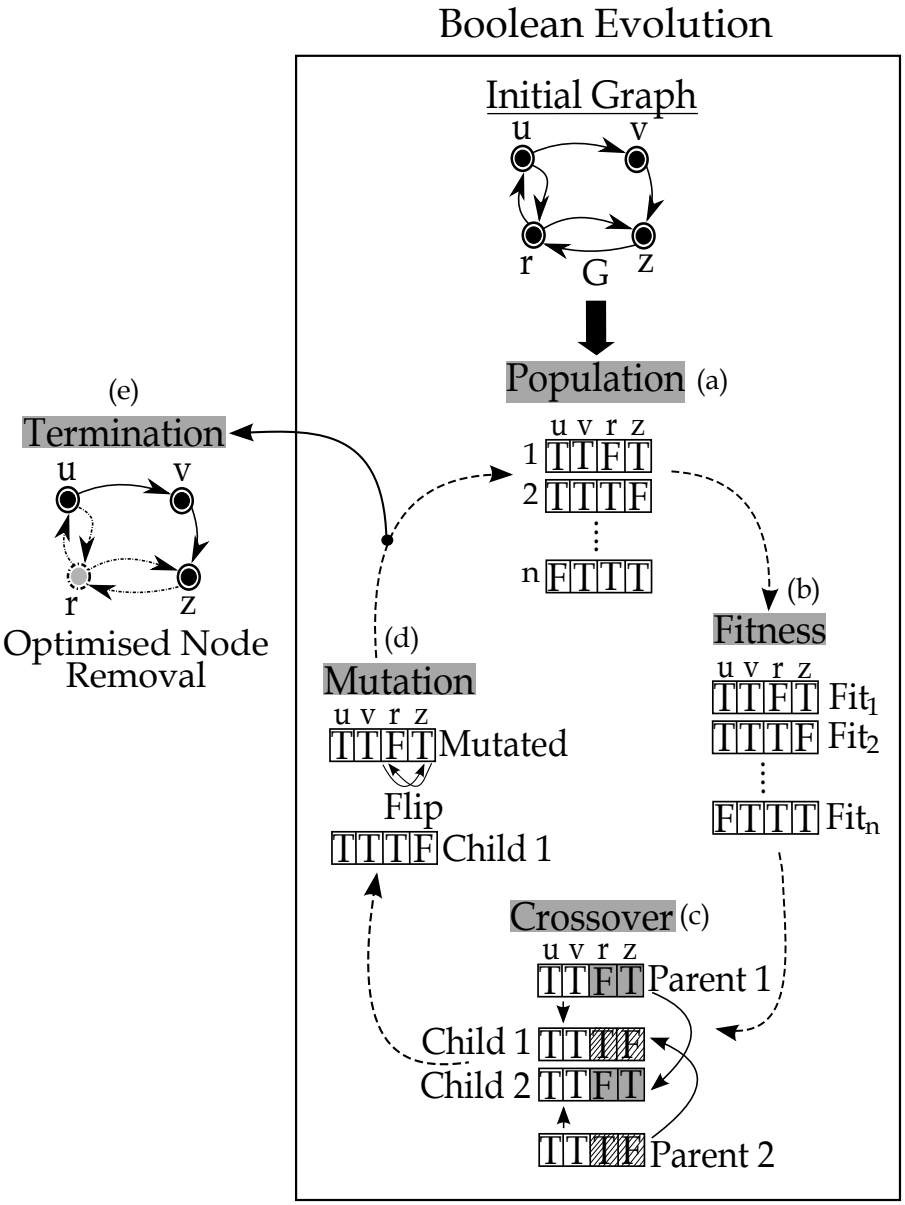



Figure 3.2: Optimising node attacks with the use of boolean evolution: An initial randomly generated population of boolean strings (a) is evaluated based on average shortest path (b). Their fitness score is used for the selection of best parents to generate new offsprings through crossover (c). Boolean strings are mutated in order to maintain diversity in the population (d). All the previous steps are repeated for many generations. The algorithm terminates by providing a well estimated solution of the needed combination of nodes to be removed (e).

we used a normalisation function that regularises the number of removed nodes back to the default one. Moreover, in order to combat the phenomenon of good candidates being lost due to crossover or mutation, we use a feature called elitism. This guarantees that a percentage of the elite boolean strings will be copied unchanged in the next generation. If for instance the size of our population is 100 then an elitism of 5 will mean that 5% of the fittest candidates will be copied unmodified in the next generation. All four steps of the GA are repeated for many cycles until a predefined number of iterations (Fig. 3.2, e).

## 3.4 Implementation

Our genetic algorithm has been developed through an extensible framework, called Watchmaker [122], that implements Java based genetic algorithms. The Watchmaker framework offers the basic outline of an evolutionary algorithm. It uses a loop that creates one generation per iteration and contains template functions that perform fitness evaluation, selection and crossover. The programmer needs to populate all the needed functions, adjust the mutation probability and choose a selection strategy. Finally, an evolutionary engine orchestrates all the needed steps of the genetic algorithm.

In our implementation, we specifically utilise an enhanced tournament selection process provided by the Watchmaker framework. A probability parameter is used as a mechanism to adjust the selection pressure. This process involves the generation of random values between zero and one that are compared with a predefined probability. If the probability is larger from the predefined one then the fitter boolean string is selected while if it is smaller the weaker one is selected. In practice, the probability needs to be above 0.5 in order to favour the selection of fitter candidates. By giving a higher probability in individuals of higher quality new better generations are created that eventually can help GA to focus in more promising areas in the search space [123].

What differs at most in each GA implementation is the used fitness function. In our implementation, each boolean string is evaluated based on the computation of average connected distance. In essence, the removed nodes represented in each boolean string are depicted on different adjacency matrices (Fig. 3.3, b,d).

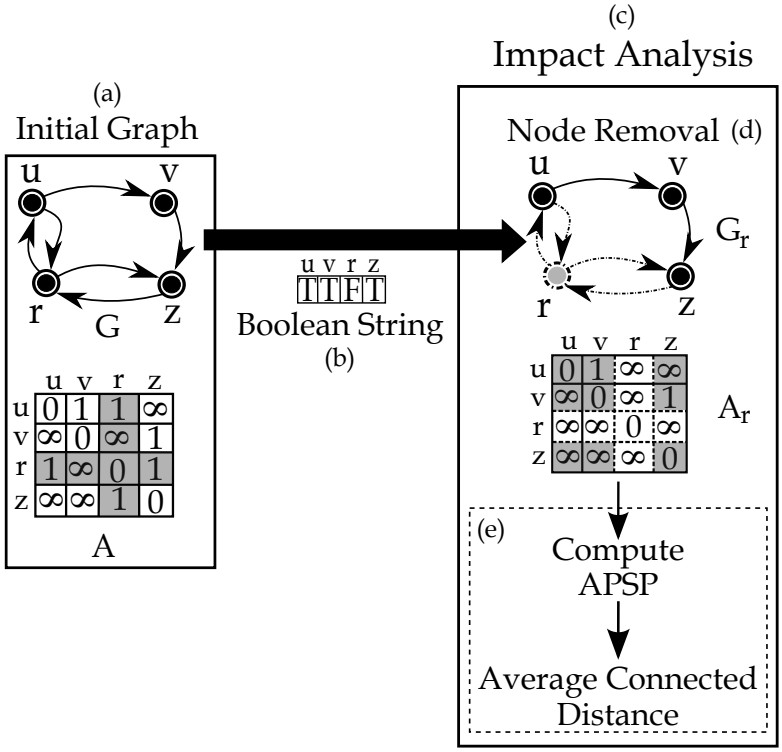



Figure 3.3: Basic steps of Graph Impact Analysis: Let $G$ be the initial graph represented as adjacency matrix $A$ (a). Each boolean string dictates the removal of certain nodes (b). Let $r$ be the removed node then this is implemented by creating a new adjacency matrix $A_r$ without the in- and out-going edges of $r$. The impact created on graph $G$ by the removed node $r$ is measured by computing APSP for the matrix $A_r$ and then calculating the average connected distance.

By computing the all pairs shortest paths (APSP) on each of them we provide the needed values for the calculation of the average connected distance for each boolean string (Fig. 3.3, e).

## Node Removal

Let $G$ be a directed graph represented as an adjacency matrix $A_{ij}$. As we mentioned in Section 2.1.1, connections between vertices in an adjacency matrix are represented with values of one while non-existing edges with infinity $\infty$. The removal of a node $r$ from graph $G$ disables its functionality but also affects all the other vertices directly connected on this node. It practically means the elim-

ination the incident in-going and out-going edges of the removed node (Fig. 3.3). Edges belonging on the removed node $r$ are deleted by creating a new adjacency matrix $A_r$ and replacing the existing edge connections with infinity. The disturbances created in the graph due to the removal of node $r$ are quantitatively measured by computing the average connected distance based on resulted matrix $A_r$.

### Random and Targeted Node Attacks

The same process of node removal was also used by the random and targeted attacks that we used as benchmarks in order to compare the effectiveness of our genetic algorithm. The process of perturbation analysis evaluates how the removal of a single node or a set of nodes affects the structure of a graph. Our simulator implements the random impact analysis by using a random number generator in order to choose which node or set of nodes will be removed from the graph (Fig. 3.4). In case of removing a set of vertices it will not allow the removal of a node more than once.

On the other hand, a targeted approach removes nodes based on certain topological features. Our simulator uses network centrality measures that are closely connected with the graph robustness. Each vertex $v$ is assigned with a numeric value based on the used centrality function $CN(v)$. This simulator uses the degree $CN_{degree}(v)$ and betweenness $CN_{betweeness}(v)$ centrality functions in order to conduct targeted attacks (Fig. 3.4). As we mentioned in Section 2.1, degree centrality is defined as the number of edges connected on a node while betweenness centrality is the number of shortest paths between all nodes that pass through a vertex $v$. In particular, for directed networks the degree centrality counts both in-going and out-going edges. It is calculated as the sum of in-degree $deg^+(v)$ and out-degree $deg^-(v)$ where

$$deg(v) = deg^+(v) + deg^-(v) \tag{3.2}$$

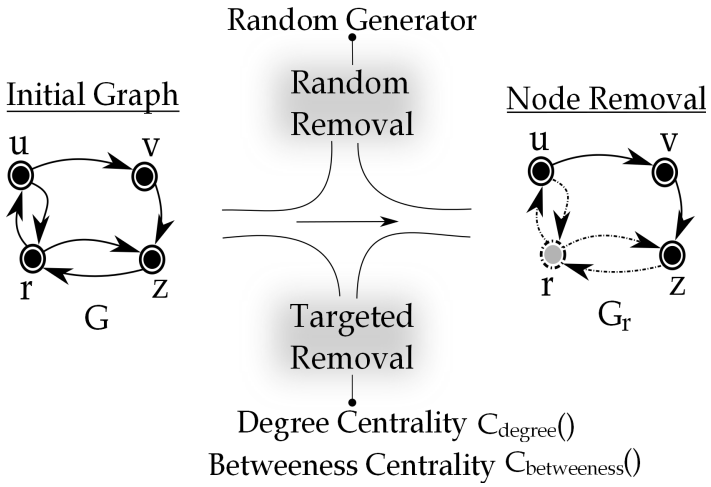



Figure 3.4: Random and Targeted node removal.

## 3.5 Evaluation

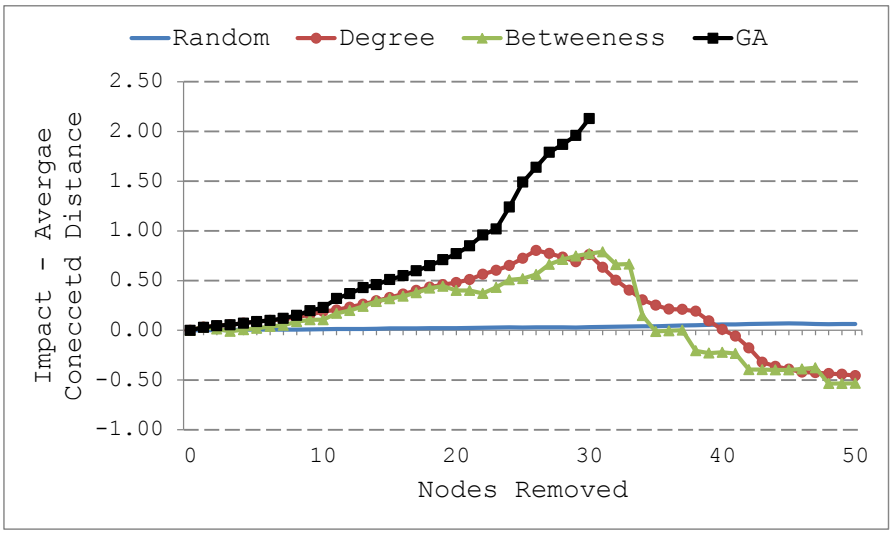
The performance of our genetic algorithm has been assessed by conducting extensive tests on real-word graphs representing protein-protein interaction (PPI) networks. The real-world graph data have been provided by e-Therapeutics [4]. The tested networks range from 87 to 561 vertices and contain up to almost 7000 edges. The experiments were conducted on a single Intel Core i5-2500 CPU processor @3.30 Ghz (4 cores) with 3.9GB RAM, running Ubuntu 12.4.

In all the experiments conducted for our GA algorithm, the population of candidates has a fixed size of 100 boolean strings while the whole GA runs for 1000 iterations. Furthermore, as mentioned in Section 3.4 we used an enhanced tournament selection process with a probability parameter known as selection pressure where we set it to 0.7. Moreover, we used a probability of 0.001 for our boolean array mutator and 5% of elitism. The algorithm terminates if there is no improvement in fitness score after 100 iterations. Also, the APSP computation was implemented with the Floyd-Warshall algorithm that has a cubic complexity (Section 2.1.1).

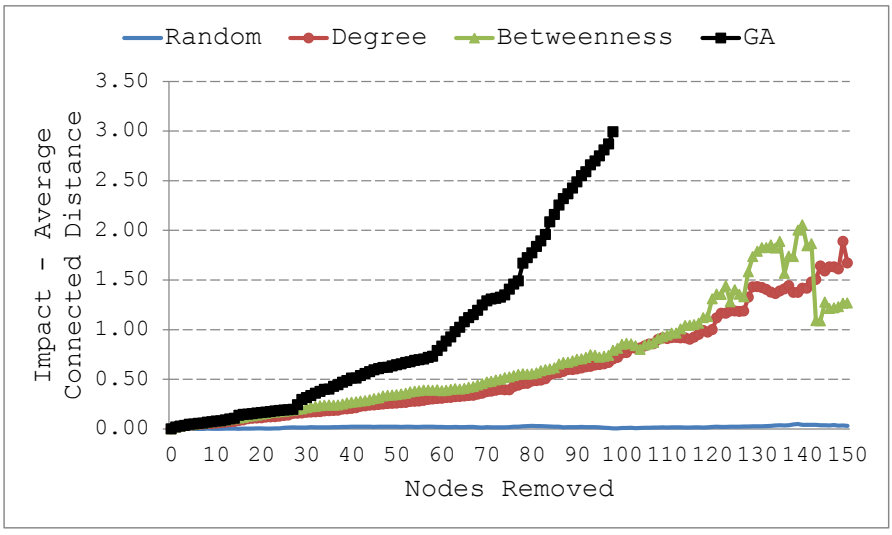
## Experimental Results

The simulation begins by measuring the average connected distance without any node removal. It proceeds by removing groups of nodes with augmentative number, beginning from an individual node till a predefined number of vertices. The effectiveness of the optimised analysis based on our genetic algorithm was evaluated based on one random and two targeted impact analysis strategies. The two targeted strategies are conducting attacks with the use of degree and betweenness centrality. After computing the degree and betweenness centrality for each node then vertices are ranked based on these topological features from the maximum to minimum value. The simulator begins by removing initially the vertices with the highest values and monitors the alternation of graph robustness by measuring the average connected distance. As nodes are sorted based on their significance inside the network, the targeted attacks on these nodes cause important perturbations in the graph comparatively to the random attacks. Due to the simulation of sequential perturbations each removal of nodes is independent from the previous one. As a result we don't need to compute the network centrality every time that we remove nodes from the graph as in each sequential step we use the initial form of the analysed graph. This means that node removals performed in the previous step are not reflected in the next ones. Consequently, a removal in the previous step will not affect the structure and connections between nodes in the next sequential step.

In all four tested cases of variable graph size (Fig. 3.5, Fig. 3.6), the random removal of nodes doesn't cause any effect in the robustness of the graphs. However, the targeted attacks based on the degree and betweenness centrality achieve greater disturbances in the structure of the graph. Attacks based on betweenness centrality are much more effective than the attacks conducted based on degree. Both the structure of the analysed graph and the nature of betweenness centrality as a measure force that behaviour. The betweenness centrality in comparison to degree centrality is a global based measure that is able to detect more critical interconnections between nodes so that's the reason leading on attacks of better effectiveness (Section 2.1). Both types of targeted node attacks, after the removal of a certain number of nodes, can lead on a point that the whole graph




(a)



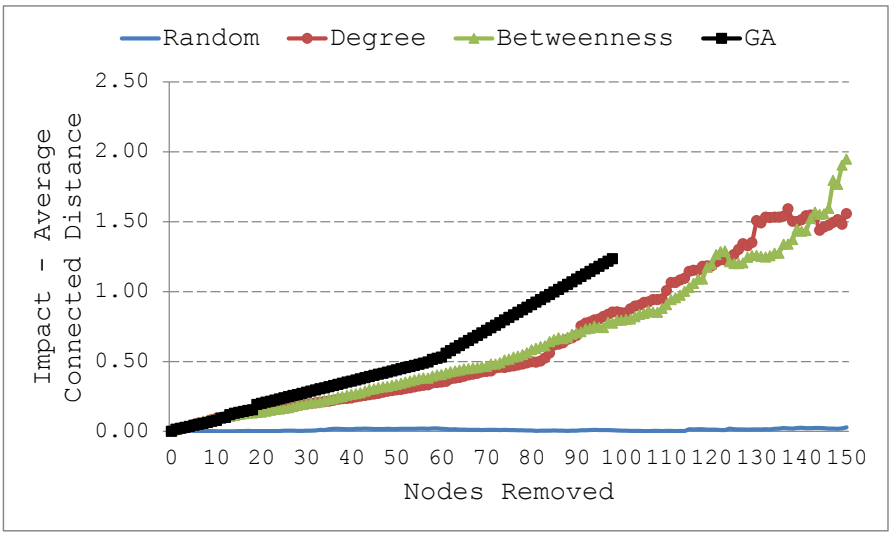

(b)

Figure 3.5: Comparing random, targeted and genetic impact analysis on real-world protein networks. Sample networks: (a) 87 nodes - 404 edges (b) 279 nodes - 3399 edges.

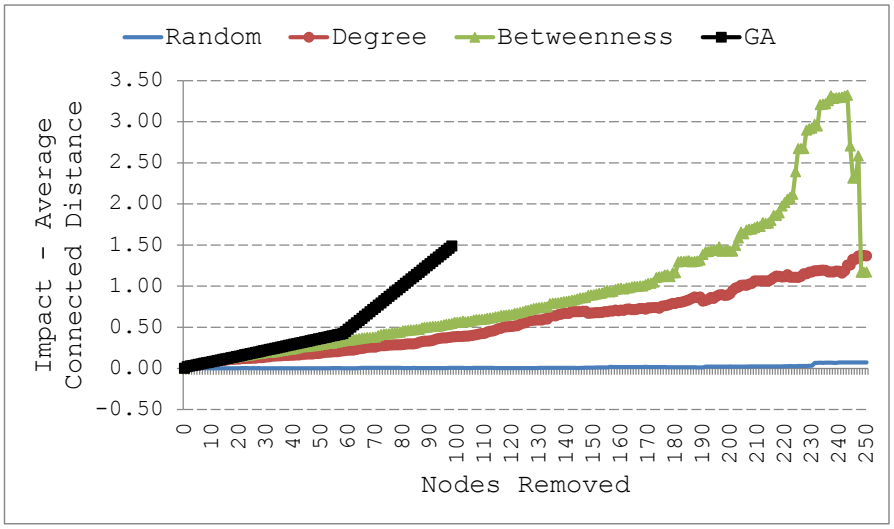
breaks down to a large number of smaller unconnected components. This point is known as percolation threshold [40], [124] and consists the critical fraction of nodes where their removal disintegrates the whole graph (Fig. 3.5, Fig. 3.6). If we keep moving nodes after the percolation point, we observe that the average connected distance is decreased and can reach negative values meaning that no many interconnections are left in the graph (Fig. 3.5, a).

Furthermore, we apply our GA to compute nodes removals ranging from individual nodes till groups of 30 nodes in the first test case (Fig. 3.5, a) and 100 nodes in the rest test cases (Fig. 3.5, b & Fig. 3.6). We observe that our GA selects much better node combinations and its effectiveness is much higher than the random and targeted strategies. In particular, Figure 3.5 depicts the impact of node removals on two real-word graphs with different size ranging from 87 till 279 nodes and 279 till 3399 edges. In both scenarios our GA algorithm for the first 10 and 20 removed nodes respectively has the same impact as the targeted and random impact analysis (Fig. 3.5, a,b). However, after that point our GA begins to attain better effectiveness and reaches a point of notable impact difference with the other strategies. Same behaviour is also regarded in the two other tested graph cases where our GA similarly as before reaches a better level of impact effectiveness (Fig. 3.6). As an overall observation we could comment that GA's effectiveness may also be affected by the structure of the graph and this can explain a better impact in some cases as in Figure 3.5, b.

Moreover, we examine the quality of our GA algorithm with an optimum solution. We compare the effectiveness of our GA with a brute force algorithm that exhaustively searches for nodes of high significance. While the brute force approach is highly computational expensive process we perform the needed evaluation only with a small graph of 87 nodes and 404 edges. We apply both brute force and GA in the exploration of effective node removals ranging from individual nodes till groups of 30 . We observe that our algorithm computes node combinations with a relatively same efficiency as the exhaustive search (Fig. 3.7). Our GA algorithm provides a sufficient heuristic search solution for the problem of graph impact analysis . However, its effectiveness can be further enhanced by alternating its genetic operators. For instance, the use of two random crossover points instead of one can explore more solutions and eventually provide more




(a)




(b)

Figure 3.6: Comparing random, targeted and genetic impact analysis on real-world protein networks. Sample networks: (a) 349 nodes - 3228 edges (b) 561 nodes - 7179 edges.

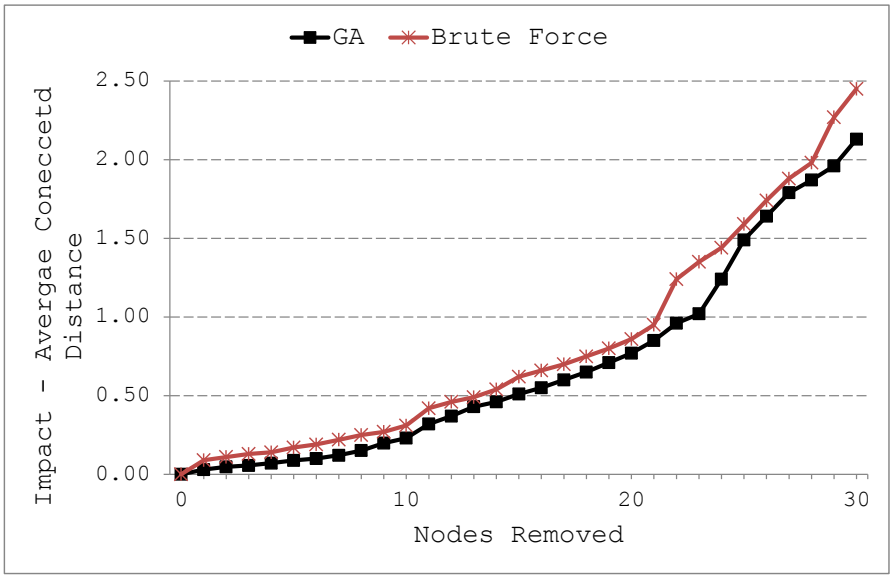



Figure 3.7: Comparing brute force and genetic impact analysis on real-world protein networks. Sample network: 87 nodes - 404 edges.

enhanced results. Moreover, GA's effectiveness could be enhanced by replacing the computationally expensive fitness function with a less accurate but low-cost fitness function [125]. This strategy is known as evaluation relaxation, however, is not ideal for our GA as it may produce results of low accuracy. Such margin of error is not tolerable by critical applications as the drug discovery process as it may affect the pharmaceutical impact of drugs.

Our GA provides solutions of satisfactory effectiveness, however, this comes with a price. The overall execution time for all the tested graph samples is relatively high and gets even worse for larger graphs. In particular, our GA needs 9 hours to compute a group of combinations ranging from 1 till 100 nodes for a graph with 561 nodes and 7179 edges (Table 3.1). Consequently, the high computational load diminishes the overall performance of our GA. Similar problems have been countered in the past and already addressed in literature through the parallelisation strategy where the computational load is distributed among different processors [126]. Instead of workload distribution it has been also proposed either to distribute the fitness evaluation over different processors or create smaller populations that can be assessed faster [127]. In our GA the most demanding operator is the fitness function that is based on the average connected distance.

While such metric is given by the highly computational expensive APSP algorithm this results in the deterioration of our GA's overall performance. The high execution time can affect negatively the drug discovery process as it can delay the research for new drug solutions that are highly needed by patients. Therefore, in the next two chapters we target in the acceleration of APSP computation where we propose two novel parallel APSP approaches based on different parallel processing architectures and algorithmic strategies.

Table 3.1: Execution time of used impact analysis strategies: Targeted (Betweenness, Degree), Random and Genetic. Time in Minutes.

| Nodes | Edges | Betweeness | Degree | Random | GA |
|---|---|---|---|---|---|
| 87 | 404 | 1.11 | 1.06 | 1.01 | 21.34 |
| 279 | 3399 | 2.02 | 1.36 | 1.30 | 152.4 |
| 349 | 3228 | 2.12 | 1.27 | 1.22 | 77.5 |
| 561 | 7179 | 15.09 | 11.13 | 12.18 | 569.1 |

## 3.6 Summary and Conclusions

In this Chapter, we provided a genetic algorithm that detects combinations of highly optimised node removals. Our algorithm has been evaluated on real-world protein interaction networks. It was shown that our genetic algorithm computes node removals that can affect graph's robustness more effectively than the common random and targeted impact analysis strategies. However, the heuristic search properties of the genetic algorithms are coming with a high price. All the solution candidates need to be assessed with a fitness function based on the average connected distance. Its calculation requires the computation of the highly expensive APSP algorithm for the whole graph. Such calculations lead in a higher execution time as even more fitness evaluations are required for all the solution candidates in every new population.

The overall execution time of our GA can be minimised by accelerating the APSP computation that is needed for the calculation of the average connected

distance. Parallel processing architectures can provide enhanced processing resources that can help the acceleration of the APSP computation. However, the use of parallel computing for parallel graph processing is very challenging due to the irregular structure of the real-world graphs. In the next two Chapters 4 and 5, we present two parallel APSP implementations based on GPGPU and FPGA computing. The GPU-based approach parallelises the APSP computation by utilising the divide and conquer strategy while the FPGA algorithm expresses the APSP computation in linear algebra calculations that are executed concurrently.

# Chapter 4

# Tailoring graph algorithms over GPUs: Multi-Layer Graph Decomposition

## 4.1 Introduction

Graphs describe systems as diverse as the World-Wide Web or social and biological communities. The representation of a system as a network of interconnected vertices provides one analysis method aimed at generating and understanding the global behaviour of that system [128]. Epidemiologists [22], sociologists [129], computer scientists [130] and neuroscientists [131] explore the way that viruses, ideas, information and electrochemical signals are spread along that systems respectively. While the use of graphs is expanded in a range of research fields, there is need to develop tools that measure and capture their underlying organizing principles [132]. However, computing quickly over these networks is a challenge for both algorithms and architectures due to their size and structure [133].

Networks with large node and edge counts push current network analysis techniques to their limits and the processing power available in a CPU is unable to analyze such networks efficiently. Current graph analysis packages as Pajek [134], Igraph [135] and Gephi [136] are limited by the available processing resources of

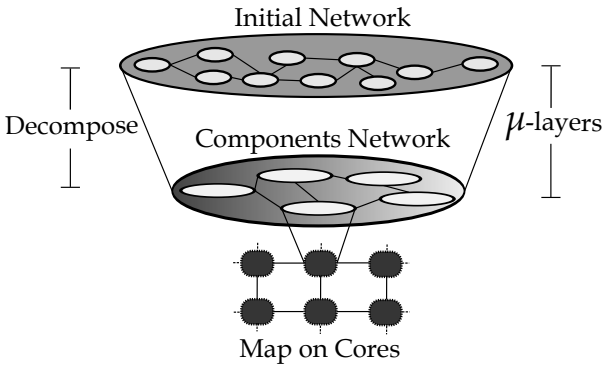



Figure 4.1: Network Decomposition: Each component of the new abstract network is mapped and processed in parallel on a different core.

commonly used workstations. Application of multi-core computing approaches could improve the efficiency of network analysis. However, such analysis pose significant challenges to parallel processing. Non-contiguous and concurrent access to global data structures with low degree of locality are the main problems [137].

Recent progress in parallel graph algorithms addresses these challenges through innovative data structures, memory layouts, and SIMD optimizations [84, 138, 139]. However, new algorithms and implementation strategies are required for efficient processing of current generation graphs on modern multi-core architectures. Such strategies should help algorithms and their implementations benefit from the properties of the graphs. The accurate segmentation and mapping of a network onto different cores is almost impossible. However, the mapping complexity can be reduced by exploiting properties of real-world networks.

Design and implementation decisions based on the nature of the graph consists an emerging research area. Current parallel APSP algorithms do not take advantage of the real-world graph properties [86, 88, 91]. Most of the recent studies used graph properties to develop trimming techniques for small-world graphs [140] and computation of biconnected components in symmetric multiprocessors [141]. Only Banerjee et al. proposed in 2015 a GPU implementation based on a graph pruning technique and used of bi-connected components (BCC) in order to compute APSP [93]. However, the success of such approach is based on finding good partitions and mainly experimented with planar graphs to ensure good partitions [91]. As we discussed in Section 2.2.5, there were many attempts

to accelerate APSP on GPU platforms. In this study, we build upon ideas from two previous works related with a divide and conquer approach that decomposes a graph in several components [91] and implementation decisions based on the nature of the graph [93]. We incorporate them into an efficient parallel graph processing pattern for GPU platforms. Due to the high clustering coefficient [142] real-world graphs can be partitioned by discovering its strongly connected components. Since graph connectivity can be used to detect similarity in a graph [143], we employ the notion of strongly connected components to partition our graph in further smaller groups of similar connectivity.

In this chapter, we present a Multi Layer Network Decomposition ($\mu$-Layer) that works as a general pattern for the analysis of networks on multi-core architectures. The $\mu$-Layer algorithm takes its name from the fact that it decomposes a network, based on its strongly connected components, into a layered organization and then uses these layers to represent that network within the multiple cores of the GPU (Fig. 4.1). The current implementation takes a two layer approach named Double Layer Network Decomposition (DLND) but, in theory, the algorithm is generalizable to more layers. We introduce a new algorithmic approach whose design is driven by the multi-core architecture of the GPU platform. A graph is decomposed into smaller modules without loss of information. We show that any graph $G = (V, E)$ can be decomposed into components that can be mapped to multiple cores. A new formula $\mu$, relates the number of blocks of threads $N_B(k)$ and the number of analysed components $C_{num}(G)$ and defines the needed level of decomposition. This is feasible due to the introduction of a novel data structure called $\Delta$, that controls the balance between the number of the cores that a multi-core processor contains and the number of components that are needed to be analyzed in parallel.

Furthermore, we present an efficient graph processing technique on GPUs, synthesised by four separate kernel functions that represent the needed steps to implement a divide and conquer approach. In this chapter, we implement the algorithmic structure needed for the all pairs shortest path computation over the $\mu$-Layer approach. We evaluate the $\mu$-Layer algorithm by using a two layer ($\mu = 2$) implementation as the number of blocks of threads in the used GPU are enough to analyse all the candidate components of the tested graphs. Consequently, there

is no need for further decomposition. The algorithm has been characterised across a number of graphs with random, scale-free and small-world structure, generated by a well known network platform called NetworkX [144]. Our experiments show that a two layer implementation ($\mu = 2$) achieved acceleration across all different network structures relative to a serial Bellman-Ford approach implemented from the igraph [135] platform and a parallel Bellman-Ford approach based on the current state-of-art parallel graph library LonestarGPU [90].

The remainder of this chapter is organised as follows: We give the background in Section 4.2. Section 4.3 gives the methodology of the new $\mu$-Layer algorithm. Section 4.4 describes the shortest path computation over a two layer ($\mu = 2$) implementation of the $\mu$-Layer algorithm. Section 4.5 shows the experimental results. Finally, we conclude with Section 4.6 that draws conclusions.

## 4.2 Background

Over the last decade, it has been revealed that real-world graphs have totally different characteristics from artificial graphs as tree and hypercubes [145, 146, 147]. They are not based on any explicit structure but they naturally evolve and grow. Graphs as social networks, web graphs and protein protein interactions belong on this category. A range of interesting properties, characterising these real-world graphs, have been already identified. The most known feature is the small-world property where the diameter of even large graphs remains small and shrinks rapidly if some edges are randomly rewired [145]. They are also characterised by high clustering coefficient meaning that nodes tend to cluster together in high degree. That feature can create a natural way for graph partitioning based on its Strongly Connected Components (SCC). In graph theory, an SCC is a maximal subgraph where there exists a path between any two vertices in the subgraph [140]. Real-world graphs tend to have a giant SCC of size O(N), where N denote the total number of nodes in the graph [147]. The rest SCCs have small size, however, they tend to be more frequent than large-sized components [148].

While each directed graph can be decomposed to a set of disjoint SCCs, we use this concept to divide our analysed graph on further smaller subgraphs that are processed concurrently. A classic sequential method for detecting SCCs in a

graph is Tarjans algorithm [149] with a linear-time complexity. However, Tarjans algorithm cannot be easily parallelised as it is an extension of depth-first search (DFS) algorithm that is inherently sequential [150]. For that reason we compute the SCCs in CPU and transfer the needed data in GPU.

Our DLND algorithm is a divide and conquer approach that processes all strongly connected components concurrently. Both component and initial graph graph are represented through the Compact Adjacency List (CAL) data structure (Section 2.3.3) that stores only the non-zero values. Blocks of threads are responsible to process concurrently each component and calculate the needed network metric. In this Chapter, we focus on the acceleration of the all-pairs shortest path (APSP) computation. We developed a framework that can be adopted by many network algorithms and function as a scaffolding for their further acceleration.

## 4.3 Methodology

The development of the $\mu$-Layer algorithm is based on the divide and conquer approach. A problem is divided into multiple sub-problems that are solved independently and the individual results are combined to produce the final answer [79]. Since graph connectivity can be used to detect similar functionality within a graph [143], we employ the notion of strongly connected components to partition our graph in further smaller subgraphs that are processed concurrently in GPU. Since NVIDIA's GPU threading model is organised on blocks of threads and the size of the analysed components is irregular, we employ a new data structure that facilitates the concurrent process of irregular sized components by thousand threads.

### 4.3.1 Overview

Let $G = (V, E)$ be an initial directed graph which can be partitioned in further smaller sub-graphs by discovering its strongly connected components. A strongly connected component is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \longrightarrow v$ and $v \longrightarrow u$, where vertices $u$ and $v$ are reachable from each other [79]. Our algorithm utilises the concept

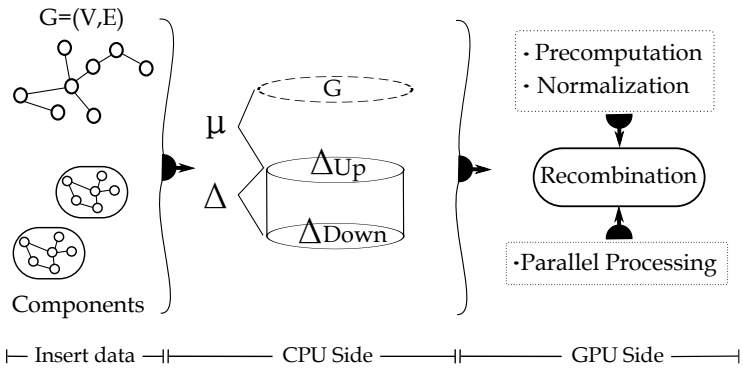



Figure 4.2: The CPU side uses the $\mu$-Layer concept to define the needed level of decomposition in order to map all the inserted components in different processing resources and synthesizes the needed double layer representation $\Delta$ based on the initial graph $G$. The last two steps of the divide and conquer approach, both parallel processing and recombination of subnetworks are processed concurrently in the GPU side.

of SCCs by mapping and processing multiple components concurrently in GPU. The methodology of our $\mu$-Layer algorithm is based on two main steps (Fig. 4.2):

- **$\mu$-Layer Decomposition:** A new concept that defines the needed level of decomposition in order to assure the analysis of all the components by the existed blocks of threads provided by the used GPU.
- **Double Layer Representation ($\Delta$):** A novel data structure that controls the balance between the number of the cores that a multi-processor contains and the number of components that are going to be analysed concurrently.

Both steps are further described in the next subsections.

### 4.3.2 $\mu$-Layer Decomposition

A graph is decomposed until its components can be mapped on the available number of block of threads that a GPU architecture has (Fig. 4.3). This is done by forcing, in every level of the decomposition, the algorithm to analyse components less or equal than the number of blocks of threads. Let $N_B(k)$ be the number of blocks of threads and $C_{num}(G)$ the number of components that are needed to be analysed. If the number of components exaggerate the number

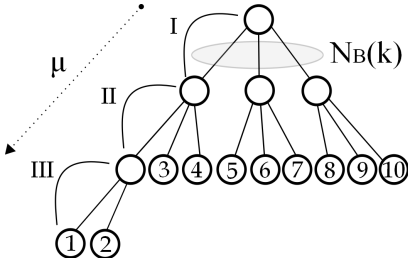



Figure 4.3: $\mu$ defines the number of layers that a network needs to get decomposed in order to fit inside a GPU. It is closely dependent with the number of available blocks of threads $N_B(k)$. For this specific scenario, where the number of components is equal with 10 and there are only 3 blocks of threads to analyse them, the needed levels of decomposition are equal with $\mu = 3$. Symbols I, II, III denote respectively the specific level of decomposition.

of available blocks of threads then the needed level of decomposition can be expressed through the inequality:

$$\begin{aligned} N_B(k)^{\mu} &\geq C_{num}(G) \\ \mu \cdot log_2(N_B(k)) &\geq log_2(C_{num}(G)) \\ \mu &\geq \frac{log_2(C_{num}(G))}{log_2(N_B(k))} \end{aligned} \tag{4.1}$$

where the number of needed levels of decomposition $\mu$ can be further expressed as the ceiling of the logarithm of the number of components with base the number of available blocks of threads :

$$\mu = \left\lceil log_{N_B(k)}(C_{num}(G)) \right\rceil \tag{4.2}$$

For instance, suppose the $\mu$-Layer algorithm needs to analyse 10 components but the available blocks of threads are only 3 (Fig. 4.4). Then due to the equation 4.2 the needed level of decomposition will be equal with $\mu = 3$. Three levels of decomposition are needed to fit all the components. The first level contains all the separate components. In the next level, three groups are formulated due to the number of available blocks of threads. As one of the group exaggerates in number

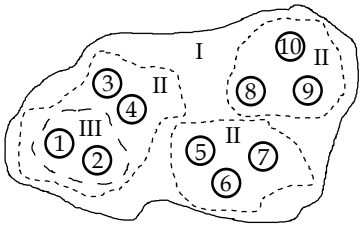



Figure 4.4: Example: In this specific scenario, where $N_B(k) = 3$ and $C_{num}(G) = 10$, the first level I contains the separate components of the initial graph. In the second level II, groups of three are formulated. As one of them contains more components than the available blocks of threads then it is further decomposed. The new third level III, that contains less components, makes possible now the mapping of the available blocks.

of components then it is further decomposed to fit on the 3 blocks of threads. This has as result the insertion of the third and final level of decomposition.

### 4.3.3 Double Layer Representation ($\Delta$)

Both the initial graph $G$ and its strongly connected components $C_1, C_2, \ldots, C_k$ are used to formulate the double layer representation ($\Delta$) in the CPU side. The $\Delta_{Up}$ layer denotes an abstract form of the initial graph G synthesized by its components and the lower layer $\Delta_{Down}$ stores the actual subnetworks of each component.

Suppose that the component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ has strongly connected components $C_1, C_2, ..., C_k$. The vertex set $V^{SCC}$ is $\{v_1, v_2, ..., v_k\}$, and it contains a vertex $v_i$ for each strongly connected component $C_i$ of $G$ (Fig. 4.5). There is an edge $(v_i, v_j) \in E^{SCC}$ if $G$ contains a directed edge $(x, y)$ for some $x \in C_i$ and some $y \in C_j$ [79]. This process forms the upper layer $\Delta_{Up}$ of the double layer representation $\Delta$. In Figure 4.6, each node of the $\Delta_{Up}$ level represents a condensed component $u_i$. Their initial form is stored as individual subnetworks $S_i$ in the $\Delta_{Down}$ layer.

In essence the nodes of the graph $G$ are the union of all components, where $V = C_1 \cup C_2 \cup \ldots C_k$. The new abstract graph $\Delta_{Up} = (V_{Up}, E_{Up})$ depicted in figure 4.7 can be defined as:

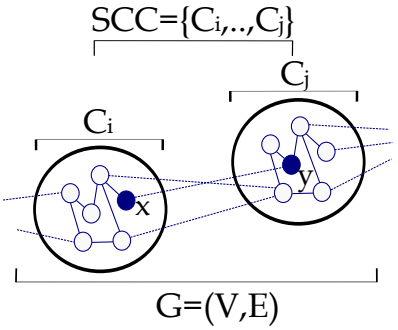



Figure 4.5: A set $SSC$ of strongly connected components is detected on the initial graph $G = (V, E)$. If at least one directed edge $(x, y)$ exists between the nodes of components $C_i$ and $C_j$, then a directed edge will exist between nodes on the abstract network $G^{SCC}$.

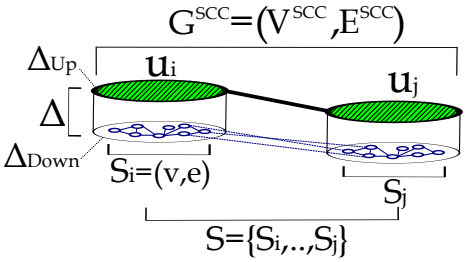



Figure 4.6: The outcome of the graph decomposition is a double layer representation $\Delta$. Each node $u_i$ of the component graph $G^{SCC}$ in the $\Delta_{Up}$ layer represent a condensed component. The $\Delta_{Down}$ layer stores the original subnetwork $S_i$ of each component individually.

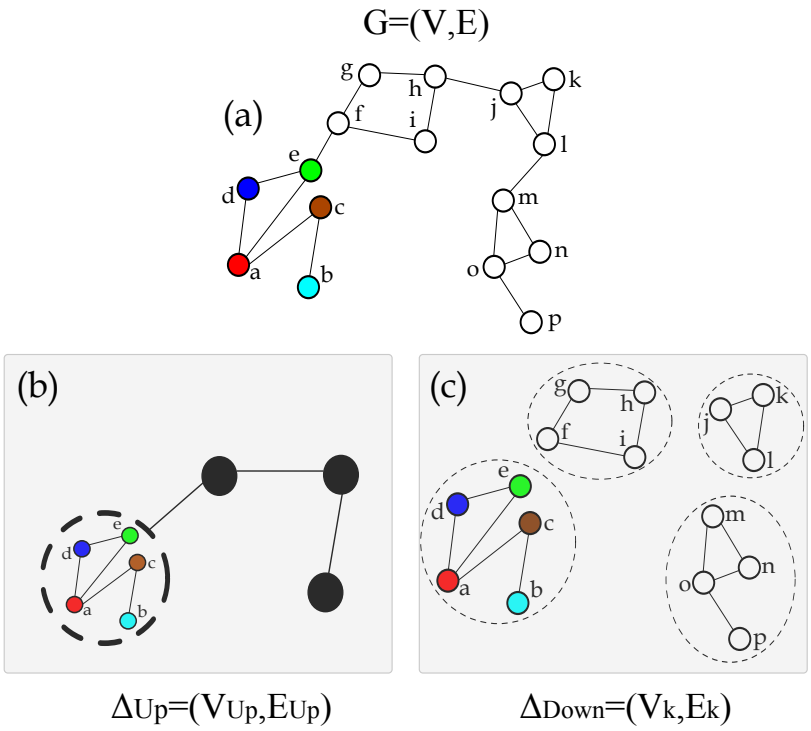



Figure 4.7: (a) $\mu$-layer algorithm decomposes an initial graph $G = (V, E)$ to two further new graphs: (b) $\Delta_{Up}$ where each node represents a condensed component and (c) a second graph $\Delta_{Down}$ that stores the initial state of the graph components.

$$\Delta_{Up} = \begin{cases} V_{Up} = \{C_1, \ldots, C_k\} \\ E_{Up} = \{(i,j) \mid (v,u) \in E, v \in C_i, u \in C_j\} \end{cases}$$

and the $\Delta_{Down} = (V_k, E_k)$ that corresponds to a subgraph of $G$ induced by $C_k$, as:

$$\Delta_{Down} = \begin{cases} V_k = C_k \\ E_k = \{(v,u) \mid (v,u) \in E, v \in C_k, u \in C_k\} \end{cases}$$

The content of the $\Delta$ structure is totally related with the number of the $\mu$-Layer decomposition (Fig. 4.8). In the specific scenario where $\mu = 3$, the $\Delta$ representation has 3 stages of evolution. The first level contains a network synthesised by the strongly connected components of the initial graph G. In the next step, where $\mu = 2$, groups of three components are formulating new

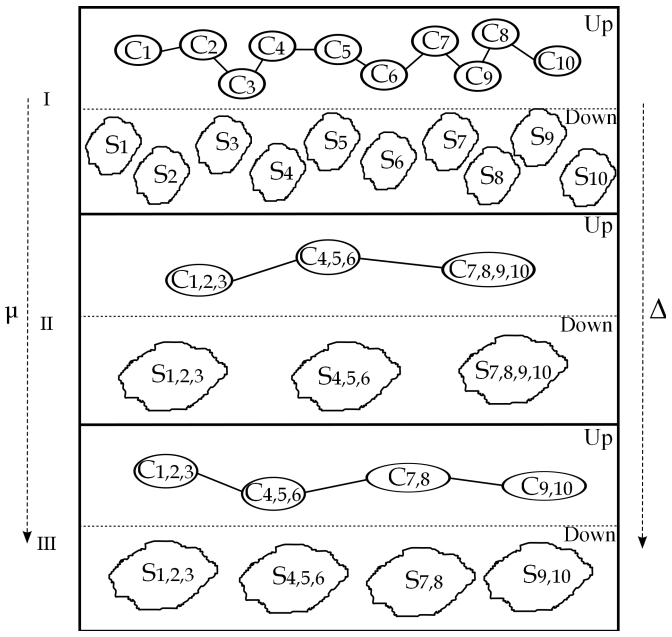



Figure 4.8: While $\mu$ is getting increased, the content of the $\Delta$ structure is evolved. For instance when $\mu = 3$, in the first level of decomposition the data structure contains the components of the initial network $G$. In the next level, groups of three components from the above level formulate new individual components. As the last component exceeds the number of three, this forces the last level of decomposition, where two further components are synthesised.

individual components based on the previous stage. As the third group contains more than three components, then it is furthered decomposed in the third level ($\mu = 3$) and formulates two more individual components. In each stage, the upper level $\Delta_{Up}$ of the $\Delta$ contains a network of individual nodes that represent the synthesis of more subnetworks that their real form is stored in the lower level $\Delta_{Down}$.

## 4.4 Implementation

The previous description of the $\mu$-Layer decomposition was generic for any value of $\mu$. In this section, we describe the implementation of the Double Layer Network Decomposition where $\mu = 2$. Our DLND algorithm is a divide and conquer approach that processes all the strongly connected components concurrently. We apply DLND on the calculation of APSP by using a parallel implementation of the Bellman-Ford algorithm [79]. Blocks of threads calculate APSP concurrently on each component and results from each component are combined to provide the all pairs shortest paths over the whole graph. Our decision to use the APSP calculation was driven by the fact that the APSP problem forms the basis of numerous network metrics and centralities needed by graph analysis tools as mentioned in Section 2.1.

### Overview

The DLND implementation of the Bellman-Ford algorithm is outlined in pseudo-code in Algorithm 3. The Double Layer Network Decomposition (DLND) algorithm, where $\mu = 2$, begins with the transformation of the initial graph $G = (V, E)$ to a compact adjacency list $G_{cal}$ that minimizes the memory storage (Algorithm 3). The same tactic is followed for the network of components $G^{scc}_{cal}$. Both $G_{cal}$ and $G^{scc}_{cal}$ compact adjacency lists are combined to form the double layer data structure $\Delta = (\Delta_{Up}, \Delta_{Down})$ that decomposes the initial graph $G$ into smaller size parts (Fig. 4.9). Briefly, in words, the algorithm consists of the following steps:

1. **Precomputation:** Needed variables as the largest component and needed

size of the new normalised structure are computed.

2. **Normalization:** As the number of nodes in each component is irregular, a new array is set up based on the largest component detected in the previous step in order to create a homogeneous data structure easily separable in equally numbered data portions.

3. **Parallel Processing:** Multiple blocks of threads compute the all pairs shortest path in each sub-network concurrently.

4. **Recombination:** The components detected on the shortest path in the upper layer are combined together concurrently based on common vertices existing between them, in order to provide the final answer.

We decided to use four separate kernels in order to accelerate not only the section of graph processing but also all the processes related with the pre- and post-processing of graphs. The two first kernels related with the precomputation and normalisation could be also executed in the CPU side. However, as GPU provides enhanced processing resources and the nature of the problems is embarrassingly parallel we decided to use GPU in order to enhance the overall speedup. Details of the individual steps are given in the following sub-sections.

## Precomputation Kernel

The precomputation $\text{Kernel}_{pre}(\ SCC, Size)$ takes as input a set of strongly connected components $SCC$ and detects the largest one $C_{Max}$. Its size multiplied with the number of components $C_{num}$ provides the size of the new array that needs to be allocated in the parallel processing kernel $Kernel_{paral}(\ Size, \Delta_{norm}\ )$ in order to implement the normalisation of $\Delta$. The precomputation $\text{Kernel}_{pre}$ as all the other kernels of the $\mu$-Layer algorithm are launched with the same number of blocks of threads $N_B(k)$. Its value is defined by the number of strongly connected components $C_{num}(G)$. The existence of that pairing is due to the implementation of the specific scenario where $\mu = 2$. There are enough blocks of threads available that can be equally allocated on the components without the need of further decomposition (Fig. 4.10). Threads $N_t(k)$ of each block of threads

**Algorithm 3** The multithreaded DLND algorithm that decomposes the initial network to multiple components that are mapped and processed by several blocks of threads.
**Input:** Initial directed graph $G(V, E)$ and Strongly Connected Components $S$CCs.
**Output:** Shortest path $S_{pi}$.

```
1: function MAIN( )
2:     Memory Allocation(Host)
3:      Synthesize Δ = (Δ_Up, Δ_Down)
4:      D^i_sp[] ← Compute all pairs shortest path in Δ_Up
5:     Memory Allocation(Device)
6:     Data Transfer(Host ⟹ Device)
7:      Kernel_pre [< N_B(k) >](SCC, Size)
8:                          ▷ [< .. >] denotes the kernel launch with N_B(k) threads
9:      Kernel_paral [< N_B(k) >](Size, Δ, Δ_norm)
10:    for u ∈ V do
11:        Kernel_comp [< N_B(k) >](Δ_norm, S_sp)
12:     Kernel_combin [< N_B(k) >](S_sp, S_pi)
13:    Data Transfer(Device ⟹ Host)
```

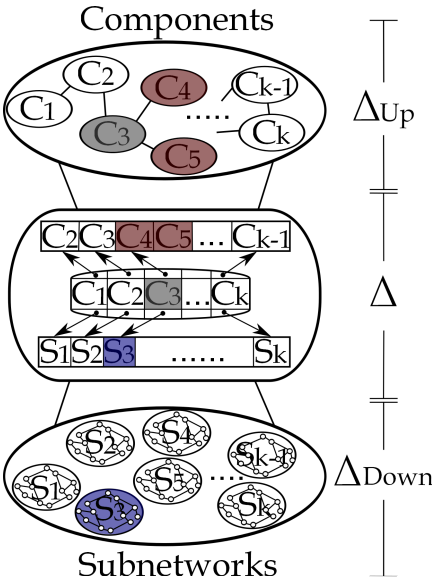



Figure 4.9: The decomposed network is stored on a Double-Layer data structure ($\Delta$). The $\Delta_{Up}$ is the network of components and the $\Delta_{Down}$ stores the related subnetworks. Both layers are implemented on a Compact Adjacency List (CAL).

are responsible for processing not only the allocated vertices but also its outgoing vertices (Fig. 4.11).

**Algorithm 4** Kernel that pre-computes needed variables. Detects the largest component and calculates the needed size of the new normalised structure.
**Input:** Strongly Connected Components $S$CCs.
**Output:** $S$ize of the biggest $S$CC.

```
1: function KERNEL_pre( SCC, Size )
2:     do in parallel
3:         C_Max ← Size of largest SCC
4:         Size ← C_Max × C_Num
5:   return Size
```

## Normalization Kernel

The CAL data structure representing the graph should be able to be partitioned into equally sized parts before insertion into the GPU. Separate block of threads will be responsible for their analysis. However, the number of nodes contained within the components $C$ is not uniform. Consequently, the structure representing the data should be normalised (Fig. 4.10). This is done by detecting the component with the largest number of nodes and based on that value a new homogeneous data structure $New[Size]$ stores the normalised data representation $\Delta_{norm}$. If the number of nodes in a component is less than the size of the array, then the empty cells are filled with a value denoting that. When the kernel is launched, equal parts of data are assigned on different blocks of threads. Each thread is responsible for processing each node of the component and its outgoing edges (Fig. 4.11).

## Parallel Processing Kernel

Once a kernel is launched each block of threads accesses a unique part of the CAL data structure $\Delta$ which denotes a different subnetwork $S_i$. By processing a copy of the same code, the shortest paths $S_{sp}$ are computed concurrently in all subnetworks. However, data dependencies don't allow the total parallelization of the SP algorithm.

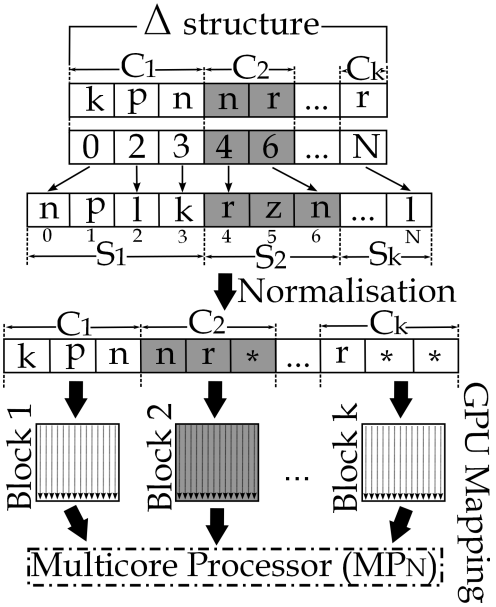



Figure 4.10: Each separated segment of the $\Delta$ data structure denotes a component $C_i$ with its subnetwork $S_i$. Data are normalised before they mapped on the different blocks of threads.

**Algorithm 5** Kernel related with the parallel process of subnetworks. Rearranging and normalising networks.
**Input:** $\Delta$ double layer data structure and $S$ize of the biggest $S$CC.
**Output:** $\Delta_{norm}$ normalised $\Delta$ data structure.

1: **function** KERNEL$_{paral}$( $S$ize, $\Delta$, $\Delta_{norm}$ )
2: **do in parallel**
3: Rearrange networks by using $New[$Size$]$
4: $\Delta_{norm} \leftarrow$Normalised $\Delta$
5: **return** $\Delta_{norm}$

The Bellman-Ford algorithm computes the distance of a node $v$, by comparing its distance with the sum of the distance of the node $u$ plus the weight that lays between the two nodes (Section 2.1.2). Results containing previously computed distances are expected by multiple threads working in parallel. This is solved by iterating the same code times proportional to the number of nodes $u \in V$. This can be described as a sequential loop that in each iteration internally conducts parallel execution. Instead of using two kernels as in [111], only one kernel with an external for loop was used. A new array stores in a unique position the distances of each node from the source node inside the subnetworks. The size of the array scales from zero, which denotes the source node, until the total number of nodes contained in the subnetwork. As the network is partitioned its scale changes. The correct definition of the distance array is needed for the normal function of the SP algorithm. Consequently, a rearrangement of the nodes in each subnetwork is needed. Figure 4.11 shows threads $t_1, t_2, t_3$ belonging to Block1 penetrating the nodes of the subnetwork. Threads are assigned only on the nodes of the component and they are responsible for processing every attached node on them. As the shortest path computation is based on the Bellman-Ford algorithm, then its relaxation step [151] is executed concurrently by multiple threads running on GPU. For instance, thread $t_1$ is responsible to compute both relaxation inequalities :

$$d[n] > d[k] + W_{kn} \tag{4.3}$$

$$d[p] > d[k] + W_{kp} \tag{4.4}$$

as node $k$ has two outgoing edges to node $n$ and $p$, where $W_{kn}$ and $W_{kp}$ are the weights between node pairs $k$ - $n$ and $k$ - $p$ respectively. The same process is happening in parallel in all the nodes of each components.

In Algorithm 6, at the beginning of the kernel an inequality statement ensures that the number of threads *id* launched by the Grid will not exceed the size of the data structure in Figure 4.12. The needed boundaries that should not be outreached, are given by multiplying the number of components $C_{num}$ with the size of each component $C_{size}$. The same strategy is followed for the individual

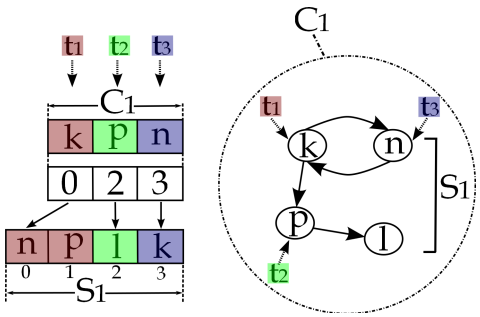



Figure 4.11: Parallel Thread blocks are processing individually multiple sub-networks. Each thread is allocated on each node and is responsible for all the connected edges.

**Algorithm 6** The CUDA specific Update function of the Computation Kernel. It shows the use of block threads for concurrent access in data.
**Input:** $\Delta_{norm}$ normalised $\Delta$ data structure.
**Output:** $S_{sp}$ Shortest Paths (SP) in each subnetwork $S$.

```
1: function KERNEL_comp( Δ_norm, S_sp )
2:     id ← threadIdx.x + blockIdx.x * blockDim.x
3:     stride ← blockDim.x * gridDim.x
4:     id_bl ← blockIdx.x
5:     id_th ← threadIdx.x
6:     while id < C_num * C_size do
7:         ⋮
8:         function DISTANCE_update( )
9:             if id_th < C_size then
10:                if U_dist[id_th + (id_bl * C_size)] <
11:                    D_dist[id_th + (id_bl * C_size)] then
12:
13:                    D_dist[id_th + (id_bl * C_size)] =
14:                        U_dist[id_th + (id_bl * C_size)]
15:        ⋮
16:        id+ = stride;
```

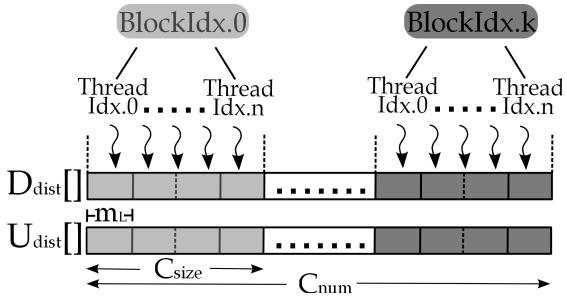



Figure 4.12: Accessing data concurrently. Each block of threads $BlockIdx.k$ is responsible for a $C_{sise}$ component. Each thread $ThreadIdx.n$ compares in parallel the distances stored in two separate arrays $Ddist[]$ and $Udist[]$. Threads should not excced the total size of the structure $C_{size}$.

threads that are accessing separately each cell of size $m_L$. The statement $(id_{bl} * C_{size})$ enclosed between the brackets of both $D_{dist}$ and $U_{dist}$ arrays in the update function $\text{Distance}_{update}(\ )$, provides parallel data access across all the components with the help of $id_{bl}$ blocks of threads. The $D_{dist}$ array stores all the computed distances while the $U_{dist}$ is used for consistency. Distances are firstly stored in the update array $U_{dist}$ and then they replace the distances in $D_{dist}$ if only they are smaller than $U_{dist}$. The addition of the thread id $id_{th}$ in both $D_{dist}$ and $U_{dist}$ expressions will force threads to access each cell individually. It is used to compare the node distances of the two arrays $D_{dist}$ and $U_{dist}$ that are stored in separate cells (Fig. 4.12). All the threads assigned to $D_{dist}$ are responsible to compare concurrently and update, if it is needed, its values.

## Recombination Kernel

The last step of the $\mu$-Layer algorithm is the composition of the independent SP results $S_{sp}$ produced by each subnetwork into a unified answer $S_{pi}$ concerning the total network. The computation of the all pairs shortest paths in the components network $G^{SCC}$ was preceded of the parallel processing process. This provided an array $D^i_{sp}[]$ for each single SSSP, in the beginning of the algorithm 3, denoting the backtracking path from a node $C_{k-1}$ to a source node $C_i$ (Fig. 4.13).

The representation of the components tracked on each SSSP are normalised back to the size of the initial graph. Each pair of components is processed con-

**Algorithm 7** Kernel responsible for combining subnetworks to provide the final answer for the APSP computation.
**Input:** $S_{sp}$ SP in each subnetwork $S$.
**Output:** Shortest paths $S_{pi}$.

```
1: function KERNEL_combin( S_sp, S_pi )
2:     S_j ← predecessor of S_i
3:     do in parallel for all S_i, S_j ∈ C_sp
4:      for all v_i ∈ S_i and u_j ∈ S_j then
5:       if v_i = u_j then
6:          if d[v_i] = s then
7:             d[S_ij] ← d[v_i] + d[u_j]
8:          elseif d[v_j] > d[v_i] then
9:             d[S_ij] ← d[v_i]
10:         end If
11:      else
12:         Add new node u_i on S_ij
13:         Store new node's u_i distance on d[S_ij]
14:      end if
15:      S_ij ← S_i ∪ S_j
16:   S_pi ← S_ij ∈ P
17:   return S_pi
```

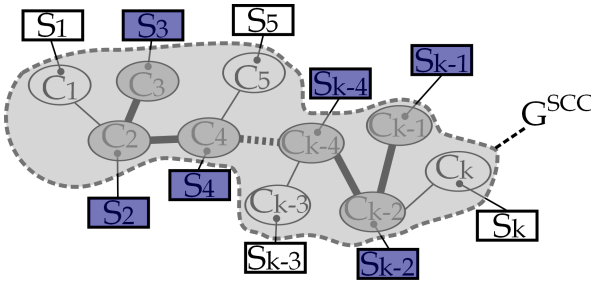



Figure 4.13: Each component $C_i$ is related with a subnetwork $S_i$ denoting its previous state. Let the SSSP from source node $C_3$ to $C_{k-1}$ contains the related subnetworks from $S_3$ to $S_{k-1}$ then the APSP distances of pairs of components are processed concurrently by the combination kernel. Similar nodes are tracked, their distances are updated to the minimum cost path and stored on a new APSP matrix. All produced APSP matrices are further compared to provide the final APSP result $S_{pi}$.

currently by multiple blocks of threads. The combination kernel is responsible to detect similar nodes between the processed pair of components $S_{ij}$ and update their distances $d[S_{ij}]$ by storing the smallest values. The product of all comparisons between pairs of components for a single SSSP is stored on a new $n \times n$ matrix, where $n$ the number of nodes in graph $G$. All steps described above are repeated for all the SSSPs in $G^{SCC}$ and their APSP matirx products are further compared to provide the final APSP result $S_{pi}$ (Algorithm 7).

## Performance Analysis

The development of the $\mu$-Layer algorithm is based on the Divide and Conquer approach. The philosophy of such an algorithm supposes that a graph problem of size $n$ is decomposed into two or more smaller subnetworks that can be processed in less time $T_s$ than the normal $T(n)$. Let $T_d(n)$ be the time needed for the decomposition and $T_c(n)$ the time to combine each of these smaller solutions to one final answer, then the total computation time is given by [152]:

$$T(n) = T_d(n) + T_s + T_c(n) \tag{4.5}$$

Based on the same logic, the $\mu$-Layer algorithm implements a parallel divide and conquer approach, where both the smaller problems and the recombination

process are implemented through a series of kernels running in parallel on multiple GPU cores.

In this study we used the $\mu$-Layer algorithm with two levels of decomposition ($\mu = 2$) and as such this version is called Double Layer Network Decomposition (DLND). DLND's execution time $T_{DLND}(n)$ is expressed as the time spent both in CPU and GPU. Due to the equation (4.5), the execution time can be expressed as:

$$T_{DLND}(n) = \overbrace{T_{\Delta}(n)}^{\text{CPU}} + \overbrace{(T_{k_{pre}}(n) + T_{k_{paral}}(n) + T_{k_{comp}}(n))}^{\text{GPU}} + \underbrace{T_{k_{combin}}(n)}_{\text{GPU}} \tag{4.6}$$

where the total time needed by the DLND is expressed through the summation of the time spent by CPU to develop the Double Layer Representation $\Delta$, plus the time needed by the three kernels ($T_{k_{pre}}(n)$, $T_{k_{paral}}(n)$,$T_{k_{comp}}(n)$ ) to process in parallel the allocated components and the kernel $T_{k_{combin}}(n)$ that combines the produced information to synthesize the all pairs shortest path.

## 4.5 Evaluation

DLND's performance has been assessed by conducting extensive tests with networks of different structure, number of nodes and edge density. We initially evaluate our DLND algorithm with real-world graphs, provided by e-Therapeutics [4]. The tested graphs represent protein-protein interactions and their size range till 3487 nodes and 57949 edges (Table 1). Futhermore, the algorithm has been evaluated by using four different network generators provided by the NetworkX platform [144]. The tested networks ranged from 512 to 8192 vertices and contain up to almost 2.5 million edges (Table 2). The experiments were conducted on a single NVIDIA GeForce GTX 560 card with 1GB memory and 7 multiprocessors (336 cuda cores) controlled by an Intel Core i5-2500 CPU processor @3.30 Ghz (4 cores) with 3.9GB RAM, running Ubuntu 12.4. Based on equation 2.5, the speedup is computed as the ratio of DLND's execution time $T_{DLND}(n)$ to serial's

running time $T_{Serial}(n)$ :

$$Speedup = \frac{T_{DLND}(n)}{T_{Serial}(n)} \tag{4.7}$$

## Results

In Figure 4.14, we evaluate DLND's performance over real-world graphs representing protein interaction networks [4]. DLND's performance is compared with a similar Bellman-Ford approach implemented with the c++ igraph platform [135] and a GPU-based approach implemented with the state-of-art LonestarGPU graph library [90]. Both parallel DLND and LonestarGPU implementations are slower than the CPU-based igraph implementation for the first small graph samples (Fig. 4.14). Overheads created due to data transfer from CPU to GPU are causing the overall degradation in performance. However, DLND begins to achieve a faster APSP computation beyond the graph sample with size of 2000 nodes. In the largest graph sample, DLND accelerates APSP computation by 1.8 times in comparison to the igraph library and 1.3 times to the GPU-based approach (Table 1). However, in order to further characterise DLND's performance and assess its capability for further acceleration, we conducted a series of tests with graphs of variable number of nodes, edges and structure.

We conducted experiments with graphs of variable number of nodes and edges. Vertices ranged from 512 to 8192 and arcs from 2K to 2.5M. Based on the edge density probability, each sample of the same distribution and set of nodes, had four different sets of edges, that gradually increased in relation to their probability that ranged from 0.01 to 0.04. The number of edges was generated based on the edge density equation [30]:

$$density(G) = \frac{m}{n(n-1)} \tag{4.8}$$

where $n$ the number of nodes and $m$ the number of edges. Density is defined as the ratio of the number of actual to possible edges. The use of different edge density corresponded to a graph with a unique number of edges. The parameters of the generating functions has been accordingly adjusted in order to produce

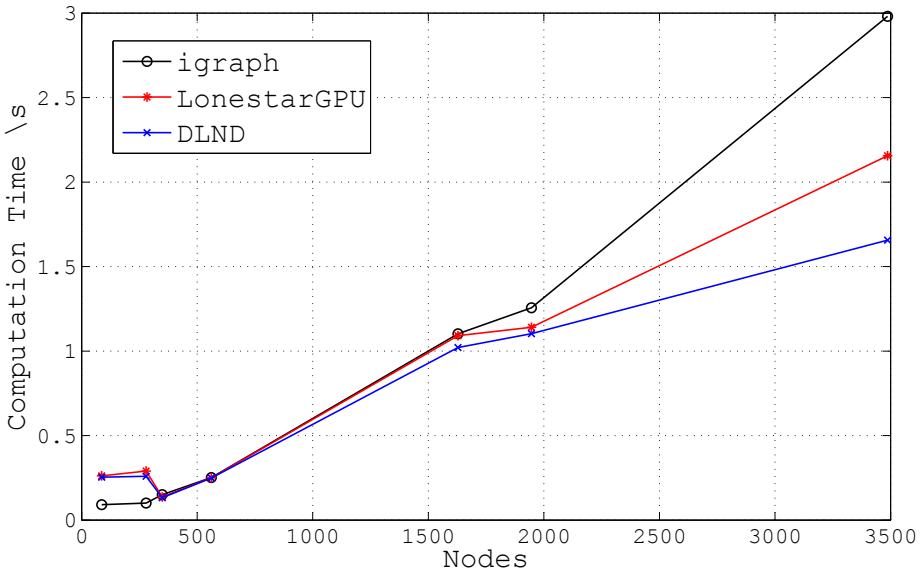



Figure 4.14: DLND's performance compared with a sequential Bellman-Ford approach (igraph) and GPU-based one (LonestarGPU), over real-world graphs.

graph samples with the right number of edges emerged by equation 4.8.

Figure 4.15 compares DLND's performance over scale-free graphs [144] where the edge density is gradually increased from $p$=0.01 to $p$=0.04. In all four figures it is clearly depicted that DLND consumes less time in contrast to the serial igraph implementation. While the edge density is getting larger DLND's execution time remains almost the same but the execution time of the serial implementation is doubled each time. DLND is 1.5 times faster than the GPU-based (LonestarGPU) approach and accelerates the shortest path computation 3.43 times more than the serial Bellman-Ford (Table 3).

The algorithm has been tested over networks with different structural characteristics. The tested graphs with small world, random, scale free and powerlaw distribution of average clustering coefficient have been generated by the NetworkX platform [144]. The Newman-Watts model [153] has been used to generate small-world graphs with high clustering coefficient and average shortest path [154]. Scale-free networks characterised by a powerlaw distribution has been generated based on the Barabasi-Albert model [146]. Such graphs tend to have few nodes with many connections and most nodes with very few link connections [116]. A

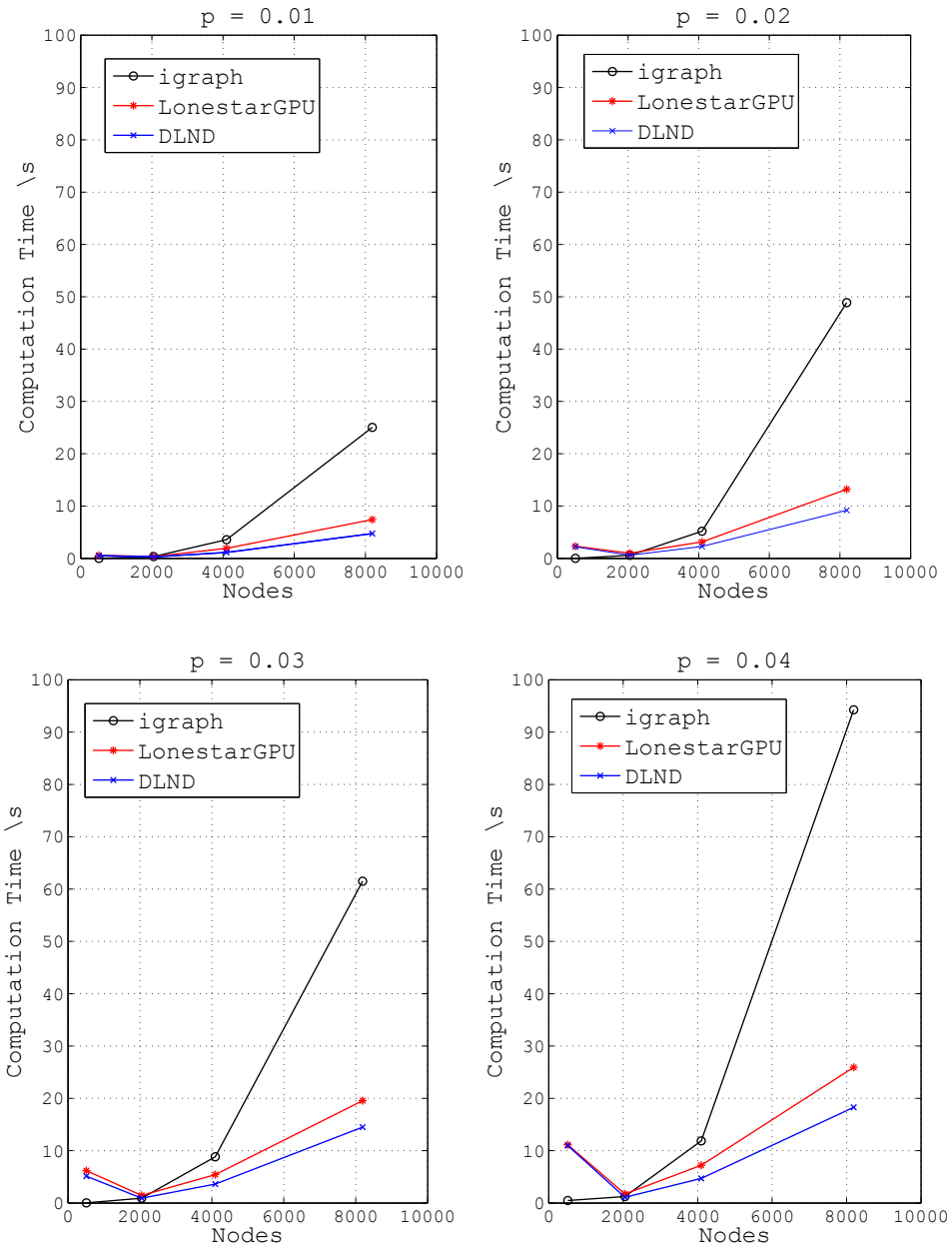



Figure 4.15: DLND $T_{DLND}(n)$ compared with igraph and LonestarGPU, over graphs of same structure and same probability of edges density ( $p$ ) but scalable number of nodes.

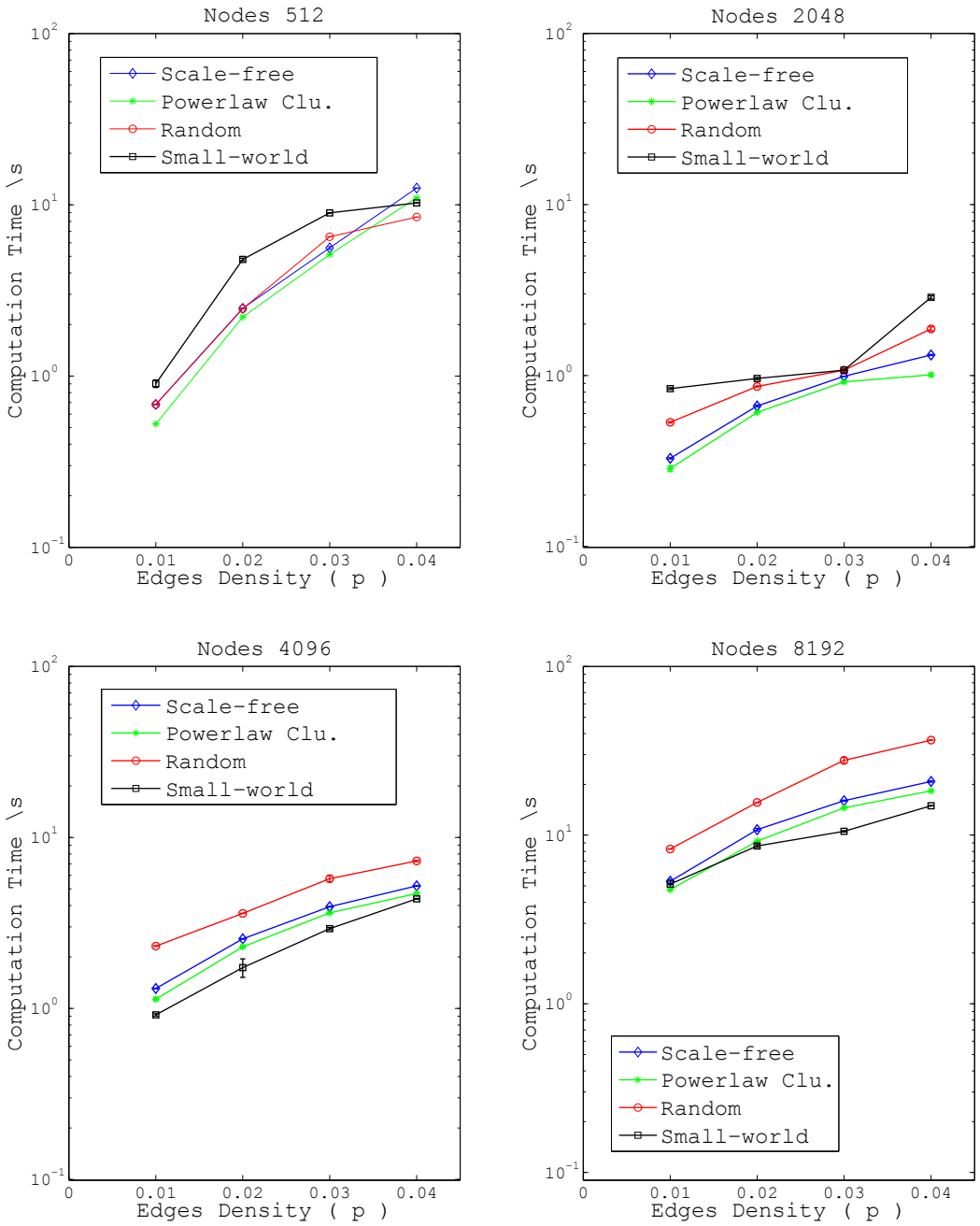



Figure 4.16: DLND execution time $T_{DLND}(n)$ over networks of different distribution with same set of nodes each time but scalable probability of edges density ( $p$ ).

similar model named as Holme-Kim grows graphs with powerlaw distribution but an approximate average clustering coefficient [155]. Random networks has been produced through the Erdos-Reyni generator [156], [157].

Figure 4.16 depicts a collection of data obtained from the execution of the DLND over networks of different structure but with the same number of nodes. The first two figures shows that better performance is achieved for powerlaw networks with average clustering. In contrast the two next figures, where the number of nodes is increased, show that DLND achieves better acceleration for networks of small-world structure. The variance of the time execution can be justified by considering the fact that networks belonging in this family are characterised by higher clustering coefficient than the others. Consequently, the larger number of components the better the mapping and distribution to blocks of threads of the GPU. Furthermore, we also tested igraph and LonestarGPU implementations over the same graph samples. The maximum speedups achieved by DLND over igraph and LonestarGPU were 5.31 and 2.56 times improvement, respectively (Table 3).

Furthermore, apart from evaluating the performance of our DLND algorithm we also verify the correctness of its results. While DLND's APSP results are provided by approximation we compared them with results produced from a common APSP approach based on the sequential Bell-man ford algorithm. In particular, we repeated their execution for 1000 times and used the average values from both algorithms in order to validate our APSP results. Our experiments showed a 3 percent error between the APSP results of our DLND and the common APSP approach. Such errors are generated in the recombination process where independent APSP results from each component are combined together to provide the final APSP answers. These results are combined based on the condensed upper layer network. However, all the connections between components in the upper layer are replaced by design with single edges. This process affects the initial connectivity of the graph where some direct paths between nodes belonging in the analysed components used to exist. Consequently, their absence from the upper layer network may create some errors in path computation.

Our experiments have shown that our DLND algorithm in comparison to similar GPU-based approaches can provide a maximum speedup of 2.56x for the

APSP computation. However, we expected higher speedups due to the enhanced processing resources that GPUs are equipped with. The embedded graph properties as well as the dependencies existing between graph data prevented a full scale parallelisation and mapping in the GPU. As a result, we applied the strategies of graph division and parallel processing. In particular, the analysed graphs are partitioned in smaller components that are concurrently processed in GPU. Each component is processed by a different block of threads that is executed concurrently in the available GPU cores. By applying this strategy we concluded that same speedups would be also achievable in a multi-core CPU. It appears that enhanced processing resources cannot offer enough acceleration in problems as the parallel graph processing due to their irregular structure and dependencies between data.

Finally, based on the observed speedups we concluded that our DLND algorithm is not so efficient for small scale graphs. However, when we increased the amount of the workload by using larger graphs then we observed better accelerations ranging from 1.5x to 2.56x. It appears that communication overheads between CPU and GPU and also the small workload of biological graphs contributed in the not so efficient acceleration. Our DLND algorithm is more suitable for larger graphs with higher edges density and higher clustering coefficient. Such graphs provide enough workload in GPU and help to overcome any additional communication overheads.

## 4.6 Summary and Conclusions

This work studies both processes of tailoring and implementing graph analysis algorithms on parallel multi-core architectures as GPUs. We proposed a novel algorithm that is based on the divide and conquer approach. The new method decomposes a graph to further abstract layers where smaller subgraphs are processed by multiple concurrent blocks of threads. The proposed method was rigorously evaluated by using real-world and artificially generated graphs as small-world, scale-free, random and an average clustering coefficient version of powerlaw distribution with variable edge density and number of nodes. Our algorithm demonstrated a 5.31x speed-up relative to a serial APSP implementation and 2.56 times

faster than a GPU-based approach.

# Chapter 5

# Linear algebra approach for parallel graph exploration on FPGAs: case study of APSP

## 5.1 Introduction

Graphs describe systems as diverse as social [129] and biological communities [22]. Traversing quickly these graphs is a challenge for both algorithms and architectures due to their size and structure. A graph traversal corresponds on the systematic exploration of all nodes and edges. The most common used algorithm for graph exploration is breadth first search (BFS) and consists a key subroutine in other graph algorithms.

Algorithms as BFS have data-driven computation dictated by the structure of the graph. Common software and hardware cannot provide efficient implementations as they mostly favour regular computations with low memory footprints and penalize fine-grained random memory accesses with poor spatial and temporal locality. Such problems lead to low compute capacity utilisation with execution times dominated by memory latency. Reconfigurable computing as field programmable gate arrays (FPGAs) can tackle this with the use of customised hardware design and software flexibility. Many studies have attempted to implement graph exploration on FPGAs. Most of them focused in memory

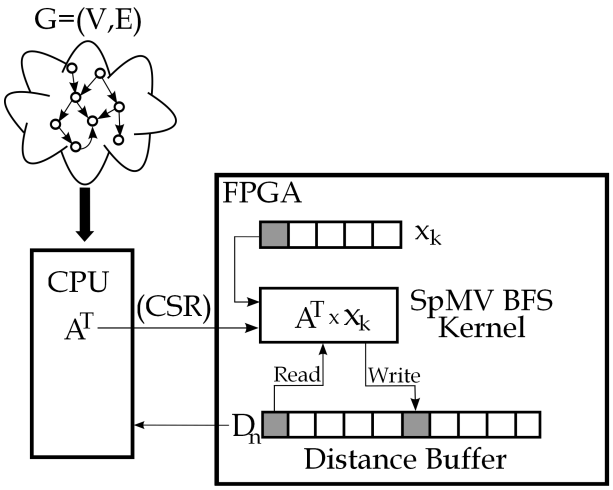



Figure 5.1: Computing BFS based on linear algebra through a fully pipeline implementation.

optimization techniques and targeted on specific FPGA platforms that optimize memory latency [101], [158].

Most parallel BFS implementations are based on the concept of reusing the common sequential BFS algorithm as a core subroutine in their parallel designs. However, this approach doesn't solve the problem of irregular data access as common serial BFS amplifies it more. The use of linear algebra has recently proved a major alternative approach in distributed memory systems [102]. Graph algorithms can be expressed as a sequence of linear algebraic operations where BFS is equivalent with the fundamental operation of matrix vector multiplication. In this Chapter, we propose using a linear algebra based implementation of BFS in FPGA. We suggest to map the BFS computation on a reconfigurable platform based on a sparse matrix vector multiplication (SpMV). In SpMV based BFS, new nodes are discovered through consecutive multiplications of the transposed graph adjacency matrix $A^T$ multiplied with an initialised vector that its values denote the source node and the ones that will be discovered (Fig. 5.1).

We propose the transformation of complex sparse graph algorithms to a sparse matrix matrix multiplication (SpMM) algorithm. This is a generic method for mapping sparse graph algorithms to FPGAs. The main property enabling embarrassing parallelism is that matrix-matrix multiply might be thought as a collection of matrix-vector multiplies with same matrix but different vectors. This approach

provides us with an embarrassingly parallel execution strategy where graph data dependencies are no more existed and can be easily implemented on dataflow implementation. As each matrix vector multiply corresponds on a BFS search this can easily provide the distance of a visited node from its source only by storing the level that it has been discovered. This process is commonly known as single source shortest path (SSSP) computation (Section 2.1.1). As a case study for our design we use the SpMV based BFS to compute the all pairs shortest paths (APSP) of a graph by simply executing multiple BFS searches from all nodes of graph. So SSSP results from individual BFS searches synthesise the APSP.

As we discussed in Section 2.2.5, there were many attempts to accelerate BFS on FPGA platforms. In this study, we build upon ideas from two previous works related with concurrent execution of multiple sequential BFS searches [81] and expression of BFS as a linear algebraic operation equivalent with sparse matrix vector multiplication (SpMV) [102]. We incorporate them into an optimized graph traversal solution that is defined as the computation of multiple concurrent sequential BFS searches implemented as SpMVs on FPGA.

In this Chapter, we present a custom hardware accelerator for parallel graph exploration that works both for dense and sparse graphs. We introduce an FPGA friendly dataflow architecture for APSP computation based on linear algebra which has a non-trivial parameter space presented in Section 5.3. We present a case study of computing all pairs shortest paths (APSP) based on concurrent BFS searches in Section 5.3.4. We provide a fully pipelined implementation that avoids any stalls in Section 5.4. Finally, we evaluate the proposed design in Section 5.5 on a range of graphs and compare its performance with well established CPU and GPU implementations.

## 5.2 Background

### Graph representation

A graph $G$ can be written as $G = (V, E)$, where $V$ is a set of $N$ vertices and $E$ is a set of $M$ directed edges. It can be represented as $N \times N$ matrix $A$ where $N =\mid V \mid$ and $A(i, j) = 1$ whenever $(i, j)$ is an edge (Section 2.1.1). Instead

of using a whole adjacency matrix $A$, graph's representation can be compressed through the Compressed Sparse Row (CSR) data structure (Section 2.3.3) that stores only the non-zero values.

## Breadth First Search (BFS)

Given a source vertex $s$, breadth-first search systematically explores the edges of $G$ to discover every vertex that is reachable from $s$. BFS forms a fundamental building block for many graph algorithms as finding maximum-flow/minimum-cut [159] and detecting community structure [160] in graphs. It plays significant role in shortest path computation as it can naturally compute the distance of a visited node from $s$ by storing the current BFS exploration level. The algorithm discovers all vertices at distance $k$ from $s$ before discovering any vertices at distance $k+1$ (Section 2.1.3). The end of BFS level $k$ consists of the barrier synchronisation before BFS proceeds in the next level $k+1$. The majority of novel parallel BFS implementations follow the general structure of the level synchronous BFS algorithm by adapting it to better fit on the underlying architecture. Algorithm 8 is a high level description of a parallel level synchronous BFS implementation. It exposes the nature of parallelism but abstracts several important details related with the needed queue data structure. Operations as push and pop that are needed while discovering new nodes and expanding frontier are not included. It mainly explains how distances of nodes are computed related with the BFS level. The distance of a new discovered node will be equivalent with $bfsLevel+1$, where $bfsLevel$ equals with zero when BFS search is on source node.

However, an alternative implementation of the level synchronous BFS can be given through linear algebraic operations. The search of each BFS level from source node $s$ is computationally equivalent to a sparse matrix - vector multiplication (SpMV) where $x_s^k$ denotes the $k^{th}$ frontier, represented as vector with integer variables. The exploration of level $k+1$ in BFS is algebraically equivalent to [102], [161] :

$$x_s^{k+1} \longleftarrow A^T \otimes x_s^k \tag{5.1}$$

**Algorithm 8** Level-synchronous BFS

**Input:** $G(V, E)$, source vertex $u_s$
**Output:** Array $distance[1..n]$ with $distance[i] = minimumdistance(u_s, u_i)$

```
1:  parallel for each node u_i ∈ V do
2:      distance[i] ← ∞
3:  end parallel for
4:  distance[s] ← 0, bfsLevel ← 0
5:  repeat
6:      done ← true
7:      parallel for each node u_i ∈ V do
8:          if (distance[i] = bfsLevel) then
9:              for (u_j adjacent to u_i) do
10:                 if (distance[j] = ∞) then
11:                     distance[j] ← bfsLevel + 1
12:                     done ← false
13:     end parallel for
14:     bfsLevel = bfsLevel + 1
15: until done == true
```

where $A^T$ corresponds on the transposed adjacency matrix and $\otimes$ implies the matrix-vector multiplication process through a special semiring that refers to abstract algebra axioms. The algorithm does not have to store the previous frontiers explicitly as multiple sparse vectors. Multiplying $x_s^k$ by $A^T$ gives nodes two steps away and so on. For instance, let $a$ be the source node of graph $G$ (Fig. 5.2) expressed through vector $x_a^1$ where $x_a^1(a) = 1$ and distance of source node $D_n[a]$ equal with zero. The multiplication of $A^T$ with $x_a^1$ picks out the nodes $c, b$. As this is the first BFS level then the distance of $D_n[b]$ and $D_n[c]$ will be equal with level $L_1$. This multiplication is repeated until all nodes of the graph are discovered. Multiplication stops when the product of vector $x_a^2$ with $A^T$ discovers the last node $d$. The result of this process is an array $D_n$ that contains the distances from a source node to all the others which is commonly known as single source shortest path (SSSP) computation [79].

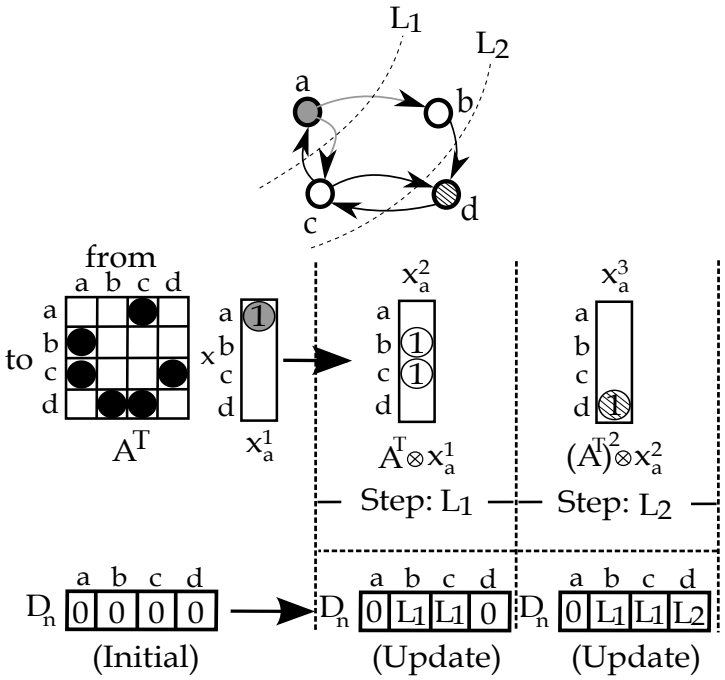



Figure 5.2: BFS based on SpMV: Each multiplication of the transposed sparse matrix $A^T$ with the vector $x_s^k$ discovers new nodes in the next BFS level. The distance of a node from the source node is equal with the current BFS level except the distance of the source node that is zero.

### BFS based algorithms: case of all pairs shortest path (APSP)

If the same procedure of BFS search is repeated for all nodes then all pairs shortest path (APSP) can be computed. The use of BFS algorithm for computing APSP is named as all-pairs breadth-first-search (AP-BFS) [53] and its complexity is $O(|V|(|E|+|V|))$. As we discussed in Section 2.1.1, APSP can also derive from repetitive SSSP computation based on algorithms as Dijkstra ($O(|E|+|V|\log|V|)$) and Bellman-Ford ($O(|V||E|)$ or from traditional APSP algorithms as Floyd-Warshal where its computational complexity is $O(|V|^3)$ [79].

In this study, we provide an APSP case study based on a BFS FPGA-based implementation. The following commutative diagram illustrates the relation between graph and linear algebra problems:

$$\begin{array}{ccccc} G & \dashrightarrow & BFS & \dashrightarrow & APSP/SSSP \\ \downarrow & & \downarrow & & \downarrow \\ A^T & \dashrightarrow & SpMV & \dashrightarrow & SpMM \end{array} \tag{5.2}$$

A graph $G$ that is represented as an adjacency matrix $A^T$ is explored by BFS implemented as sparse matrix vector multiplication (SpMV). The result of multiple SpMVs that form a sparse matrix matrix multiplication (SpMM) correspond to multiple SSSPs that APSP consists of.

## 5.3 Hardware Architecture

In this section we propose a reconfigurable computing solution for efficient graph exploration based on SpMV. Figure 5.3 shows the general structure of our design. Multiple hardware pipes equivalent to BFS threads, equipped each with independent BRAM buffer, SpMV circuit and control logic, are concurrently executing independent SSSP computations.

The logic of each BFS thread is implemented through our SSSP kernel. The adjacency matrix representing graph $G$ is stored in DRAM. All needed vectors are hosted in BRAM. Pipes share the same adjacency matrix stored in DRAM but only one vector is assigned on each of them. Matrix data are streamed contiguously and repeatedly from DRAM to BRAM. If vectors are small in size

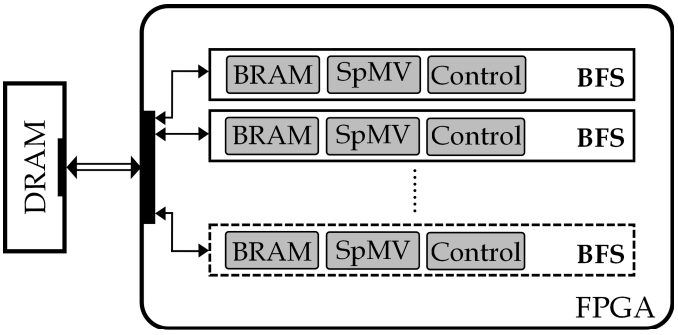



Figure 5.3: Overview of our reconfigurable graph exploration algorithm. Multiple BFS threads with independent BRAM buffers, SpMV kernel and contol logic searching graph from different source nodes.

then they are placed in BRAM. Otherwise if their size exaggerates BRAM then matrix blocking is applied. Each BRAM buffer implements a window in the vector of its pipe, stored in DRAM. In each cycle vector's next entry is transferred from DRAM and stored in BRAM. The same technique is applied to old vector entries that are sent from BRAM back to DRAM once they are not needed any more. This traffic is fully overlapped with computation. The size of graph data transferred from DRAM to BRAM is only constrained by the size of the used shifting window.

In our design, the number of hardware pipes is less than the total number of nodes in graph. Consequently, less pipes need to process more than one of the source nodes. Each pipe begins a BFS search from a new source node only after it has finished its execution with the previous one. All pipes conduct the multiplication of current entry of the global shared matrix with the entry corresponding to the same position in their local vector. This process forms the parallel execution of concurrent BFS threads where if it is repeated for all source nodes of the graph, provides as outcome the all pairs shortest paths (APSP). However, the nature of the design implies that each pipe may finish its SSSP on a moment totally different from other threads, so synchronisation strategy is ever required.

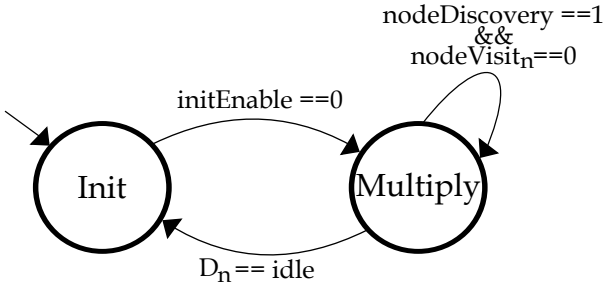



Figure 5.4: Main states of the SSSP kernel state machine: Initialisation and Multiplication.

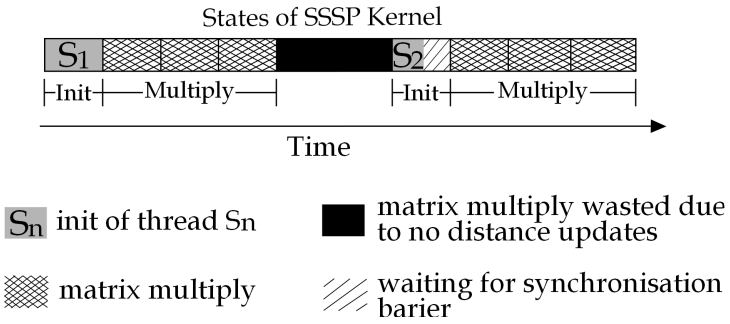



Figure 5.5: States of SSSP kernel evolving in time.

### 5.3.1 SSSP Kernel

Our SSSP kernel based on SpMV computes the single source shortest path (SSSP) from a given node to all the other nodes. Vector $D_n$ used by SpMV is stored in BRAM. It is initialised to $n^{th}$ unit vector to denote source node $S_n$ from which it will start exploring the given graph. If new nodes are discovered then data in BRAM are overwritten in place in order to include the new distances. Array $D_n$ stores the distances between source node $S_n$ and the current visited node. Its binary format is also used for next SpMVs as it points out from which nodes to continue search. The SSSP kernel also uses a binary array $nodeVisit_n$ stored in BRAM. It is updated synchronously with $D_n$ whenever a new node is discovered and is used to check if node has been already visited in the past. Algorithm 9 illustrates the SSSP computation for unweighted edges. Weighted graphs could also be supported, if additionally to $D_n$ and $nodeVisit_n$ array we add a vector of total paths to each pipe.

The mechanics of the SSSP kernel can be summarised in two main states (Fig.

5.4):

- **Initialisation:** The arrays $D_n$ and $nodeVisit_n$ are initialised with zero in the very first cycles until $initEnable$ signal, that denotes the initialisation period is over. Only $nodeVisit_n[S_n]$ is initialised with one indicating that source node $S_n$ has already been visited.

- **Multiplication:** As multiply process constantly uses vector $x_s^k$ in order to check its values then random memory access is needed. In our design this is avoided by keeping vector in BRAM. Only one nonzero entry is processed per cycle so there would be only one random memory access on vector. As there is no need to duplicate vector data in BRAM then more resources are free to be used and larger number of pipes are available to be launched.

  The product of a common matrix vector multiplication is computed by multiplying all the values of matrix row with vector values and adding them all together. In our implementation graph data are represented with CSR which means that only non zero values (nnz) are stored. These nnzs correspond to values of one so multiplication with vector $x_s^k$ is not needed as it does not alter the final result. Only addition is needed which is equivalent with the logical OR operation (Algorithm 9). The accumulation of the logical OR operation is implemented by using a feedback loop. Distance data are translated on the fly into binary format and are used as input to feedback accumulator. The result of accumulation that denotes the existence of a new node or not is written back to BRAM. Also there is a short stall stage at the end of matrix multiply which guarantees that accumulator will write correctly the data to BRAM before next state begins. In contrast no stalls exist at the matrix multiply state due to the sequential implementation of SpMV.

  If the product of accumulation $nodeDiscovery$ is equal with one and $nodeVisit_n$ is zero this means the discovery of a new node that has never been visited before in the past. In such case the distance of the discovered node is updated with the current BFS step. SSSP kernel is terminated either if the whole matrix multiply has been finished and did not discovered any new

nodes or if system counters denote that all nodes has been already discovered. This technique helps to save a wasted matrix multiply that could happen if we have already discovered all nodes at this matrix multiply. At the end of the SSSP kernel, stored distances are read and moved back to DRAM while in the same time we overwrite them with the data of $n^{th}$ column of unit matrix.

**Algorithm 9** SSSP Kernel

**Input:** $G(V, E)$, source vertex $S_n$
**Data:** Array $nodeVisit_n[1..n]$
**Special Function:** $binary(n)$, if $n > 0$ returns 1 otherwise 0
**Output:** Array $D_n[1..n]$

1: $initEnable \leftarrow 1$
2: **for** each node $u_i \in V$ **do**
3: $\quad D_n[i] \leftarrow \infty$
4: $\quad nodeVisit_n[i] \leftarrow 0$
5: $D_n[S_n] \leftarrow 0$, $nodeVisit_n[S_n] \leftarrow 1$
6: $initEnable \leftarrow 0$, $bfsStep \leftarrow 0$
7: $bin_x \leftarrow 0$
8: **if** $(initEnable == 0)$ **then**
9: $\quad$ **repeat**
10: $\quad\quad$ **for** each node $u_i \in V$ **do**
11: $\quad\quad\quad nodeDiscovery[i] \leftarrow 0$
12: $\quad\quad\quad$ **for** (all $nnz \in u_i$) **do**
13: $\quad\quad\quad\quad bin_x \leftarrow binary(D_n(i))$
14: $\quad\quad\quad\quad nodeDiscovery[i] \leftarrow nodeDiscovery[i] \parallel bin_x$
15: $\quad\quad\quad\quad$ **if** $(nodeDiscovery[i] == 1 \;\&\&\; nodeVisit_n[i] == 0)$ **then**
16: $\quad\quad\quad\quad\quad D_n[i] \leftarrow bfsStep + 1$
17: $\quad\quad\quad\quad\quad nodeVisit_n[i] \leftarrow 1$
18: $\quad\quad\quad bfsStep \leftarrow bfsStep + 1$
19: $\quad$ **until** $D_n == idle$

### 5.3.2 Parallelisation and Synchronisation

As we discussed earlier, the SSSP kernel is responsible to explore a graph from a certain source node $S_n$. Multiple instances of the same SSSP kernel are named as

BFS threads. They can execute parallel BFS searches and compute SSSPs from multiple source nodes $S_n$. Algorithm 10 describes the parallel exploration of all nodes for a given graph $G$. It takes as input the transposed graph $G^T$ and the number of used pipes $numPipes$ that denote the number of same circuit copies that can be deployed on FPGA. The 'synchronise' statement below the SSSP kernel call indicates the synchronisation of concurrent BFS threads. Synchronisation is mostly needed when a BFS thread terminate its graph exploration and it will need to start a new search from a different source node. In this case it will need to synchronise the reinitialisation of its buffers and output SSSP results on a matrix $distAll$. This matrix will host all results from all BFS threads. After the termination of their execution matrix $distAll$ will provide the all pairs shortest paths (APSP) computation.

**Algorithm 10** All-pairs shortest path: BFS SpMV-based

**Input:** $G^T$: transposed graph with CSR representation,
$numPipes$: number of concurrent pipes
**Output:** $distAll[][]$: matrix with all pairs shortest paths

1: **parallel for** each node $u \in V$ **do**
2: $Kernel_{SSSP}(G^T, numPipes)$;
3: $Syncronise()$;
4: **end parallel for**

The synchronisation of BFS threads plays a crucial role in the overall structure of our algorithm. As we discussed in Section 5.2 each new BFS search $x_s^{k+1}$ is equivalent with the multiplication of transposed matrix $A^T$ with a vector $x_s^k$. The variable $k$ is closely related with structure of the analysed graph and more precisely with the diameter of the graph that corresponds in the longest shortest path in the graph. Multiple BFS threads using the same graph can be independently terminated after various number of BFS steps. This is mostly affected by the variable $k$, so their execution needs to be precisely synchronised. For instance, let two parallel BFS threads A and B that start concurrent BFS searches from nodes $a$ and $b$ (Fig. 5.6). Both of them share the same transposed adjacency matrix $A^T$ for the multiplication with vector $x_{a,b}^k$ that corresponds on two different BFS searches. Threads A and B proceed independent and search graph until there are no more nodes left to be discovered. Thread A stops after two BFS

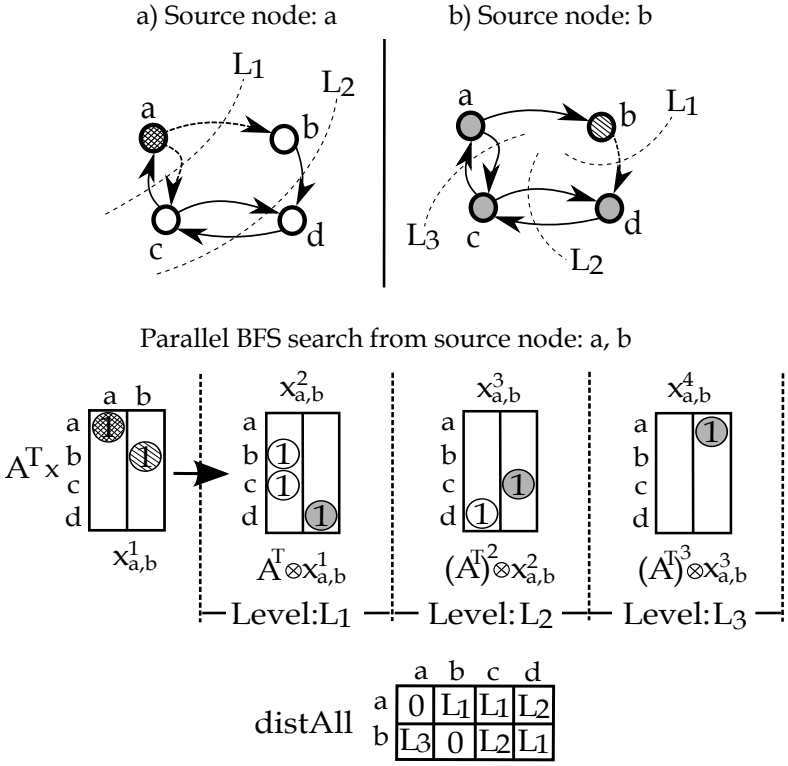



Figure 5.6: Parallel BFS threads: (a) starts BFS search from source node a and (b) starts from source node b.

step but Thread B needs one more step in order to finish its graph exploration. Here is clear that variable $k$ is different for both threads and synchronisation is needed.

All BFS threads are globally synchronous while running through the shared matrix $A^T$. The biggest challenge is the case where one BFS thread needs to run for a different source node $S_n$. It has to enter the initialisation stage in order to send data from its buffers out to DRAM and reinitialise them. There are two strategies to synchronise threads as A and B (Fig. 5.7). Both synchronisation strategies begin with the initialisation of thread A and B for the very first cycles. In order to better understand these two strategies let us take the scenario where A is finishing the matrix multiply step and thread B starts the multiplication step. Then their synchronisation can be done as:

- **Single thread-wait:** In this case, thread A can proceed in the initialisation

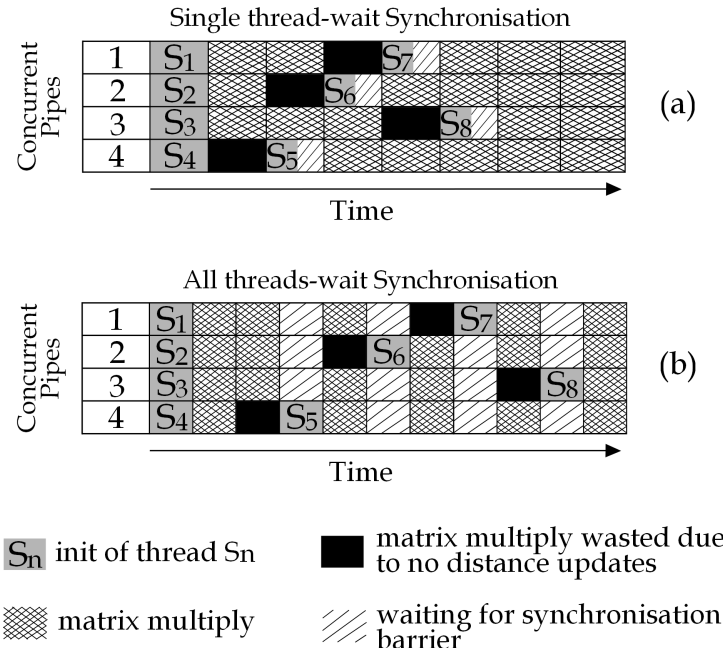



Figure 5.7: Single thread-wait (a) and all threads-wait synchronisation (b) of BFS threads.

step while thread B is progressing in matrix multiply. So matrix multiply and initialisation steps may overlap for different threads. In this case thread A needs to wait for the other threads after finishing its initialisation.

- **All threads-wait:** On the contrary, all threads have to wait for thread A to finish its initialization so that they all start doing matrix multiply in a global synchronous mode.

### 5.3.3 Design Trade-offs

Certain design trade-offs need to get into consideration while developing the APSP SpMV-based algorithm as they can determine its further performance. These trade-offs can be categorised as:

- **Synchronisation:** There are two synchronisation strategies. One of them allows threads to overlap so thread initialisation can overlap with the rest threads. Otherwise, a whole cycle needs to get spent from all threads due to the initialisation of a single thread.

- **Reading Data:** In each cycle read and write functions compete for access on the same data. Such situation can be avoided by either spending an extra cycle or by implementing an extra array as binary mask in order to process data on the same cycle. So here there is a trade-off on whether we prefer to spend more time or more memory.

- **Shifting Window:** Graph data are stored in the DRAM. Depending on the size of the shifting window data are moved on BRAM. We don't constrain the size of the used vector but the size of the shifting window. So, the trade-off here is that either we have small vectors and large number of threads or large vectors with less threads.

- **Parallelism:** The internal achievable level of parallelism is closely related with the number of used threads. The trade-off lies on the fact that we need to choose either small number of more complex processing elements (PE) or large number of PEs with less complexity.

### 5.3.4 Performance Modelling

Let $r$ be the total number of nodes in graph and $k$ the number of BFS steps. Then the total number of ticks needed to compute APSP through SpMV on FPGA can be given by:

$$ticks \quad = \quad r \times Init_{cycles} + k \times Proc_{cycles} + Sync_{cycles} \tag{5.3}$$

where $Init_{cycles}$ stands for the initialisation process, $Proc_{cycles}$ for the needed cycles to process matrix and $Sync_{cycles}$ denotes the cycles spent for synchronisation. The cycles needed for initialisation are equivalent with the total number of nodes $V$ in graph $G$ and cycles to process matrix are equal with the number of nnz values which are equal with the total number of graph edges $E$. So equation 5.3 can also be written as:

$$ticks \quad = \quad V^2 + k \times E + Sync_{cycles} \tag{5.4}$$

The cycles needed for synchronisation can vary as they are affected by the synchronisation strategy that will be chosen. Let $\tau$ be the number of threads, $nnz$ the number of non-zero values and $vecSize$ the size of the vector $x_i^j$. Also, let us take the scenario where a thread A starts a BFS search from a new source node $S_n$. If thread A overlaps its initialisation step with other threads conducting matrix multiply then it wastes $nnz - vecSize$ cycles waiting for these threads. If that is done in the same time for $\tau$ threads then $\tau \times (nnz - vecSize)$ cycles will be wasted in total. The cycles for the single thread-wait synchronisation approach can be expressed as:

$$SyncSW_{cycles} = \tau \times (nnz - vecSize) \tag{5.5}$$

If, instead, all threads wait for $\tau$ threads to complete the initialisation step then these $numPipes - \tau$ threads will waste $(numPipes - \tau) \times vecSize$ cycles, each time we potentially start matrix multiply. The cycles needed for all threads-wait synchronisation can be expressed as:

$$SyncAW_{cycles} = (numPipes - \tau) \times vecSize \tag{5.6}$$

The choice of synchronisation strategy depends both on the frequency of BFS thread restarts for new source nodes and the graph sparsity. As we discussed earlier in subsection 5.3.2 the number of steps that BFS threads can execute in the same graph can vary from thread to thread. This means that some BFS threads can stop sooner or later than others so they would have to start a new BFS search from a new source node. Consequently this affects the overall frequency of BFS thread restarts. Overheads in both synchronisation strategies are adding up due to different numbers of $\tau$ each time. Except the unevenly number of BFS threads $\tau$, graph sparsity plays a crucial role in the overall performance of the algorithm. The difference $(nnz - vectorSize)$ affects the overall synchronisation time as it is closely related with the graph sparsity. The dense the graph, the larger this factor $(nnz - vectorsize)$ becomes.

As there are two synchronisation strategies, then equation 5.4 that describes

the total number of ticks due to equation 5.5 that uses single thread-wait synchronisation will become:

$$ticks = V^2 + k \times E + \tau(E - V) \tag{5.7}$$

On the contrary, if all threads-wait synchronisation is used then the total number of ticks due to equation 5.6 is becoming:

$$ticks = V^2 + k \times E + (numPipes - \tau) \times V \tag{5.8}$$

The total time to compute APSP based on FPGA is affected by clocking FPGA at different frequency rates.

## 5.4 Implementation

The APSP SpMV-based algorithm has been implemented on a Maxeler MaxWorkstation [162] equipped with an Intel Core i7 with 16GB of RAM. It is coupled with a Virtex-6 XC6VVSX475T FPGA equipped with 24GB private dynamic RAM and 38Mb on-chip BRAM. The FPGA is linked with the CPU through a PCI bus that attains a maximum throughput of 2GB/s. Our APSP SpMV-based design operates on a maximum frequency of 150 Mhz with 128 processing elements (Table 5.1).

Our algorithm has been tested for a series of graphs representing protein-protein interaction (PPI) networks. The real-world graph data have been provided by e-Therapeutics [4]. Also, for our benchmark algorithms we used a single NVIDIA GeForce GTX 560 card (336 CUDA cores) with 1GB memory and an Intel Core i5-2500 CPU processor @3.30 Ghz (4 cores) with 3.9GB RAM, running Ubuntu 12.4. In particular, all the sequential algorithms used for our experiments have been executed on the Intel Core i5-2500 CPU processor. We decided to use this certain CPU as we utilised the same CPU processor in Chapter 4 in order to assess the performance of our GPU algorithm. The use of the same CPU in both

Chapters 4, 5 will help us to compare the performance of our GPU- and FPGA-based algorithms over the same real-world biological graphs. Further details can be found in Section 6.

Table 5.1: Device Utilisation.

| Num. of PEs | Slice LUTs | BRAMs | FFs | Max.Frequency (MHz) |
|---|---|---|---|---|
| 128 | 47212 | 1956 | 31235 | 150 |

## 5.5 Evaluation

In this section, we validate the performance of our APSP SpMV-based implementation that we describe in Section 5.3. We evaluate our algorithms with real-world graphs [4] ranging from 561 nodes till 3487 and 404 edges till 57949 (Table 4). Additionally, we use generated graphs with small world, random and scale free distribution based on the NetworkX platform [144]. The Newman-Watts model [153] has been used to generate small-world graphs with high clustering coefficient and average shortest path [154]. Scale-free networks characterised by a powerlaw distribution has been generated based on the Barabasi-Albert model [146]. Such graphs tend to have few nodes with many connections and most nodes with very few link connections [116]. Random networks has been produced through the Erdos-Reyni generator [156], [157].

The performance of the APSP SpMV-based algorithm has been compared with the igraph c++ library, in particular with an optimised approach that computes APSP on graphs without weights [135]. Additionally, we evaluate the performance of our algorithm with a similar parallel AP-BFS-based GPU implementation for APSP computation [163], currently used as a state-of-art framework for the acceleration of brain network analysis.

### Results

In Figure 5.8, we evaluate the performance of our algorithm with real-world graphs in comparison to a sequential and parallel APSP implementation. For

the first tested small size graphs, both FPGA- and GPU-based implementations have a larger execution time than the sequential igraph (Table 4). This is caused due to overheads created by data transferred between CPU and GPU,FPGA devices respectively. Both parallel implementations begin to attain acceleration on graphs with size larger of 2000 nodes. Our FPGA implementation achieves a 1.9 times faster APSP computation over the sequential igraph platform and 1.5 times faster than the GPU parallel implementation.

Furthermore, we verify our performance analysis model by using the same sample of real-world graphs. We compare the real execution time with the predicted one that is given by our performance analysis model based on equation 5.6. As we observe in Figure 5.8 the real execution time is higher than the predicted one for all the graph samples ranging from 87 nodes till 1946. Such difference in the execution time is caused due to the additional time needed for the communication between CPU and FPGA. The time needed to transfer the data from CPU to FPGA as well as the small workload leads in higher execution time. However, the two lines representing the predicted and real time execution in figure 5.8 almost merge in the graph sample of 3487 nodes. This means that our model predicts correctly the execution time of our algorithm however the high communications overheads for small graphs are not reflected in our model.

Furthermore, in order to further characterise the performance of our algorithm we conducted experiments with graphs of incremental edges density. As our design was build with a restricted size of arrays (5000 cells) this gave us the upper limit of graphs' size that we could test our algorithm. Consequently, we used a fixed number of 5000 nodes with a variable edges density (p) ranging from 0.01 to 0.06 where the largest tested graph contains 1.499.700 edges. Figure 5.10 depicts the performance of our algorithm in comparison to igraph and the GPU-based approach. While the edges density is getting increased this has a positive impact on the performance of our algorithm as it achieves a 4 times faster APSP computation in comparison to igraph and 1.9 times faster than the GPU AP-BFS implementation (Fig. 5.10, a) (Table 5).

We also tested the performance of our algorithm over graphs with variable size of nodes and fixed size of edges density (p = 0.06). All three APSP implementations have a similar behaviour as in the previous experiments of variable

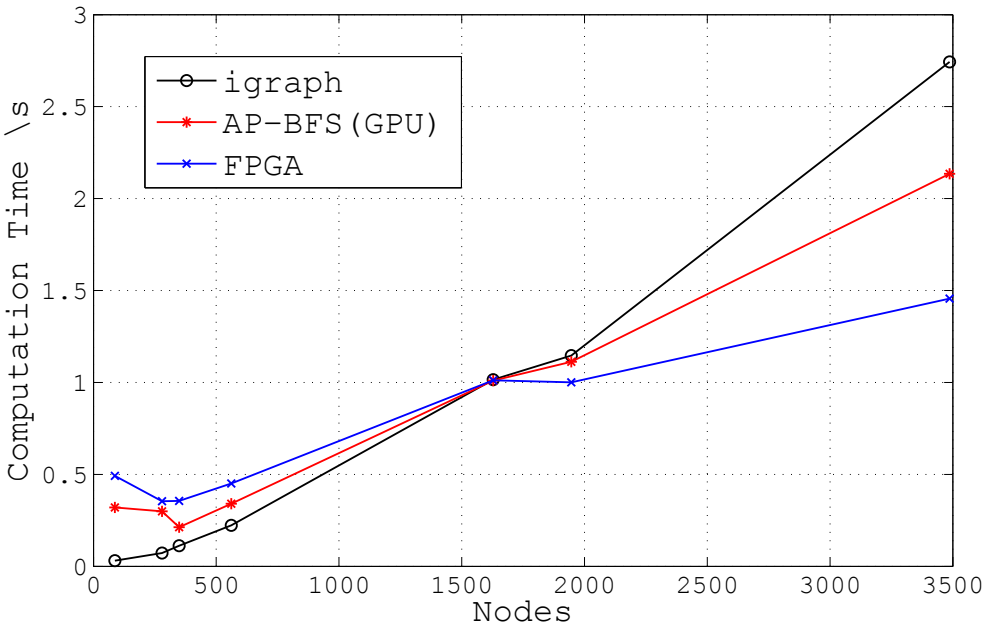



Figure 5.8: SpMV's performance compared with a CPU-based APSP approach (igraph) and GPU-based one (AP-BFS), over real-world graphs [4].

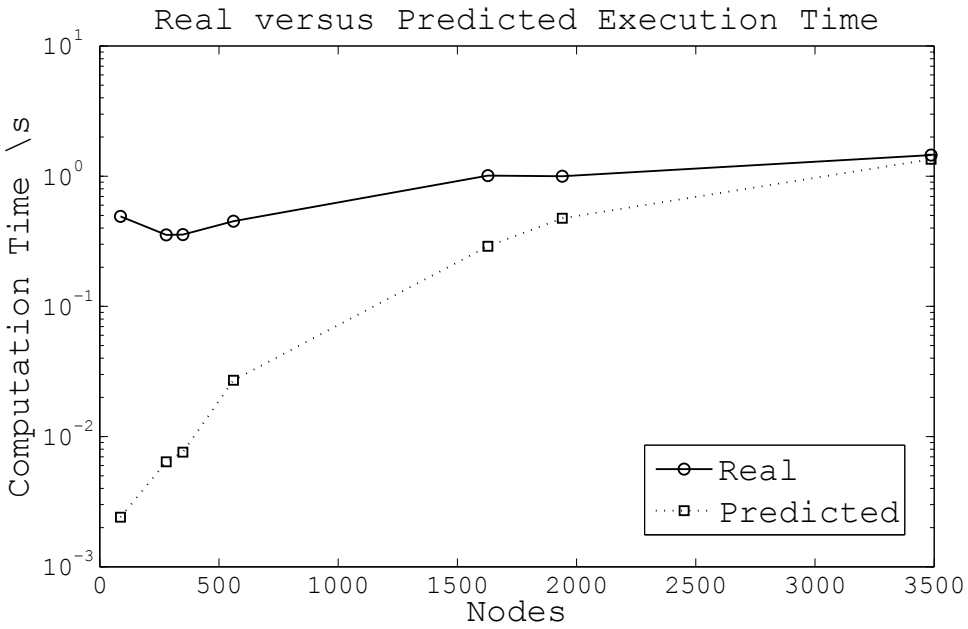



Figure 5.9: Verification of our performance model. Comparison with real execution time by using the same sample of real-world graphs.

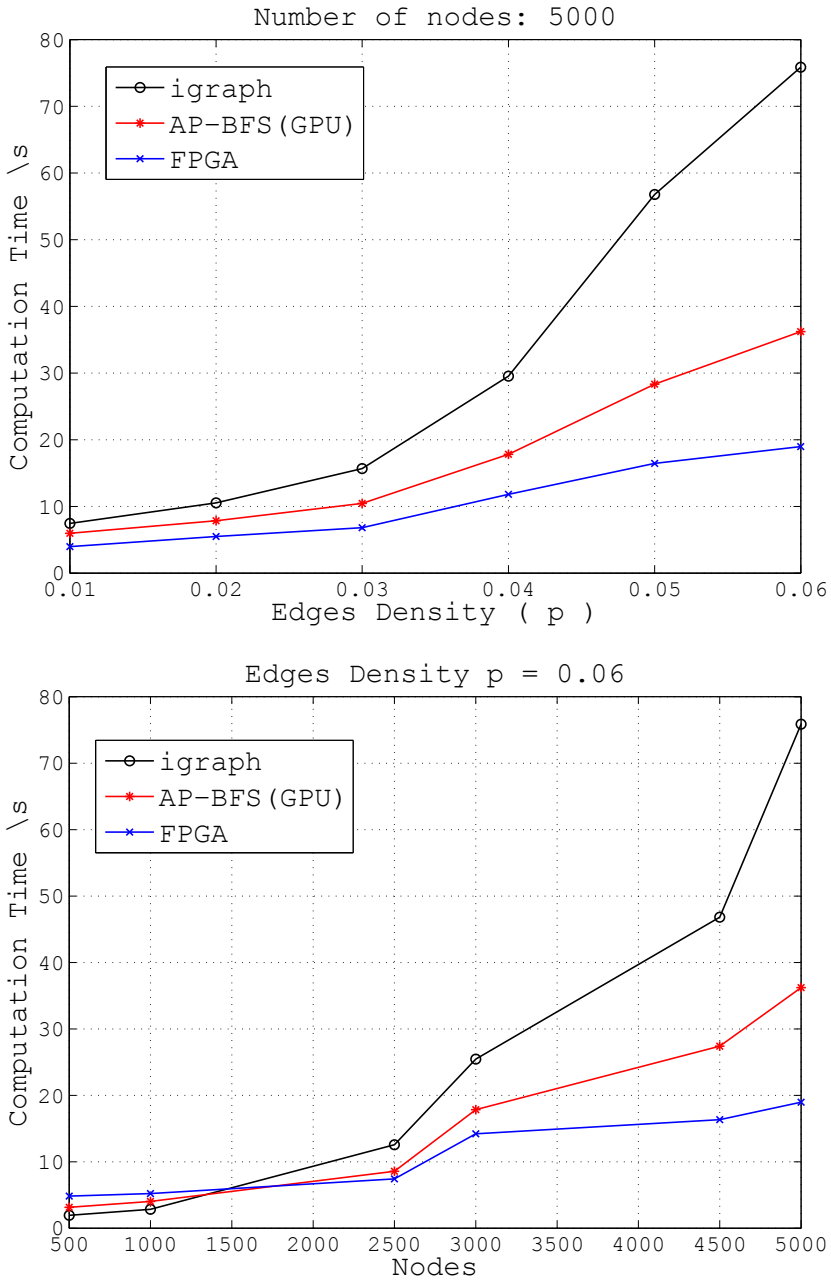



Figure 5.10: SpMV's performance compared with a CPU-based APSP approach (igraph) and GPU-based one (AP-BFS), over generated scale-free graphs [144] of: (a) variable edges density (p), (b) variable size of nodes.

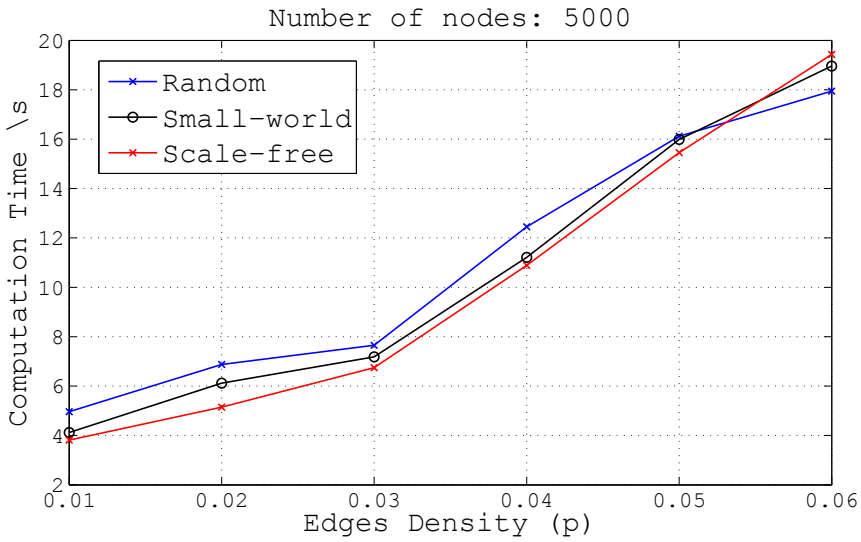



Figure 5.11: SpMV's performance over generated graphs [144] of variable structure and edges density (p = 0.06) but fixed size of nodes.

edges density (Fig. 5.10, a). However, it can be observed in Figure 5.10, b that both parallel implementations compared to igraph gain better performance beyond the graph sample of 2500 nodes. In contrast, FPGA's performance is further improved beyond the graph sample of 4500 nodes in comparison to igraph and GPU (Table 6). Furthermore, we evaluate how the performance of our algorithm reacts on graphs of variable structure as small world, random and scale-free distribution. The APSP SpMV-based algorithm is not affected by the nature of the graph (Fig. 5.11) as its implementation is based on common matrix multiplication that actually neutralises any dependencies between structure and graph algorithm as it could happen with common sequential algorithms (Table 7).

Furthermore, we observe that GPU's performance is not so efficient in such graph problems. As it is depicted in Figures 5.8 and 5.10, GPU's execution time is worse than a single CPU. This can lead us in the conclusion that such architectures may not be ideal for parallel graph algorithms. GPUs may be characterised by enhanced processing resources, however, the communication overheads between CPU and GPU can result in poor speedups. GPUs are ideal for embarrassingly parallel problems that can be partitioned in smaller independent sub-problems and processed concurrently by multiple threads. However,

graphs are synthesised by highly interconnected data that cannot be easily partitioned. As a result, we used in this Chapter the concept of linear algebra in order to overcome any dependencies between graph data. The APSP computation is transformed in independent matrix vector multiplications that can be easily executed concurrently. The same approach is also ideal for parallel architectures as the GPU platform. The use of independent sub-problems without any inter-communication between them will help to fully exploit GPU's enhanced computational capabilities.

## 5.6 Summary and Conclusions

In this Chapter, we proposed a fully pipelined implementation of the BFS algorithm based on the sparse matrix matrix multiplication (SpMM). As a case study for our design we used the SpMV based BFS to compute the all pairs shortest paths (APSP) of a graph by simply executing multiple BFS searches from all nodes of graph. We provided a FPGA friendly hardware architecture that works both for dense and sparse graphs. Our algorithm's performance has been compared with CPU- and GPU-based APSP implementations. Our FPGA implementation achieved a 1.9 times faster APSP computation over the sequential igraph platform and 1.5 times faster than the GPU parallel implementation.

# Chapter 6

# Discussion

## Network Pharmacology

Network pharmacology consists an emerging field that combines both experiments and computation for the development of new drugs by using protein-protein interaction networks (PPIs) [3]. Over the last century, the pharmaceutical industry developed drugs by identifying 'drugable' proteins that can be used for the development of compounds with desired actions against such proteins [5, 6, 7]. A great percentage of drugs function by binding to particular proteins in order modify their biochemical and biophysical operations, however, they cause side effects on a variety of other non-targeted functions [8]. In contrast, network pharmacology uses a more effective, targeted and systematic drug discovery approach:

> *"Based on advances in chemical biology and network science, network pharmacology is a distinctive new approach to drug discovery. It involves the application of network analysis to determine the set of proteins most critical in any disease, and chemical biology to identify molecules capable of targeting that set of proteins".* [4].

A pioneer of network pharmacology is E-therapeutics, a drug discovery company founded in 2001:

> *"e-Therapeutics developed a proprietary network pharmacology platform for analyzing networks of proteins associated with particular dis-*

*eases. After these analyses - which number in the millions - drug candidates with optimal impact are identified."* [164] (p. 1).

The analysis of PPIs allows the detection of specific nodes that could be used as effective targets for drug intervention [13]. Instead of focusing on particular druggable targets, it was shown that is more advantageous to target a set of proteins [14]. A promising area in network-based pharmacology is the ability to compute combinations of protein complexes, which will produce better synergistic effects when targeted together [15]. However, computing efficient protein combinations is still a bottleneck in computational biology despite many already developed computational techniques based on network centrality features [16].

Common graph analysis tools as Cytoscape [34], [35] and Nexcade [16] provide an automated mechanism for perturbation analyses on Protein to Protein Interaction networks, by using only targeted attacks based on centrality measures as outlined in 2.1. However, they can't predict what would be the best combination of nodes to remove from a graph in order to cause a higher impact. Such high-impact combinations are unlikely to be found unless they are specifically optimised or searched for [36].

Therefore, in order to optimise the removal of nodes that achieve higher impact than random and targeted attacks, we presented in Chapter 3 a novel perturbation analysis approach based on genetic algorithms. Our GA uses a population of boolean strings that represent different node removal patterns. Over time the survival of the fittest candidates favours better combinations of node removals. As this process is repeated over hundred times, it eventually converges in a well estimated combination of nodes where their removal produce higher impact.

In this context, we evaluated our GA with real-world Protein Protein Interaction networks and showed that the effectiveness of our genetic algorithm is much higher than common perturbation analysis strategies based on random and targeted node attacks. The size of such real-world PPI graphs range from hundred till 1500 nodes and 500 till 30000 edges. The ability to compute such high impact node combinations based on our GA would help researchers in network pharmacology [33] to produce better multi-target drugs for complex diseases as cancer and diabetes with better efficiency and lower toxicity. Nevertheless, the core functionality of such perturbation analysis tools as our GA is based on the

APSP computation. In particular, our GA uses the graph metric of average connected distance in order to characterise the graph robustness variation after the removal of a group of nodes. However, the computation of average connected distance is based on the shortest cost paths between all pairs of nodes. Such highly computationally expensive algorithm as APSP with the additional burden of GA's strategy to compute repeatedly new candidate solutions leads the performance of graph analysis tools in a very low level. Therefore, the use of high performance computing and parallel graph processing can enhance their performance and further accelerate the drug discovery process which is so important for million human lives.

## Parallel graph processing

The generation of molecular interaction data is getting increased exponentially [18] due to biological related technologies as mass spectrometry [165, 166] and data mining techniques [167]. Therefore, as biological data are getting increased this has as result graphs representing such data to get also bigger [17]. As the scale and complexity of graph problems is getting increased, common processing units as CPUs cannot cope with the need for high memory and computing resources. Such poor processing performance consists a major hurdle for the drug discovery process:

> *"e-Therapeutics has led the development of Network Pharmacology since its foundation. However, network analysis processed on conventional workstations is inefficient and time-consuming. International databases for active compounds and protein interactions are growing rapidly. These factors lead to a clear demand for more efficient computing methodology for the necessary networked biochemical analysis."* [168] (p. 4)

Current graph analysis packages as Pajek [134], Igraph [135] and Gephi [136] are limited by the available processing resources of commonly used workstations. Application of multi-core computing approaches could improve the efficiency of network analysis. However, such analysis pose significant challenges to parallel

processing. Non-contiguous and concurrent access to global data structures with low degree of locality are the main problems [137]. Recent progress in parallel graph algorithms addresses these challenges through innovative data structures, memory layouts, and SIMD optimizations [84, 138, 139]. However, new algorithms and implementation strategies are required for efficient processing of current generation graphs on modern multi-core architectures. Such strategies should help algorithms and their implementations benefit from the properties of the graphs. The accurate segmentation and mapping of a network onto different cores is almost impossible. However, the mapping complexity can be reduced by exploiting properties of real-world networks.

Biological networks as PPI networks are very diverse as they are generated from data belonging on diverse sources both computationally and experimentally. This makes it difficult to produce a general conclusion about their embedded properties [169]. In most cases, PPI networks [170] as all real-world graphs are characterised by the 'scale-free' and 'small-world' property. The small-world phenomenon [171] refers to a short average length from node to node while the scale-free [146] property indicates an overall power-law distribution with nodes of high degree called hubs. Discovering efficient parallel graph processing algorithms means overcoming challenges related with graph partitioning and mapping on parallel computing resources as multi-core GPUs. Small-world networks are characterised by high clustering coefficient that shows the high tendency of such graphs to be divided into clusters [169]. This property was exploited in Chapter 4, where provided the core idea of using natural graph properties to partition our graphs through strongly connected components. As connectivity reveals similarity, this helps to gather the edges of close connected nodes on the same component.

In Chapter 4, we developed an efficient parallel network analysis pattern that incorporates the natural graph properties in order to create a representation that can help in the parallel processing of smaller subgraphs. It can be potentially used to compress the graph data that are needed to be analysed. We presented a Multi Layer Network Decomposition ($\mu$-Layer) that works as a general pattern for the analysis of networks on multi-core architectures. The $\mu$-Layer algorithm takes its name from the fact that it decomposes a network, based on its strongly connected

components, into a layered organization and then uses these layers to represent that network within the multiple cores of the GPU. The current implementation takes a two layer approach named Double Layer Network Decomposition (DLND) but, in theory, the algorithm is generalizable to more layers. We introduced a new algorithmic approach whose design is driven by the multi-core architecture of the GPU platform. A graph is decomposed into smaller modules without loss of information. We showed that any graph $G = (V, E)$ can be decomposed into components that can be mapped to multiple cores. A new formula $\mu$, relates the number of blocks of threads $N_B(k)$ and the number of analysed components $C_{num}(G)$ and defines the needed level of decomposition. This is feasible due to the introduction of a novel data structure called $\Delta$, that controls the balance between the number of the cores that a multi-core processor contains and the number of components that are to be analyzed in parallel.

We evaluated our DLND algorithm through APSP computation based on a parallel approach of Bellman-Ford algorithm. We used the Bellman-Ford algorithm due to its capability to process negative weight edges that may represent the antagonism between proteins. The performance of our algorithm has been assessed with real-word PPIs ranging from 87 till 3487 nodes and 404 till 57949 edges. Our DLND algorithm accelerated APSP computation by 1.8 times in comparison to a sequential implementation and 1.3 times to a GPU-based approach. In order to further characterise our algorithm we evaluated its performance with artificially generated graphs of variable edges density, structure and number of nodes. The maximum speedups achieved by DLND over the sequential and parallel implementation were 5.31 and 2.56 times improvement, respectively.

However, as the acceleration of the GPU-based approach in Chapter 4 was ranging between 1.3 to 1.8 times improvement we hypothesized that a parallel graph algorithmic approach without any graph dependencies and pre-processing steps could create less overheads and potentially provide a better acceleration. Therefore, we developed a linear algebraic approach that transforms the APSP computation on a simple matrix-matrix multiplication. Algorithms as BFS have data-driven computation dictated by the structure of the graph. Common software and hardware cannot provide efficient implementations as they mostly favour regular computations with low memory footprints and penalize fine-grained ran-

dom memory accesses with poor spatial and temporal locality. Such problems lead to low compute capacity utilisation with execution times dominated by memory latency. Reconfigurable computing as field programmable gate arrays (FPGAs) can tackle this with the use of customised hardware design and software flexibility.

The use of linear algebra has recently proved a major alternative approach in distributed memory systems [102]. Graph algorithms can be expressed as a sequence of linear algebraic operations where BFS is equivalent with the fundamental operation of matrix vector multiplication. In Chapter 5, we proposed using a linear algebra based implementation of BFS in FPGA. We suggested to map the BFS computation on a reconfigurable platform based on a sparse matrix vector multiplication (SpMV). We proposed the transformation of complex sparse graph algorithms to a sparse matrix matrix multiplication (SpMM) algorithm. This is a generic method for mapping sparse graph algorithms to FPGAs. The main property enabling embarrassing parallelism is that matrix-matrix multiply might be thought as a collection of matrix-vector multiplies with same matrix but different vectors. This approach provided us with an embarrassingly parallel execution strategy where graph data dependencies are no more existed and could be easily implemented on dataflow implementation. The performance of our FPGA implementation was evaluated with real-world PPIs and achieved 1.9 times faster APSP computation over the sequential platform and 1.5 times improvement over the GPU parallel implementation. We also assessed the performance of our FPGA implementation over graphs of increasing edges density and variable structure. Our FPGA implementation achieved a maximum speedup of 4 times improvement over the sequential APSP implementation.

Both parallel graph processing approaches proposed in Chapters 4 and 5 contributed in the acceleration of the APSP computation. Such acceleration had a positive impact in the genetic algorithm presented in Chapter 3 as it helped to accelerate its overall execution time. When our GA was initially evaluated in Chapter 3 we observed that its execution time was high enough even with small graphs ranging from 87 to 561 nodes. In particular, the execution time ranged from several minutes to almost 9 hours (Table 3.1). Therefore the last two chapters of our thesis focused in the acceleration of the APSP computation. We utilised the same real-world graphs in order to assess and compare their per-

formance. The size of the graphs ranged between 87 nodes - 404 edges and 561 nodes - 7179 edges. The highest execution time of our GA was observed in the larger tested graph (561 nodes - 7179 edges) where 569 minutes were spent for the computation of highly critical node removals. Both parallel approaches in Chapter 4 and 5 accelerated the overall GA process by 1.8x and 1.9x respectively. In particular, GA's execution time was diminished from 569 minutes (9 hours) to 316 and 299 minutes respectively.

Such acceleration is very crucial for the overall drug discovery process. It will help to compute faster the needed groups of proteins that if targeted together can produce better drugs and save millions of lives. *"The project would enable e-Therapeutics to analyse larger and more realistic biochemical datasets and to do so much faster."* [168] (p. 14), as mentioned by e-Therapeutics. Both accelerated versions of the GA algorithm have not been yet fully utilised in e-Therapeutic's drug production line. However, small scale integration tests have shown that several days will be saved by using these two accelerated versions of our GA algorithm. This will create an impact in the day-to-day drug discovery process and eventually help in the faster development of drugs with high pharmaceutical efficiency.

# Chapter 7

# Conclusions

Recent advances in drug discovery process dictate the use of graph analysis tools that provide efficient and on-time analysis of biological graphs as PPIs. Predicting sets of proteins that if targeted together may provide better synergistic effects is an emerging field in network pharmacology. However, classic graph analysis tools cannot provide efficient predictions. Furthermore, most of the graph analysis tools suffer from low performance due to the size of the analysed graphs and use of common CPU processors. Despite the existence of architectures with many parallel execution resources, there are no any efficient parallel graph processing patterns for emerging architectures as FPGAs and GPUs. Existing strategies are not exploiting the natural graph properties and linear algebra operations to create better parallel graph processing implementations on such architectures.

This thesis investigated the potential benefits of evolutionary and high performance computing for the optimization and acceleration of graph analysis. The development of algorithms related with optimised perturabtion analysis and parallel graph processing served as the basis for this endeavour.

The remainder of this chapter recalls the main contributions of this thesis and presents open questions and directions for future research. An overall statement about the study concludes the thesis.

## 7.1 Main contributions

In Chapter 3, the development of a genetic algorithm (GA) that searches for highly optimised node removals that achieved higher impact than random and targeted attacks. It uses a population of boolean strings that represent different node removal patterns. Over time the survival of the fittest candidates favours better combinations of node removals. As this process is repeated over hundred times, it eventually converges in a well estimated combination of nodes where their removal produce a more efficient impact in graph's robustness.

In Chapter 4, a general pattern for graph analysis on multi-core GPUs by exploiting the properties of the analysed graphs has been developed. The MLND algorithm has a multi-functional character that is novel in the area of parallel network processing. It uses a data structure that can be used to control the balance between the number of the cores that a multi-core processor contains and the number of components that are going to be analyzed in parallel. At the same time this approach acts as a compressor while is able to decompose a network into smaller modules without losing information regarding its initial state and process each component concurrently in order to compute the all pairs shortest path.

In Chapter 5, the construction of a linear based graph algorithm that computes the APSP on FPGA based on a sparse matrix vector (SpMV) multiplication approach that is highly concurrent. The all pairs shortest path computation is implemented through a breadth first search (BFS) algorithm based on linear algebra. New nodes are discovered though consecutive multiplications of the transposed adjacency matrix of the graph multiplied with a vector that its values denote the source and the discovered nodes. This approach provided us with an embarrassingly parallel problem where graph data dependencies are no more existed and was implemented on dataflow computing.

## 7.2 Directions for future research

The work presented in this thesis has exploited the benefits of evolutionary and high performance computing for optimised and high performance graph analysis.

However, many questions are in need of further investigation and many opportunities for new studies have also emerged.

In Chapter 3, the evaluation of our GA algorithm not be restricted only in data produced from the biological community but also from a different field as air traffic where we can detect what are the most valuable targets for terrorist attacks. In Chapter 4, use our novel data structure for the computation of other network features as closenss and betwweness centrality on different components. In Chapter 5, use more enhanced hardware with more execution resources that will let us experiment with larger graphs and observe how our SPMV algorithms operates on such cases.

## 7.3 Conclusion

Despite the establishment of the evolutionary and high performance computing, many challenges in the field of optimised and high performance graph analysis need to be addressed. Classical perturbation analysis tools cannot compute efficient node combinations where their removal could cause a better impact in the robustness of a PPI graph. Traditional sequential graph algorithms have evolved into a major bottleneck and alternative parallel graph processing strategies need to be developed to effectively solve existing problems. This thesis has manifested the potential use of evolutionary and high performance computing in high performance and optimised graph analysis.

# Appendix A

Table 1: Results of testing igraph (Bellman-Ford), LonestarGPU(Bellman-Ford) and DLND (Bellman-Ford) on real-world graphs from e-Therapeutics [172] for APSP computation. Time in Seconds.

| Nodes | Edges | igraph [135] | LonestarGPU [90] | DLND | Speedup over igraph |
|---|---|---|---|---|---|
| 87 | 404 | 0.091 | 0.262 | 0.254 | 0.6x |
| 279 | 3399 | 0.101 | 0.290 | 0.259 | 0.38x |
| 349 | 3228 | 0.150 | 0.137 | 0.132 | 1.13x |
| 561 | 7179 | 0.251 | 0.250 | 0.249 | 1x |
| 1628 | 26703 | 1.102 | 1.091 | 1.021 | 1.08x |
| 1946 | 36596 | 1.256 | 1.141 | 1.103 | 1.51x |
| 3487 | 57949 | 2.981 | 2.156 | 1.656 | 1.8x |

Table 2: Execution times of graphs produced by NetworkX platform [173]. Time in seconds

| Algorithm | Graph Type | Probability | Number of Vertices and edges in graph 512 2-10K | 2048 40-170K | 4096 170-700K | 8192 700K-2.5M |
|---|---|---|---|---|---|---|
| **igraph** (CPU) Execution Time (Sec) | Erdos-Reyni | p=0.01 | 0.262 | 1.317 | 4.565 | 25.687 |
| | | p=0.02 | 0.322 | 1.650 | 6.172 | 53.612 |
| | | p=0.03 | 0.571 | 2.955 | 9.893 | 74.357 |
| | | p=0.04 | 0.872 | 3.283 | 13.821 | 105.825 |
| | Newman | p=0.01 | 0.015 | 0.222 | 2.54 | 21.707 |
| | | p=0.02 | 0.020 | 0.555 | 6.205 | 33.832 |
| | | p=0.03 | 0.030 | 0.840 | 7.963 | 48.135 |
| | | p=0.04 | 0.052 | 1.175 | 12.772 | 61.767 |
| | Powerlaw | p=0.01 | 0.010 | 0.340 | 3.577 | 25.032 |
| | | p=0.02 | 0.015 | 0.582 | 5.185 | 48.873 |
| | | p=0.03 | 0.030 | 0.917 | 8.851 | 61.514 |
| | | p=0.04 | 0.475 | 1.242 | 11.873 | 94.256 |
| | Barabasi | p=0.01 | 0.010 | 0.315 | 2.547 | 21.711 |
| | | p=0.02 | 0.014 | 0.635 | 5.165 | 44.482 |
| | | p=0.03 | 0.015 | 0.957 | 7.995 | 66.785 |
| | | p=0.04 | 0.025 | 1.287 | 10.72 | 85.991 |
| **LonestarGPU** (GPU) Execution Time (Sec) | Erdos-Reyni | p=0.01 | 0.689 | 0.621 | 3.121 | 11.245 |
| | | p=0.02 | 2.481 | 0.945 | 4.710 | 20.712 |
| | | p=0.03 | 6.400 | 1.521 | 6.821 | 32.513 |
| | | p=0.04 | 8.692 | 1.829 | 8.321 | 45.245 |
| | Newman | p=0.01 | 0.123 | 0.125 | 0.199 | 9.321 |
| | | p=0.02 | 4.624 | 0.457 | 4.451 | 12.213 |
| | | p=0.03 | 9.431 | 1.325 | 5.651 | 17.456 |
| | | p=0.04 | 10.165 | 3.244 | 7.121 | 21.561 |
| | Powerlaw | p=0.01 | 0.612 | 0.345 | 1.921 | 7.414 |
| | | p=0.02 | 2.314 | 0.981 | 3.145 | 13.211 |
| | | p=0.03 | 6.185 | 1.453 | 5.432 | 19.561 |
| | | p=0.04 | 11.123 | 1.721 | 7.212 | 25.931 |
| | Barabasi | p=0.01 | 0.751 | 0.521 | 1.721 | 9.456 |
| | | p=0.02 | 2.569 | 0.956 | 4.312 | 17.313 |
| | | p=0.03 | 5.731 | 1.431 | 5.021 | 23.131 |
| | | p=0.04 | 14.451 | 1.567 | 7.931 | 30.455 |
| **DLND** (CPU + GPU) Execution Time (Sec) | Erdos-Reyni | p=0.01 | 0.682 | 0.533 | 2.316 | 8.273 |
| | | p=0.02 | 2.473 | 0.863 | 3.597 | 15.608 |
| | | p=0.03 | 6.505 | 1.075 | 5.737 | 27.723 |
| | | p=0.04 | 8.477 | 1.875 | 7.288 | 36.591 |
| | Newman | p=0.01 | 0.903 | 0.839 | 0.917 | 5.123 |
| | | p=0.02 | 4.798 | 0.961 | 1.734 | 8.626 |
| | | p=0.03 | 8.967 | 1.076 | 2.928 | 10.525 |
| | | p=0.04 | 10.264 | 2.866 | 4.374 | 14.932 |
| | Powerlaw | p=0.01 | 0.525 | 0.286 | 1.133 | 4.745 |
| | | p=0.02 | 2.209 | 0.609 | 2.291 | 9.199 |
| | | p=0.03 | 5.121 | 0.918 | 3.622 | 14.507 |
| | | p=0.04 | 10.982 | 1.101 | 4.696 | 18.294 |
| | Barabasi | p=0.01 | 0.688 | 0.328 | 1.308 | 5.323 |
| | | p=0.02 | 2.484 | 0.664 | 2.554 | 10.771 |
| | | p=0.03 | 5.603 | 0.990 | 3.932 | 15.991 |
| | | p=0.04 | 12.521 | 1.319 | 5.211 | 20.783 |

Table 3: DLND speedups over LonestarGPU and igraph implementations based on graphs produced by NetworkX platform [173].

| Algorithm | Graph Type | Probability | Number of Vertices and edges in graph 512 2-10K | 2048 40-170K | 4096 170-700K | 8192 700K-2.5M |
|---|---|---|---|---|---|---|
| **DLND** (CPU + GPU) Speedup (Times) over LonestarGPU | Erdos-Reyni | p=0.01 | 1.01x | 1.16x | 1.34x | 1.35x |
| | | p=0.02 | 1.01x | 1.09x | 1.30x | 1.33x |
| | | p=0.03 | 0.98x | 1.42x | 1.18x | 1.17x |
| | | p=0.04 | 1.02x | 0.97x | 1.14x | 1.23x |
| | Newman | p=0.01 | 0.13x | 0.16x | 0.21x | 1.81x |
| | | p=0.02 | 0.96x | 0.48x | 2.56x | 1.41x |
| | | p=0.03 | 1.05x | 1.23x | 1.92x | 1.65x |
| | | p=0.04 | 0.99x | 1.13x | 1.62x | 1.44x |
| | Powerlaw | p=0.01 | 1.16x | 1.21x | 1.69x | 1.56x |
| | | p=0.02 | 1.04x | 1.62x | 1.37x | 1.43x |
| | | p=0.03 | 1.20x | 1.60x | 1.49x | 1.34x |
| | | p=0.04 | 1.01x | 1.56x | 1.53x | 1.41x |
| | Barabasi | p=0.01 | 1.09x | 1.58x | 1.31x | 1.77x |
| | | p=0.02 | 1.03x | 1.43x | 1.68x | 1.60x |
| | | p=0.03 | 1.02x | 1.45x | 1.27x | 1.44x |
| | | p=0.04 | 1.15x | 1.18x | 1.52x | 1.46x |
| **DLND** (CPU + GPU) Speedup (Times) over igraph | Erdos-Reyni | p=0.01 | 0.38x | 2.47x | 1.97x | 3.10x |
| | | p=0.02 | 0.13x | 1.91x | 1.71x | 3.43x |
| | | p=0.03 | 0.08x | 2.74x | 1.72x | 2.68x |
| | | p=0.04 | 0.10x | 1.75x | 1.89x | 2.89x |
| | Newman | p=0.01 | 0.01x | 0.26x | 2.76x | 4.23x |
| | | p=0.02 | 0.004x | 0.57x | 3.57x | 3.92x |
| | | p=0.03 | 0.003x | 0.78x | 2.71x | 4.57x |
| | | p=0.04 | 0.005x | 0.40x | 2.91x | 4.13x |
| | Powerlaw | p=0.01 | 0.019x | 1.18x | 3.15x | 5.27x |
| | | p=0.02 | 0.006x | 0.95x | 2.26x | 5.31x |
| | | p=0.03 | 0.005x | 0.99x | 2.44x | 4.24x |
| | | p=0.04 | 0.043x | 1.12x | 2.52x | 5.15x |
| | Barabasi | p=0.01 | 0.014x | 0.96x | 1.94x | 4.07x |
| | | p=0.02 | 0.005x | 0.95x | 2.02x | 4.12x |
| | | p=0.03 | 0.002x | 0.96x | 2.03x | 4.17x |
| | | p=0.04 | 0.001x | 0.97x | 2.05x | 4.13x |

Table 4: Results of testing igraph (optimised approach without weights), AP-BFS and SpMM (FPGA) on real-world graphs from e-Therapeutics [172] for APSP computation. Time in Seconds.

| Nodes | Edges | igraph [135] | GPU [163] | FPGA | Speedup over igraph |
|---|---|---|---|---|---|
| 87 | 404 | 0.031 | 0.320 | 0.492 | 0.06x |
| 279 | 3399 | 0.072 | 0.299 | 0.354 | 0.2x |
| 349 | 3228 | 0.112 | 0.213 | 0.356 | 0.3x |
| 561 | 7179 | 0.223 | 0.341 | 0.451 | 0.4x |
| 1628 | 26703 | 1.015 | 1.010 | 1.012 | 1x |
| 1946 | 36596 | 1.146 | 1.112 | 1.001 | 1.4x |
| 3487 | 57949 | 2.743 | 2.135 | 1.456 | 1.9x |

Table 5: Results of testing igraph (optimised approach without weights), AP-BFS and SpMM (FPGA) on scale-free graphs (NetworkX [144]) of variable density for APSP computation. Number of nodes : 5000. Time in Seconds.

| Density (p) | Edges | igraph [135] | GPU [163] | FPGA | Speedup over igraph |
|---|---|---|---|---|---|
| 0.01 | 249950 | 7.451 | 5.973 | 3.967 | 1.9x |
| 0.02 | 499900 | 10.531 | 7.851 | 5.495 | 1.9x |
| 0.03 | 749850 | 15.678 | 10.456 | 6.813 | 2.3x |
| 0.04 | 999800 | 29.531 | 17.821 | 11.812 | 2.5x |
| 0.05 | 1249750 | 56.781 | 28.345 | 16.453 | 3.45x |
| 0.06 | 1499700 | 75.878 | 36.215 | 18.969 | 4x |

Table 6: Results of testing igraph (optimised approach without weights), AP-BFS and SpMM (FPGA) on scale-free graphs (NetworkX [144]) of variable number of nodes for APSP computation. Edges density (p): 0.06. Time in Seconds.

| Nodes | Edges | igraph [135] | GPU [163] | FPGA | Speedup over igraph |
|---|---|---|---|---|---|
| 500 | 14940 | 1.934 | 3.145 | 4.821 | 0.4x |
| 1000 | 59940 | 2.845 | 4.018 | 5.213 | 0.5x |
| 2500 | 374850 | 12.567 | 8.567 | 7.425 | 1.7x |
| 3000 | 539820 | 25.451 | 17.834 | 14.213 | 1.8x |
| 4500 | 1214730 | 46.821 | 27.399 | 16.345 | 2.9x |
| 5000 | 1499700 | 75.878 | 36.215 | 18.969 | 4x |

Table 7: Results of testing SpMM (FPGA) on generated graphs of variable structure and edges density for APSP computation. Number of nodes: 5000. Time in Seconds.

| Density | Edges | Random [156] | Small-world [153] | Scale-free [146] |
|---|---|---|---|---|
| 0.01 | 249950 | 4.965 | 4.123 | 3.812 |
| 0.02 | 499900 | 6.875 | 6.121 | 5.145 |
| 0.03 | 749850 | 7.653 | 7.185 | 6.745 |
| 0.04 | 999800 | 12.451 | 11.213 | 10.891 |
| 0.05 | 1249750 | 16.121 | 15.989 | 15.456 |
| 0.06 | 1499700 | 17.943 | 18.961 | 19.432 |

# Bibliography


[1] Albert-Lszl Barabsi, Rka Albert, and Hawoong Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*, 281(14):69 – 77, 2000. ISSN 0378-4371. doi: http://dx.doi.org/10.1016/S0378-4371(00)00018-2. URL http://www.sciencedirect.com/science/article/pii/S0378437100000182. 1

[2] Mikail Rubinov and Olaf Sporns. Complex network measures of brain connectivity: Uses and interpretations. *NeuroImage*, 52(3):1059 – 1069, 2010. ISSN 1053-8119. doi: http://dx.doi.org/10.1016/j.neuroimage.2009.10.003. URL http://www.sciencedirect.com/science/article/pii/S105381190901074X. Computational Models of the Brain. 1

[3] Shao-hua Shi, Yue-piao Cai, Xiao-jun Cai, Xiao-yong Zheng, Dong-sheng Cao, Fa-qing Ye, and Zheng Xiang. A network pharmacology approach to understanding the mechanisms of action of traditional medicine: Bushen-huoxue formula for treatment of chronic kidney disease. *PLoS ONE*, 9(3), 03 2014. 1, 106

[4] e-therapeutics plc. URL http://etherapeutics.co.uk/. 1, 2, 46, 75, 76, 99, 100, 102, 106

[5] Andrew L Hopkins and Colin R Groom. The druggable genome. *Nature reviews Drug discovery*, 1(9):727–730, 2002. 1, 106

[6] John P Overington, Bissan Al-Lazikani, and Andrew L Hopkins. How many

drug targets are there? *Nature reviews Drug discovery*, 5(12):993–996, 2006. 1, 106

[7] Peter Imming, Christian Sinning, and Achim Meyer. Drugs, their targets and the nature and number of drug targets. *Nature reviews Drug discovery*, 5(10):821–834, 2006. 1, 106

[8] Muhammed A Yıldırım, Kwang-Il Goh, Michael E Cusick, Albert-László Barabási, and Marc Vidal. Drugtarget network. *Nature biotechnology*, 25 (10):1119–1126, 2007. 1, 106

[9] A.-L. Barabási and Z. N. Oltvai. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics*, 5(2):101–113, 2004. 2

[10] Jing-Dong J Han, Nicolas Bertin, Tong Hao, Debra S Goldberg, Gabriel F Berriz, Lan V Zhang, Denis Dupuy, Albertha JM Walhout, Michael E Cusick, Frederick P Roth, et al. Evidence for dynamically organized modularity in the yeast protein–protein interaction network. *Nature*, 430(6995): 88–93, 2004. 2

[11] Yixue Li and Luonan Chen. Big biological data: Challenges and opportunities. *Genomics, Proteomics and Bioinformatics*, 12(5):187 – 189, 2014. ISSN 1672-0229. doi: http://dx.doi.org/10.1016/j.gpb.2014.10.001. URL http://www.sciencedirect.com/science/article/pii/S1672022914001041. Special Issue: Translational Omics. 2

[12] M.E.J. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003. ISSN 0036-1445. doi: 10.1137/S003614450342480. 2

[13] DK Arrell and A Terzic. Network systems biology for drug discovery. *Clinical Pharmacology & Therapeutics*, 88(1):120–125, 2010. 2, 107

[14] T Liu and RB Altman. Identifying druggable targets by protein microenvironments matching: application to transcription factors. *CPT: pharmacometrics & systems pharmacology*, 3(1):1–9, 2014. 2, 107

[15] Seth I Berger and Ravi Iyengar. Network analyses in systems pharmacology. *Bioinformatics*, 25(19):2466–2472, 2009. 2, 107

[16] Gitanjali Yadav and Suresh Babu. Nexcade: Perturbation analysis for complex networks. *PLoS ONE*, 2012. doi: 10.1371/journal.pone.0041827. 2, 10, 37, 39, 107

[17] Shigehiko Kanaya, Md Altaf-Ul-Amin, Samuel Kuria Kiboi, and Farit Mochamad Afendi. Big data and network biology. *BioMed research international*, 2014, 2014. 2, 108

[18] Yixue Li and Luonan Chen. Big biological data: Challenges and opportunities. *Genomics, Proteomics & Bioinformatics*, 12(5):187 – 189, 2014. ISSN 1672-0229. doi: http://dx.doi.org/10.1016/j.gpb.2014.10.001. URL http://www.sciencedirect.com/science/article/pii/S1672022914001041. Special Issue: Translational Omics. 2, 108

[19] Tomasz Kajdanowicz, Przemyslaw Kazienko, and Wojciech Indyk. Parallel processing of large graphs. *Future Generation Computer Systems*, 32:324 – 337, 2014. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2013.08.007. URL http://www.sciencedirect.com/science/article/pii/S0167739X13001726. Special Section: The Management of Cloud Systems, Special Section: Cyber-Physical Society and Special Section: Special Issue on Exploiting Semantic Technologies with Particularization on Linked Data over Grid and Cloud Architectures. 2, 3

[20] A. Lumsdaine, D. Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17 (1):5–20, 2007 2007. URL http://www.sandia.gov/~bahendr/papers/graphs-and-machines.pdf. 3, 24

[21] D.H. Jones, A. Powell, C. Bouganis, and P.Y.K. Cheung. Gpu versus fpga for high productivity computing. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 119–124, Aug 2010. doi: 10.1109/FPL.2010.32. 3

[22] Guido Caldarelli. *Scale-Free Networks: Complex Webs in Nature and Technology*. Oxford University Press, 2013. URL http://EconPapers.repec.org/RePEc:oxp:obooks:9780199665174. 7, 54, 83

[23] P. Periorellis, O.C. Idowu, S.J. Lynden, M.P. Young, and P. Andras. Dealing with complex networks of process interactions: a security measure. In *Engineering Complex Computer Systems, 2004. Proceedings. Ninth IEEE International Conference on*, pages 29–36, April 2004. doi: 10.1109/ICECCS.2004.1310901. 7

[24] Albert-László Barabási and Eric Bonabeau. Scale-free networks. *Sci. Am.*, 288(5):50–59, 2003. 7

[25] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi: 10.1126/science.286.5439.509. URL http://www.sciencemag.org/content/286/5439/509.abstract. 7

[26] Oliver Mason and Mark Verwoerd. Graph theory and networks in biology. In *IET Systems Biology, 1:89 119*, pages 89–119, 2007. 7

[27] Olusola C. Idowu, Steven J. Lynden, Malcolm P. Young, and Peter Andras. Bacillus subtilis protein interaction network analysis. In *In 2004 IEEE Computational Systems Bioinformatics Conference (CSB04*, pages 623–625, 2004. 7

[28] Bjrn H. Junker and Falk Schreiber. *Analysis of Biological Networks (Wiley Series in Bioinformatics)*. Wiley-Interscience. ISBN 0470041447. 8, 11

[29] R. Albert, H. Jeong, and A.L. Barabasi. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, 2000. 8

[30] Ulrik Brandes and Thomas Erlebach. *Network analysis. Methodological foundations*. Springer, 2005. ISBN 9783540319559 3540319557 3540249796 9783540249795. 8, 76

[31] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. Computing classic closeness centrality, at scale. *CoRR*, abs/1409.0035, 2014. URL http://arxiv.org/abs/1409.0035. 8

[32] Xiaowei Zhu, Mark Gerstein, and Michael Snyder. Getting connected: analysis and principles of biological networks. *Genes development*, 21(9): 10101024, May 2007. ISSN 0890-9369. doi: 10.1101/gad.1528707. URL http://www.genesdev.org/cgi/reprint/21/9/1010. 8

[33] Andrew L Hopkins. Network pharmacology: the next paradigm in drug discovery. *Nat Chem Biol*, 4, 11 2008. 10, 107

[34] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res*, 13(11):2498–2504, 2003. 10, 37, 107

[35] Fei Li, Peng Li, Wenjian Xu, Yuxing Peng, Xiaochen Bo, and Shengqi Wang. Perturbationanalyzer: a tool for investigating the effects of concentration perturbation on protein interaction networks. *Bioinformatics*, 26 (2):275–277, 2010. 10, 37, 107

[36] Jens-Uwe Peters. Designing multi-target drugs. edited by j.richard morphy and c.john harris. *ChemMedChem*, 8(1):166–167, 2013. ISSN 1860-7187. doi: 10.1002/cmdc.201200420. URL http://dx.doi.org/10.1002/cmdc.201200420. 10, 38, 107

[37] Jianhua Zhang, Bo Song, Zhaojun Zhang, and Mingwei Zhao. Robustness analysis of the scale-free networks. {*IERI*} *Procedia*, 10(0):177 – 183, 2014. ISSN 2212-6678. doi: http://dx.doi.org/10.1016/j.ieri.2014.09.074. URL http://www.sciencedirect.com/science/article/pii/S2212667814001221. International Conference on Future Information Engineering (FIE 2014). 10

[38] Anthony H. Dekker. Simulating network robustness for critical infrastructure networks. In *Proceedings of the Twenty-eighth Australasian Conference*

*on Computer Science - Volume 38*, ACSC '05, pages 59–67, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920-68220-1. URL http://dl.acm.org/citation.cfm?id=1082161.1082168. 10

[39] H. Jeong, S. P. Mason, A. L. Barabsi, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001. doi: 10.1038/35075138. URL http://dx.doi.org/10.1038/411041a0. 11

[40] Wendy Ellens and Robert E. Kooij. Graph measures and network robustness. *CoRR*, abs/1311.5064, 2013. URL http://arxiv.org/abs/1311.5064. 11, 49

[41] Akihiro Nakaya, Susumu Goto, and Minoru Kanehisa. Extraction of correlated gene clusters by multiple graph comparison. *Genome Informatics*, 12:44–53, 2001. doi: 10.11234/gi1990.12.44. 12

[42] Niranjan Jayadevaprakash, Snehasis Mukhopadhyay, and Mathew Palakal. Generating association graphs of non-cooccurring text objects using transitive methods. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 141–145, New York, NY, USA, 2005. ACM. ISBN 1-58113-964-0. doi: 10.1145/1066677.1066713. URL http://doi.acm.org/10.1145/1066677.1066713. 12

[43] Narendra Shenoy. Retiming: Theory and practice. *Integr. VLSI J.*, 22(1-2): 1–21, August 1997. ISSN 0167-9260. doi: 10.1016/S0167-9260(97)00002-3. URL http://dx.doi.org/10.1016/S0167-9260(97)00002-3. 12

[44] Tiago Simas and Luis Mateus Rocha. Distance closures on complex networks. *CoRR*, abs/1312.2459, 2013. URL http://arxiv.org/abs/1312.2459. 13

[45] Trotter Leslie E., David Joyner, Minh Van Nguyen, and Nathann Cohen. Algorithmic graph theory and perfect graphs. *Networks*, 13(2):304–305, January 1983. ISSN 00283045. URL http://doi.wiley.com/10.1002/net.3230130214. 13

[46] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962. ISSN 0004-5411. doi: 10.1145/321105.321107. URL http://doi.acm.org/10.1145/321105.321107. 14

[47] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, January 1977. ISSN 0004-5411. doi: 10.1145/321992.321993. URL http://doi.acm.org/10.1145/321992.321993. 14

[48] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL http://dx.doi.org/10.1007/BF01386390. 14

[49] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 1848000693, 9781848000698. 14

[50] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, 2008. ISBN 3540779779. 14

[51] Erik D. Demaine, Felix Reidl, Peter Rossmanith, Fernando Sanchez Villaamil, Somnath Sikdar, and Blair D. Sullivan. Structural sparsity of complex networks: Random graph models and linear algorithms. *CoRR*, abs/1406.2587, 2014. URL http://arxiv.org/abs/1406.2587. 15

[52] Nguyen M. Joyner, D. and N. Cohen. *Algorithmic Graph Theory*. Cambridge University Press, 2006. ISBN 0521288819. 16, 17

[53] Rezaul Alam Chowdhury. *Algorithms and Data Structures for Cache-efficient Computation: Theory and Experimental Evaluation*. PhD thesis, Austin, TX, USA, 2007. AAI3274775. 17, 28, 89

[54] J. von Neumann. First draft of a report on the edvac. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993. ISSN 1058-6180. doi: 10.1109/85.238389. 18

[55] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 9780123742605. 18

[56] P. Gepner and M.F. Kowalik. Multi-core processors: New way to achieve high system performance. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pages 9–13, Sept 2006. doi: 10.1109/PARELEC.2006.54. 18

[57] D.C. Brock and G.E. Moore. *Understanding Moore's Law: Four Decades of Innovation*. Chemical Heritage Foundation, 2006. ISBN 9780941901413. URL http://books.google.co.uk/books?id=woBkE-_SOCUC. 19

[58] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.934110. 19

[59] Zhiyi Yu. Towards high-performance and energy-efficient multi-core processors. In Krzysztof Iniewski, editor, *CMOS Processors and Memories*, Analog Circuits and Signal Processing, pages 29–51. Springer Netherlands, 2010. ISBN 978-90-481-9215-1. doi: 10.1007/978-90-481-9216-8_2. URL http://dx.doi.org/10.1007/978-90-481-9216-8_2. 19

[60] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. ISBN 0123814723, 9780123814722. 19

[61] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Commun. ACM*, 51(10):50–57, October 2008. ISSN 0001-0782. doi: 10.1145/1400181.1400197. URL http://doi.acm.org/10.1145/1400181.1400197. 20

[62] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010. ISBN 9780132180139. URL https://books.google.co.uk/books?id=490mnOmTEtQC. 20

[63] A. DeHon, J. Adams, M. deLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design patterns for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004.*

*12th Annual IEEE Symposium on*, pages 13–23, April 2004. doi: 10.1109/FCCM.2004.29. 20

[64] Y. Sato, Y. Inoguchi, W. Luk, and T. Nakamura. Evaluating reconfigurable dataflow computing using the himeno benchmark. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–7, Dec 2012. doi: 10.1109/ReConFig.2012.6416746. 20

[65] M.J. Flynn, O. Pell, and O. Mencer. Dataflow supercomputing. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 1–3, Aug 2012. doi: 10.1109/FPL.2012.6339170. 20

[66] S. Kestur, J.D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 288–293, July 2010. doi: 10.1109/ISVLSI.2010.84. 20

[67] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, March 1996. ISSN 0360-0300. doi: 10.1145/234313.234345. URL http://doi.acm.org/10.1145/234313.234345. 20

[68] Cristobal A.Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, February 2014. 22, 23, 30

[69] Stan Openshaw and Ian Turton. *High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists, and Engineers*. Routledge, New York, NY, 10001, 1999. ISBN 0415156920. 22

[70] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-53086-4. 23

[71] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: Merging the plpp and opl projects. In *Proceedings of the 2010 Workshop on*

*Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0127-5. doi: 10.1145/1953611.1953620. URL http://doi.acm.org/10.1145/1953611.1953620. 23

[72] S. Akhter and J. Roberts. *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Books by engineers, for engineers. Intel Press, 2006. ISBN 9780976483243. URL http://books.google.co.uk/books?id=E9qySgAACAAJ. 23

[73] D.A. Bader. *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC Computational Science. CRC Press, 2007. ISBN 9781584889106. URL https://books.google.co.uk/books?id=dh6o5gsn1xMC. 24

[74] T. Hamada, R. Yokota, K. Nitadori, T. Narumi, K. Yasuoka, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12, Nov 2009. doi: 10.1145/1654059.1654123. 24

[75] S. Bastrakov, I. Meyerov, V. Gergel, A. Gonoskov, A. Gorshkov, E. Efimenko, M. Ivanchenko, M. Kirillin, A. Malova, G. Osipov, V. Petrov, I. Surmin, and A. Vildemanov. High performance computing in biomedical applications. *Procedia Computer Science*, 18(0):10 – 19, 2013. 24

[76] Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. *Computing in Science and Engineering*, 10:14–19, 2008. 24

[77] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999. ISBN 0387987169. 25

[78] Katsumi Inoue, Andrei Doncescu, and Hidetomo Nabeshima. Hypothesizing about causal networks with positive and negative effects by meta-level abduction. In *Inductive Logic Programming*, pages 114–129. Springer, 2011. 25

[79] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511. 25, 58, 61, 65, 87, 89

[80] All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing*, 2015. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2015.06.008. URL http://www.sciencedirect.com/science/article/pii/S0743731515001069. 26

[81] B. Betkaoui, Yu Wang, D.B. Thomas, and W. Luk. Parallel fpga-based all pairs shortest paths for sparse networks: A human brain connectome case study. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 99–104, Aug 2012. doi: 10.1109/FPL.2012.6339247. 26, 29, 30, 85

[82] Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms. *Journal of Computing in Civil Engineering*, 27(3):263–273, 2013. doi: 10.1061/(ASCE)CP.1943-5487.0000220. URL http://dx.doi.org/10.1061/(ASCE)CP.1943-5487.0000220. 26

[83] Paulius Micikevicius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '04, June 21-24, 2004, Las Vegas, Nevada, USA, Volume 3*, pages 1359–1365, 2004. 26, 29

[84] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77219-7, 978-3-540-77219-4. URL http://dl.acm.org/citation.cfm?id=1782174.1782200. 26, 29, 34, 55, 109

[85] D Eppstein and Z Galil. Parallel algorithmic techniques for combinational computation. *Annual Review of Computer Science*, 3(1):233–283, 1988.

doi: 10.1146/annurev.cs.03.060188.001313. URL http://dx.doi.org/10.1146/annurev.cs.03.060188.001313. 26, 29

[86] Aydin Buluc, John R. Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Comput.*, 36(5-6):241–253, June 2010. ISSN 0167-8191. doi: 10.1016/j.parco.2009.12.002. URL http://dx.doi.org/10.1016/j.parco.2009.12.002. 27, 29, 55

[87] T. Okuyama, F. Ino, and K. Hagihara. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, pages 284 –291, dec. 2008. doi: 10.1109/ISPA.2008.40. 27, 29

[88] Kazuya "MATSUMOTO, Naohito NAKASATO, and Stanislav G." SEDUKHIN. "blocked united algorithm for the all-pairs shortest paths problem on hybrid cpu-gpu systems". *"IEICE Trans. Inf. and Syst."*, "95" ("12"):"2759–2768", "2012". ISSN "0916-8532". 27, 29, 55

[89] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *IPDPS*, pages 349–359. IEEE Computer Society, 2014. 27, 29

[90] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76. IEEE Computer Society, 2009. ISBN 978-1-4244-4184-6. 27, 57, 76, 116

[91] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier. Efficient multi-gpu computation of all-pairs shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 360–369, May 2014. doi: 10.1109/IPDPS.2014.46. 28, 29, 55, 56

[92] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-89791-854-1. doi: 10.1145/369028.369103. URL http://dx.doi.org/10.1145/369028.369103. 28

[93] Dip Sankar Banerjee, Ashutosh Kumar, Meher Chaitanya, Shashank Sharma, and Kishore Kothapalli. Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *Journal of Parallel and Distributed Computing*, 76:81 – 93, 2015. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2014.11.006. URL http://www.sciencedirect.com/science/article/pii/S074373151400224X. Special Issue on Architecture and Algorithms for Irregular Applications. 28, 29, 55, 56

[94] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.46. URL http://dx.doi.org/10.1109/SC.2010.46. 29

[95] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88, Oct 2011. doi: 10.1109/PACT.2011.14. 29

[96] Lijuan Luo, M. Wong, and Wen-Mei Hwu. An effective gpu implementation of breadth-first search. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 52–55, June 2010. 29

[97] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6): 1543–1552, June 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.111. 29

[98] Shice Ni, Yong Dou, Dan Zou, Rongchun Li, and Qiang Wang. Parallel graph traversal for fpga. *IEICE Electronics Express*, 11(7):20130987–20130987, 2014. doi: 10.1587/elex.11.20130987. 29

[99] Qingbo Wang, Weirong Jiang, Yinglong Xia, and V. Prasanna. A message-passing multi-softcore architecture on fpga for breadth-first search. In *Field-*

*Programmable Technology (FPT), 2010 International Conference on*, pages 70–77, Dec 2010. doi: 10.1109/FPT.2010.5681757. 29

[100] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Convey hc-1 – the potential of fpgas in numerical simulation. *Preprint Series of the Engineering Mathematics and Computing Lab*, 0(07), 2010. ISSN 2191-0693. URL http://journals.ub.uni-heidelberg.de/index.php/emcl-pp/article/view/11674. 29

[101] O.G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 228–235, May 2014. doi: 10.1109/IPDPSW.2014.30. 29, 30, 84

[102] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. *CoRR*, abs/1104.4518, 2011. URL http://arxiv.org/abs/1104.4518. 29, 84, 85, 86, 111

[103] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University, December 2013. 30

[104] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107, June 2008. doi: 10.1109/SASP.2008.4570793. 31, 34

[105] R. Farber. Introduction to gpgpus and massively threaded programming. In *Bioinformatics High Performance Parallel Computer Architectures*, pages 29–48, 2010. doi: 10.1201/EBK1439814888-c2. 31

[106] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009. ISSN 0163-5964. doi:

10.1145/1555815.1555775. URL http://doi.acm.org/10.1145/1555815.1555775. 31

[107] W. Vanderbauwhede and K. Benkrid. *High-Performance Computing Using FPGAs*. SpringerLink : Bücher. Springer, 2013. ISBN 9781461417910. URL https://books.google.co.uk/books?id=uRh_AAAAQBAJ. 32

[108] Xue-Jie Zhang and Kam-Wing Ng. A review of high-level synthesis for dynamically reconfigurable fpgas. *Microprocessors and Microsystems - Embedded Hardware Design*, 24(4):199–211, 2000. 32

[109] O. Pell and V. Averbukh. Maximum performance computing with dataflow engines. *Computing in Science Engineering*, 14(4):98–103, July 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2012.78. 32

[110] K.A. Hawick, A. Leist, and D.P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, 36(12):655 – 678, 2010. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2010.07.002. URL http://www.sciencedirect.com/science/article/pii/S0167819110001055. 34

[111] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77219-7, 978-3-540-77219-4. URL http://dl.acm.org/citation.cfm?id=1782174.1782200. 34, 70

[112] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28. IEEE, 2014. 34

[113] J.L. Gustafson. Fixed time, tiered memory, and superlinear speedup. In *Distributed Memory Computing Conference*, volume 2, pages 1255–1260, Apr 1990. doi: 10.1109/DMCC.1990.556383. 36

[114] J.L. Gustafson. The consequences of fixed time performance measurement. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume ii, pages 113–124 vol.2, Jan 1992. doi: 10.1109/HICSS.1992.183285. 36

[115] Réka Albert and Albert-László Barabási. Dynamics of complex systems: Scaling laws for the period of boolean networks. *Phys. Rev. Lett.*, 84:5660–5663, Jun 2000. doi: 10.1103/PhysRevLett.84.5660. URL http://link.aps.org/doi/10.1103/PhysRevLett.84.5660. 37

[116] Steven H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, March 2001. ISSN 0028-0836. doi: 10.1038/35065725. URL http://dx.doi.org/10.1038/35065725. 37, 77, 100

[117] Marc Vidal, Michael E. Cusick, and Albert-Lszl Barabsi. Interactome networks and human disease. *Cell*, 144(6):986 – 998, 2011. ISSN 0092-8674. doi: http://dx.doi.org/10.1016/j.cell.2011.02.016. URL http://www.sciencedirect.com/science/article/pii/S0092867411001309. 37

[118] Christian M. Schneider, Andr A. Moreira, Jos S. Andrade Jr., Shlomo Havlin, and Hans J. Herrmann. Mitigation of malicious attacks on networks. *CoRR*, abs/1103.1741, 2011. 37

[119] J. C. Doyle, D. Alderson, L. Li, S. H. Low, M. Roughan, S. Shalunov, R. Tanaka, and W. Willinger. The robust yet fragile nature of the internet. volume 102(40), pages 14123–14475, 4 Oct 2005. 37

[120] Enys Mones, Nuno A. M. Arajo, Tams Vicsek, and Hans J. Herrmann. Shock waves on complex networks. *Scientific Reports*, 4(4949), 2014. URL http://arxiv.org/abs/1402.4302. 39

[121] Lance D. Chambers. *The Practical Handbook of Genetic Algorithms: Applications, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2000. ISBN 1584882409. 39

[122] D.W. Dyer. Watchmaker framework for evolutionary computation. URL http://watchmaker.uncommons.org. 43

[123] Huayang Xie, Mengjie Zhang, and Peter Andreae. An analysis of constructive crossover and selection pressure in genetic programming. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1739–1748, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-697-4. doi: 10.1145/1276958.1277297. URL http://doi.acm.org/10.1145/1276958.1277297. 43

[124] Reuven Cohen, Daniel Ben-avraham, and Shlomo Havlin. Percolation critical exponents in scale-free networks. *PHYSICAL REVIEW*, Jan 2002. 49

[125] Kumara Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Technical report, 2002. 51

[126] Dudy Lim, Yew soon Ong, and Bu sung Lee A. Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, 23, 2007. 51

[127] Matthew J. Brauer, Mark T. Holder, Laurie A. Dries, Derrick J. Zwickl, Paul O. Lewis, and David M. Hillis. Genetic algorithms and parallel processing in maximum-likelihood phylogeny inference. *Molecular Biology and Evolution*, 19(10):1717–1726, 2002. URL http://mbe.oxfordjournals.org/content/19/10/1717.abstract. 51

[128] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003. 54

[129] C.T. Butts. Social network analysis: A methodological introduction. *Asian Journal of Social Psychology*, 11(1):13–41, 2008. 54, 83

[130] Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010. ISBN 978-0-19-920665-0. 54

[131] E. Bullmore and O. Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009. 54

[132] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(1):47–97, January 2002. doi: 10.1103/RevModPhys.74.47. URL http://link.aps.org/doi/10.1103/RevModPhys.74.47. 54

[133] David Ediger, Karl Jiang, Jason Riedy, David A. Bader, Courtney Corley, Rob Farber, and William N. Reynolds. Massive social network analysis: Mining twitter for social good, 2010. 54

[134] Vladimir Batagelj, Vladimir Batagelj, Andrej Mrvar, and Andrej Mrvar. Pajek - analysis and visualization of large networks. In *Graph Drawing Software*, pages 77–103. Springer, 2003. 54, 108

[135] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. URL http://igraph.sf.net. 54, 57, 76, 100, 108, 116, 119, 120

[136] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. 2009. URL http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154. 54, 108

[137] David A. Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 547–556, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2380-3. doi: 10.1109/ICPP.2005.55. URL http://dx.doi.org/10.1109/ICPP.2005.55. 55, 109

[138] L. Munguia, D.A. Bader, and E. Ayguade. Task-based parallel breadth-first search in heterogeneous environments. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, Dec 2012. doi: 10.1109/HiPC.2012.6507474. 55, 109

[139] D.P. Scarpazza, O. Villa, and F. Petrini. Efficient breadth-first search on the cell/be processor. *Parallel and Distributed Systems, IEEE Transactions*

*on*, 19(10):1381–1395, Oct 2008. ISSN 1045-9219. doi: 10.1109/TPDS.2007.70811. 55, 109

[140] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 92:1–92:11, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503246. URL http://doi.acm.org/10.1145/2503210.2503246. 55, 57

[141] Guojing Cong and D.A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 45b–45b, April 2005. doi: 10.1109/IPDPS.2005.100. 55

[142] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, January 2002. doi: 10.1103/RevModPhys.74.47. 56

[143] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013. URL http://arxiv.org/abs/1311.3144. 56, 58

[144] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008. 57, 75, 77, 100, 103, 104, 119, 120

[145] Carsten Grabow, Stefan Grosskinsky, Jürgen Kurths, and Marc Timme. Collective relaxation dynamics of small-world networks. *Phys. Rev. E*, 91:052815, May 2015. doi: 10.1103/PhysRevE.91.052815. URL http://link.aps.org/doi/10.1103/PhysRevE.91.052815. 57

[146] Albert-Lszl Barabsi and Rka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi: 10.1126/science.286.

5439.509. URL http://www.sciencemag.org/content/286/5439/509.abstract. 57, 77, 100, 109, 120

[147] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Comput. Netw.*, 33(1-6):309–320, June 2000. ISSN 1389-1286. doi: 10.1016/S1389-1286(00)00083-9. URL http://dx.doi.org/10.1016/S1389-1286(00)00083-9. 57

[148] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 611–617, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: 10.1145/1150402.1150476. URL http://doi.acm.org/10.1145/1150402.1150476. 57

[149] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972. 58

[150] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(85)90024-9. URL http://www.sciencedirect.com/science/article/pii/0020019085900249. 58

[151] Michael J. Bannister and David Eppstein. Randomized speedup of the bellman-ford algorithm. *CoRR*, abs/1111.5414, 2011. 70

[152] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0306459701. 74

[153] M. E. J. Newman and D. J. Watts. Renormalization group analysis of the small-world network model. *Physics Letters A*, 263:341–346, 1999. 77, 100, 120

[154] Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *Circuits and Systems Magazine, IEEE*, 3(1):6–20, 2003. doi: 10.1109/MCAS.2003.1228503. 77, 100

[155] Petter Holme and Beom J. Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2):026107, 2002. doi: 10.1103/PhysRevE.65.026107. 80

[156] P. Erdos and A. Renyi. On random graphs i. *Publ. Math. Debrecen*, 6:290, 1959. 80, 100, 120

[157] E. N. Gilbert. Random graphs. *Annals of Mathematical Statistics*, 30(4): 1141–1144, 1959. 80, 100

[158] B. Betkaoui, Yu Wang, D.B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15, July 2012. doi: 10.1109/ASAP.2012.30. 84

[159] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on cuda. First Workshop on General Purpose Processing on Graphics Processing Units, 2007. 86

[160] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review*, E 69(026113), 2004. 86

[161] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011. ISBN 0898719909, 9780898719901. 86

[162] Maxeler technologies ltd. website. URL `http://www.maxeler.com/products/desktop/`. 99

[163] Yu Wang, Haixiao Du, Mingrui Xia, Ling Ren, Mo Xu, Teng Xie, Gaolang Gong, Ningyi Xu, Huazhong Yang, and Yong He. A hybrid cpu-gpu accelerated framework for fast mapping of high-resolution human brain connectome. *PLoS ONE*, 8, 05 2013. 100, 119, 120

[164] e-therapeutics leverages gridgain solution for complex and memory-intensive network pharmacology. *GridGain Systems*, 2014. URL http://www.gridgain.com/wp-content/uploads/2014/09/GridGain_Case_Study_e-Therapeutics.pdf. 107

[165] AC Gavin, M Bsche, R Krause, P Grandi, M Marzioch, A Bauer, J Schultz, JM Rick, AM Michon, CM Cruciat, M Remor, C Hfert, M Schelder, M Brajenovic, H Ruffner, A Merino, K Klein, M Hudak, D Dickson, T Rudi, V Gnau, A Bauch, S Bastuck, B Huhse, C Leutwein, MA Heurtier, RR Copley, A Edelmann, E Querfurth, V Rybin, G Drewes, M Raida, T Bouwmeester, P Bork, B Seraphin, B Kuster, G Neubauer, and G Superti-Furga. Functional organization of the yeast proteome by systematic analysis of protein complexes. *Nature*, 415(5):141–7, 2002-01-10 00:00:00.0. ISSN 0028-0836. doi: 10.1038/415141a. 108

[166] Yuen Ho, Albrecht Gruhler, Adrian Heilbut, Gary D Bader, Lynda Moore, Sally-Lin Adams, Anna Millar, Paul Taylor, Keiryn Bennett, Kelly Boutilier, et al. Systematic identification of protein complexes in saccharomyces cerevisiae by mass spectrometry. *Nature*, 415(6868):180–183, 2002. 108

[167] Alexander Nikitin, Sergei Egorov, Nikolai Daraselia, and Ilya Mazo. Pathway studiothe analysis and navigation of molecular networks. *Bioinformatics*, 19(16):2155–2157, 2003. doi: 10.1093/bioinformatics/btg290. URL http://bioinformatics.oxfordjournals.org/content/19/16/2155.abstract. 108

[168] Grant application and proposal form. *Knowledge Transfer Partnerships*, 2011. 108, 112

[169] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, Pantelis G Bagos, et al. Using graph theory to analyze biological networks. *BioData mining*, 4(10):1–27, 2011. 109

[170] Matteo Pellegrini, David Haynor, and Jason M Johnson. Protein interaction networks. *Expert Review of Proteomics*, 1(2):239–249, 2004. doi: 10.1586/14789450.1.2.239. URL http://dx.doi.org/10.1586/14789450.1.2.239. 109

[171] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998. 109

[172] Claire Ainsworth. Networking for new drugs. *Nat Med*, 17(10):1166–1168, October 2011. ISSN 1078-8956. doi: 10.1038/nm1011-1166. 116, 119

[173] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA, 2008. 117, 118