UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

HIAGO MAYK GOMES DE ARAÚJO ROCHA

Optimizing Graph Processing Execution on NUMA Machines

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Antonio Carlos S. Beck

Porto Alegre
February 2024

CIP — CATALOGING-IN-PUBLICATION

Rocha, Hiago Mayk Gomes de Araújo

Optimizing Graph Processing Execution on NUMA Machines / Hiago Mayk Gomes de Araújo Rocha. – Porto Alegre: PPGC da UFRGS, 2024.

134 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2024. Advisor: Antonio Carlos S. Beck.

1. Parallel Graph Processing. 2. Thread and Data Mapping. 3. NUMA Systems. 4. Graphs' High-Level Features. 5. Single-Source Graph Algorithms. I. Beck, Antonio Carlos S.. II. Título.

"The greatest triumphs of science are born out of the struggles and failures of countless experiments."

— Marie Curie

ACKNOWLEDGEMENTS

I dedicate this thesis to my beloved mother, Maria Marta, who has always supported me with her unwavering love and dedication. I am incredibly grateful to have such an extraordinary mother by my side.

I would like to express my deepest appreciation to my girlfriend, Luiziani Marinho, for her patience and unwavering support during the most challenging times throughout this long journey.

My sincere gratitude goes to my advisor, Prof. Dr. Antonio Carlos S. Beck, for his invaluable guidance and mentorship, not only for this work but also for shaping my career as a researcher. I am also grateful for the collaboration of Prof. Dr. Arthur Lorenzon, Dra. Janaina Schwarzrock, and Prof. Dr. Vicenç Querol. Thank you all for dedicating your time and effort to the projects we worked on together.

Lastly, I extend my thanks to my lab colleagues. I wish each of you the utmost success in your academic endeavors.

ABSTRACT

In recent years, there has been an unprecedented growth of interconnected data built on top of graph data structures, with graph applications processing large amounts of information. Graph algorithms, such as Breadth-First-Search (BFS) and PageRank (PR), directly benefit several fields, such as scientific computing, neuroscience, and social network analysis, and execute on High-Performance Computing (HPC) servers. These HPC systems are usually Non-Uniform Memory Access (NUMA) machines, where the memory access time depends on the memory location in relation to the cores, so performance strongly depends on how threads and pages (data) are mapped to different processor nodes, and the number of actives cores. Given their highly irregular communication pattern and poor data locality, graph processing are more sensitive to such alternative configurations, introducing additional challenges. Therefore, considering that the ideal thread/data mapping and number of active threads may change according to the system (e.g., microarchitecture and number of cores), graph algorithm, input graph, or even the source vertex, rightly choosing the ideal configuration is not straightforward. In this scenario, this thesis proposes new approaches to finding such near-optimal configurations for graph algorithms executing on NUMA machines. To achieve that, we leverage the unique features that characterize graph data (e.g., the number of vertices and clustering coefficient) to use in a machine learning framework. Our experimental results, considering different input graphs and algorithms executing on three NUMA machines, and comparing them with other approaches for tuning the number of threads, thread mapping, and/or data mapping, reveal the effectiveness of our proposed methods in improving algorithm execution time while significantly reducing energy consumption.

Keywords: Parallel Graph Processing. Thread and Data Mapping. NUMA Systems. Graphs' High-Level Features. Single-Source Graph Algorithms.

Otimizando a Execução de Algoritmos de Processamento de Grafos em Máquinas NUMA

RESUMO

Nos últimos anos, houve um crescimento sem precedentes de dados interconectados construídos sobre estruturas de dados de grafos, com aplicações de grafos processando grandes quantidades de informações. Algoritmos de grafos, como Breadth-First-Search (BFS) e PageRank (PR), beneficiam diretamente várias áreas, como computação científica, neurociência e análise de redes sociais, executando em servidores de Computação de Alto Desempenho (HPC). Esses sistemas de HPC geralmente são compostos por máquinas de Acesso Não Uniforme à Memória (NUMA), onde o tempo de acesso à memória depende da localização da memória em relação aos núcleos, portanto, o desempenho depende fortemente de como as threads e páginas (dados) são mapeadas para diferentes nós do processador e o número de núcleos ativos. Dada a natureza altamente irregular do padrão de comunicação e a baixa localidade dos dados, o processamento de grafos é mais sensível a essas configurações alternativas, introduzindo desafios adicionais. Portanto, considerando que o mapeamento ideal de threads/dados e o número de threads ativas podem mudar de acordo com o sistema (por exemplo, microarquitetura e número de núcleos), algoritmo de grafos, grafo de entrada ou até mesmo o vértice de origem, escolher adequadamente a configuração ideal não é uma tarefa trivial. Nesse cenário, esta tese propõe novas abordagens para encontrar configurações otimizadas para algoritmos de grafos executando em máquinas NUMA. Para alcançar isso, aproveitamos as características únicas que descrevem a estrutura dos grafos (por exemplo, o número de vértices e coeficiente de agrupamento) para usar em um framework de aprendizado de máquina. Nossos resultados experimentais, considerando diferentes grafos de entrada e algoritmos executados em três máquinas NUMA, e comparando-os com outras abordagens para ajuste do número de threads e mapeamento de threads e dados, revelam a eficácia de nossos métodos propostos em melhorar o tempo de execução do algoritmo, reduzindo significativamente o consumo de energia.

Palavras-chave: Processamento Paralelo de Grafos, Mapemanto de Threads e Dados, Sistemas NUMA, Características de Alto Nível dos Grafos, Algoritmos de Grafos de Fonte Única.

LIST OF ABBREVIATIONS AND ACRONYMS

ANN	Artificial Neural Network
BC	Betweenness Centrality
BFS	Brearth-First Search
CC	Connected Components
CFS	Completely Fair Scheduler
CSR	Compressed Sparse Row
DRAM	Dynamic Random-Access Memory
DSE	Design Space Exploration
EDP	Energy-Delay Product
GA	Genetic Algorithm
GAPBS	GAP Benchmark Suite
HPC	High-Performance Computing
LLC	Last Level Cache
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
OS	Operating System
PR	PageRank
QAP	Quadratic Assignment Problem
SSSP	Single-Source Shortest Path
TLB	Translation Lookaside Buffer
WWW	World Wide Web
NT	Number of Threads
TM	Thread Mapping
PM	Page/Data Mapping

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Graph theory provides an abstract model to represent entities and their relationship in the form of graph data structures. Many real-world problems can intuitively be cast into this format, such as social networks, business intelligence, electronic commerce, and so on (SAHU et al., 2017). Besides the broad applicability, real-world graphs are continuously growing. For example, the number of active users in the Facebook social network had grown from 1.23 billion in December 2013 (DOEKEMEIJER; VARBANESCU, 2014) to 3.063 billion in September 2023[1]. Moreover, the largest publicly-available World Wide Web (WWW) graph, extracted from a crawl in 2012, contains over 3.5 billion pages and 128.7 billion links (DHULIPALA; BLELLOCH; SHUN, 2021). Therefore, processing such large-scale graphs available today is non-trivial and has been attracting the attention of researchers from academia and industry.

In the past years, several methods, e.g., machine learning and data mining, have been proposed to extract useful information from large graphs on distributed memory or external memory systems, e.g., PEGASUS (KANG; TSOURAKAKIS; FALOUTSOS, 2009), Pregel (MALEWICZ et al., 2010), and Galois (NGUYEN; LENHARTH; PINGALI, 2013). However, thanks to the growing number of cores and memory capacity in high-performance computing (HPC) systems, input graphs like the largest publicly available mentioned above can fit into the memory of a single commodity multicore server (DHULIPALA; BLELLOCH; SHUN, 2021). Because of that, we have witnessed the development of several methods capable of quickly analyzing large graphs on shared memory machines in top-end HPC servers, e.g., Ligra (SHUN; BLELLOCH, 2013), Polymer (ZHANG; CHEN; CHEN, 2015), and GPOP (LAKHOTIA et al., 2020).

These modern HPC systems are based on Non-Uniform Memory Access (NUMA) machines, where the memory access time depends on the Dynamic Random-Access Memories (DRAM) location in relation to the cores, which are distributed across multiple nodes connected by high-speed interconnect links. Since DRAM is usually shared for all applications running in any node, NUMA increases the memory bandwidth, which implies more data being written/stored in memory simultaneously. Thus, shared-memory NUMA architectures suit well for memory-bound applications, such as graph algorithms running large input graphs, e.g., social network graphs (YAN et al., 2017).

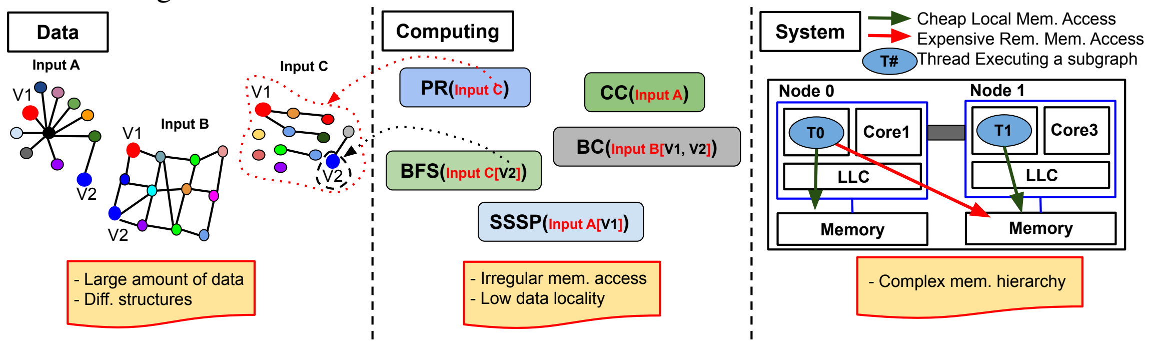
In such systems, threads running on a specific core may access memory directly in

[1]Statista - Number of monthly active Facebook users worldwide as of 4th quarter 2023 (Accessed in February 20th, 2024).

the same core's node (local access) or access other nodes' memories (remote access) via the interconnect links - with much higher latency. Thus, the application performance in a NUMA system strongly depends on which cores the threads are mapped and which data mapping policy is used (DASHTI et al., 2013; DIENER et al., 2015; SCHWARZROCK et al., 2020). Given that, one of the most effective ways to increase the performance of applications on NUMA systems is by improving the **Thread Mapping (TM)** and **Data/Page Mapping (PM)** across the available resources (cores, nodes, and memories) (DASHTI et al., 2013; DIENER et al., 2014). However, while TM and PM may mitigate some of the intrinsic NUMA penalties, e.g., costly remote memory accesses (ZHANG; CHEN; CHEN, 2015), another performance limitation remains w.r.t. the parallel applications' scalability, where the increase in the amount of resources (e.g., cores) will not proportionally deliver the same performance improvement levels. The reasons are well-known in literature (LUAN et al., 2022; SCHWARZROCK et al., 2020; MALAVE; SHINDE, 2022), and so to achieve the best efficiency in terms of hardware usage, in many times applying thread throttling, which artificially decreases the **Number of Threads (NT)**, may also be as important (SCHWARZROCK et al., 2020). On top of that, optimizing graph applications is a nontrivial task: their irregular communication pattern incurs high irregular data dependency and poor data locality, putting extra pressure on such NUMA machines (ZHANG; CHEN; CHEN, 2015). Therefore, while the strong need to efficiently process large graphs has been pushing the development of such methods for shared memory systems, the intrinsic characteristics of input graphs, algorithms, and NUMA systems challenge software developers when it comes to fully exploiting the available HPC hardware resources (LORENZON; FILHO, 2019).

Based on the discussion above, we can summarize graph processing from 3 different perspectives, as presented in Fig. 1.1: ***Data*** refers to the input graphs that come from different data sources; ***Computing*** relates to the different graph processing algorithms, which can be implemented by different graph processing frameworks (e.g., Ligra, Polymer, and Galois); and ***System*** represents different HPC machines that execute the algorithms. Fig. 1.1 also depicts the different challenges in the optimization of graph processing presented in the next section.

Figure 1.1: Graph processing in 3 different perspectives: ***Data***, the input graphs; ***Computing***, the algorithms that process the input graphs; and ***System***, the HPC machines in which the algorithms will be fired.



Source: The author.

1.1 Challenges

We have identified many challenging aspects in designing optimization strategies to execute graph applications on NUMA systems:

1. ***Input graphs' diversity and irregular structure.*** As the input data for graph analysis can be extracted from different application domains (e.g., logistics, computer network, and biology), their elements present distinct relationship patterns (topology), i.e., how vertices and edges are distributed through the graph structure. Moreover, each of them also presents an irregular structure, which causes irregular memory access with poor data locality on shared memory systems. Such diversity and irregularity are a challenge when it comes to developing an optimization strategy that must be efficient and generic. In Fig. 1.1 (***Data***), we illustrate the different input graphs, highlighting that they can represent large amounts of data and present different structures;
2. ***Graph algorithms' computation.*** Graph algorithms are widely applied as basic kernels for several practical applications: while the PageRank (PR) algorithm is the core of the Google search engine (ROGERS, 2002), the Breath-First Search (BFS) algorithm is applied in the recommendation system of Alibaba's Website (SAHU et al., 2020). With that, each graph algorithm performs a distinct computation according to its purpose. Some focus on the vertices' properties computation, e.g., Betweenness Centrality (BC) and PR, while others focus on visiting the graph's vertices/edges, e.g., BFS, Connected Components (CC), and Single-Source Shortest Path (SSSP). Although there are different strategies to implement such algorithms, e.g., vertex-/edge-/partition-centric and push-/pull-based computation (ZHANG;

CHEN; CHEN, 2015), no single strategy will deliver the best high-performance implementation of all graph algorithms. We illustrate in Fig. 1.1 (***Computing***) the different graph algorithms that a graph processing framework can implement. Notice that the algorithms receive input graphs to process. As illustrated in Fig. 1.1 (***Computing***), PR receives *Input C*, and CC receives *Input A*;

3. ***Differences in the NUMA machines.*** Even though today's NUMA processors are very fast, their performance can be affected by several factors: clock speed, cache size, main memory size, number of cores, number of NUMA nodes, and NUMA topology. On top of that, the complexity of the memory hierarchy significantly impacts the performance of memory-bound applications, like graph processing algorithms. As such, memory hierarchy can change drastically from one NUMA machine to another, and a challenge arises when designing an optimization strategy to be effective for all NUMA systems. In Fig. 1.1 (***System***), we illustrate the execution of an example application with two threads (T0 and T1) on a NUMA system comprised of 2 NUMA nodes (Node 0 and Node 1). This figure also shows the local and remote memory accesses to the system's memories, highlighting the complex memory hierarchy of the NUMA machines;
4. ***Variation in single-source algorithm executions.*** Single-source graph algorithms are the ones that require a source vertex to start processing (BEAMER; ASANOVIĆ; PATTERSON, 2015). Therefore, every time a new execution is fired (so the source vertex changes), different parts of the graph with distinct structures and amounts of vertices/edges are processed - even when using the very same graph algorithm and input data. In other words, threads will process different data, changing how they communicate and access the memory regions. Significant examples of single-source graph algorithms are BFS and SSSP, widely applied for telecom network routing, road navigation, and social network analysis (GOLDBERG; HARRELSON, 2005; BRANDES; PICH, 2007; PETERSON; DAVIE, 2007). In Fig. 1.1 (***Data*** and ***Computing***), we illustrate both (*i*) the algorithms that receive only the input graph (e.g., PR and CC) and (*ii*) the ones that receive the input graph along with a source vertex (e.g., BFS, SSSP, and BC). For example, while the PR receives the *Input C* and processes over all the vertices in each iteration, the BFS also receives the *Input C*, but it starts execution on vertex *V2* and processes only over the vertices connected to *V2*.

1.2 Optimization Opportunities

All the aspects described in the previous section impact the parallel graph processing performance because they affect how the data will be distributed throughout the NUMA system's memories, how the applications' threads will access them to communicate, and the application scalability. Therefore, optimizing graph applications by adjusting thread/data mapping and the number of threads is a non-trivial task since the best configuration of such variables may change if one of the above aspects changes. The following sections present some optimization opportunities incurred from the previously described aspects.

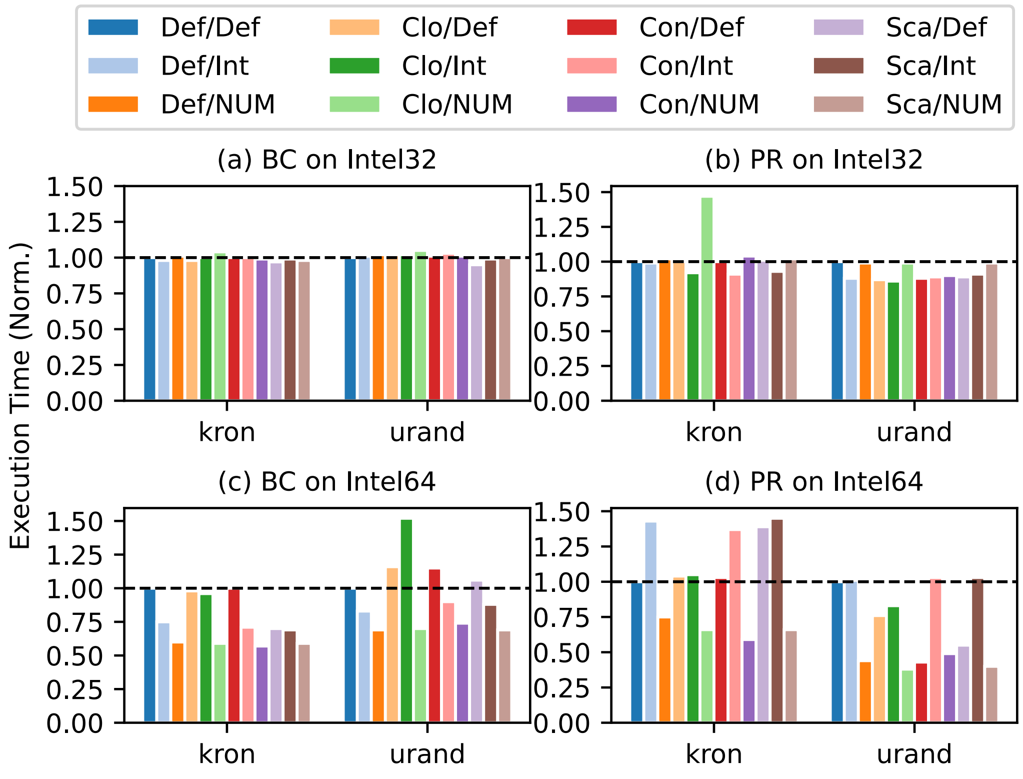
1.2.1 Impact of Thread and Data Mapping

To illustrate the impact of aspects 1-3 on the graph algorithms performance when adjusting TM and PM on NUMA systems, we show in Figure 1.2 the BC (a and c) and PR (b and d) algorithms execution time (*y*-axis) when running with different combinations of TM and PM (represented by the bars with different colors), on 2 different NUMA machines (a and b refer to *Intel32*, an Intel Xeon E5-2640v2 with 2 nodes / 16 cores / 32 threads, and c and d refer to *Intel64*, an Intel Xeon X7550 4 nodes / 32 cores / 64 threads – more details will be given later). The following configurations are evaluated for TM: Linux Operating System (OS) Default solution (**Def**), *Close* (**Clo**), *Contiguous* (**Con**), and *Scatter* (**Sca**); and for PM: First-Touch (**Def**), *Interleave* (**Int**), and *NUMA Balancing* (**NUM**) – we will also explain them in details later. Thus, each TM/PM label refers to a specific combination of the above thread and data mapping policies. We performed the experiments considering *kron* and *urand* input graphs (on the *x*-axis) (BEAMER; ASANOVIĆ; PATTERSON, 2015). We normalized the results to the **Def/Def** execution (the lower, the better).

These results highlight that *there is no unique combination of thread and data mapping policy that delivers the best performance when the* ***(1.)*** *input graph,* ***(2.)*** *algorithm, and* ***(3.)*** *NUMA machine changes*. Let us discuss some representative examples:

- For **(1.)**, when executing PR on *Intel32*, while using Con/Int configuration delivers the best performance on *kron* input graph (9% improvement), if one changes the input graph to *urand*, the best configuration is Clo/Int with 14% improvement;

Figure 1.2: BC and PR running the kron and urand on *Intel32* (Intel Xeon E5-2640v2 with 2 nodes / 16 cores / 32 threads) and on *Intel64* (Intel Xeon X7550 with 4 nodes / 32 cores / 64 threads). **Legend:** the TM/PM refers to the combination of the thread and data mapping policies.



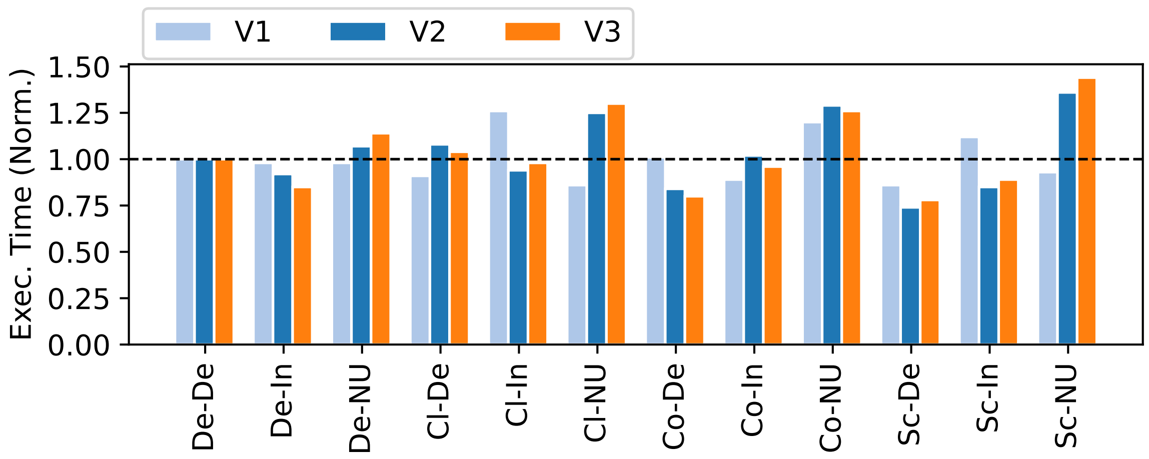
Source: The author.

- For **(2.)**, still considering the urand input graph on the same machine, if one changes the algorithm to BC, the best configuration is Sca/Def delivering 5% improvement;
- For **(3.)**, considering again the PR-urand, when it executes on *Intel64* the best configuration is Clo/NUM with 64%, which differs from the Clo/Int configuration mentioned earlier for *Intel32*.

1.2.2 Impact of Changing the Source Vertices

Let us cover the challenge **(4.)** by highlighting the performance variation when executing single-source graph algorithms from different source vertices. For that, we show in Figure 1.3, the execution time (y-axis) in the execution of the BFS on web graph starting from 3 different source vertices (V1, V2, and V3 – bars with different colors), on *Intel32* machine. We considered all the previously described TM and PM combinations in the x-axis, and we normalized to the Linux's default (De-De), represented by the black

Figure 1.3: BFS algorithm execution when it starts from 3 different source vertices (V1, V2, and V3) on the *Intel32* system. **Legend:** the TM-PM refers to the combination of the thread and data mapping policies.



Source: The author.

line). Hence, the lower the value, the greater the reduction in the execution time.

Based on such results, we also argue that *the best thread and data mapping combination for a single-source algorithm may also change if the source vertex changes*. For example, BFS-web executed from V1, V2, and V3 present 17%, 15%, and 18% performance improvement over De-De when executed with Co-In, Sc-In, and De-In, respectively.

1.2.3 Impact of Adjusting the Number of Threads

Although researchers have proposed different strategies to optimize graph applications (see chapter 3), most are unaware of the limitations of the parallel application scalability, i.e., assuming a regular behavior for the parallel execution using the maximum number of threads. Not all graph applications scale well, and their scalability will depend on the graph being processed, so reducing the number of threads (NT) may bring significant performance benefits. To illustrate that, we show in Fig. 1.4.a and Fig. 1.4.b the speedups results (*y*-axis) in two different views on *Intel32* machine: (a) the scalability of a specific graph application running different input graphs (the SSSP executing different input graphs by increasing the NT – *x*-axis); (b) the scalability of different graph applications running the same input graph (5 algorithms executing the Ber input graph). In addition, Table 1.1 shows the best number of threads when executing the 5 algorithms on *Intel88* (Intel Xeon E5-2699 v4 with 2 nodes / 44 cores / 88 threads) – explained

Table 1.1: The best number of threads on *Intel32* (Intel Xeon E5-2640v2 with 2 nodes / 16 cores / 32 threads) and on *Intel88* (Intel Xeon E5-2699 v4 with 2 nodes / 44 cores / 88 threads).

	Intel32					*Intel88*				
	BC	BFS	CC	PR	SSSP	BC	BFS	CC	PR	SSSP
road	32	32	32	32	32	40	30	44	44	44
cit	16	32	32	32	4	88	24	44	88	4
ama	14	32	32	32	1	18	36	34	34	1
CA	32	16	32	32	1	40	24	88	88	1
Ber	16	14	32	32	2	22	18	44	32	4

Source: The author.

later. For both Fig. 1.4 and Table 1.1, we consider the default policies for TM and PM (Def-Def). These results show the same challenging aspects in designing optimization strategies for graph applications previously described (see section 1.1):

- For **(1.)**, the scalability of a specific application depends on the input graph being processed. This fact is highlighted in Fig 1.4.a, where we see different scalability patterns in the SSSP executing over different input graphs;
- For **(2.)**, not all graph applications scale well by increasing NT, so the best number of threads will likely differ for different algorithms. In 1.4.b., while the PageRank (PR) and Connected Components (CC) algorithms have their best speedup when using the maximum NT (32 threads), for the Betweenness Centrality (BC), Single-Source Shortest Path (SSSP), and Breath-First Search (BFS) algorithms, it is better reducing the NT to 16, 2, and 14, respectively;
- For **(3.)**, the best number of threads may change if the machine changes. Taking the cit graph as an illustrative example in Table 1.1, the best number of threads for its execution differs on *Intel32* compared to *Intel88* in most algorithms, except for SSSP.

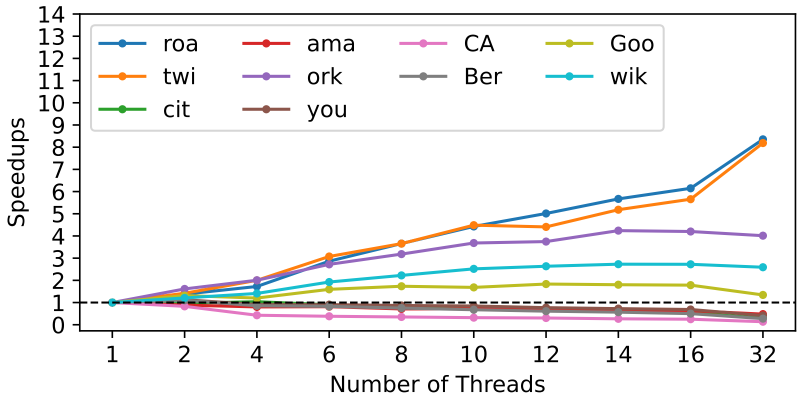
1.3 Contribution of this work

Below, we describe the steps to develop this thesis, as illustrated in Fig. 1.5.

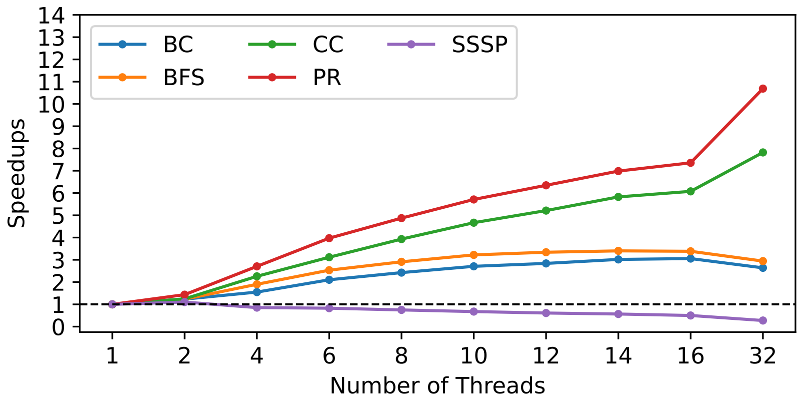
- **Design Space Exploration (DSE): TM+PM.** We analyzed the potential of tuning thread and data mapping in executing graph applications on NUMA machines. For that, we performed a DSE considering the combinations of several standard thread and data mapping policies (see **1** in Fig. 1.5). Based on that, we show that no

Figure 1.4: Optimization potential of tuning NT.

(a) SSSP executing over different input graphs with an increasing NT.



(b) Execution of 5 algorithms over the Ber input graph with an increasing the NT.



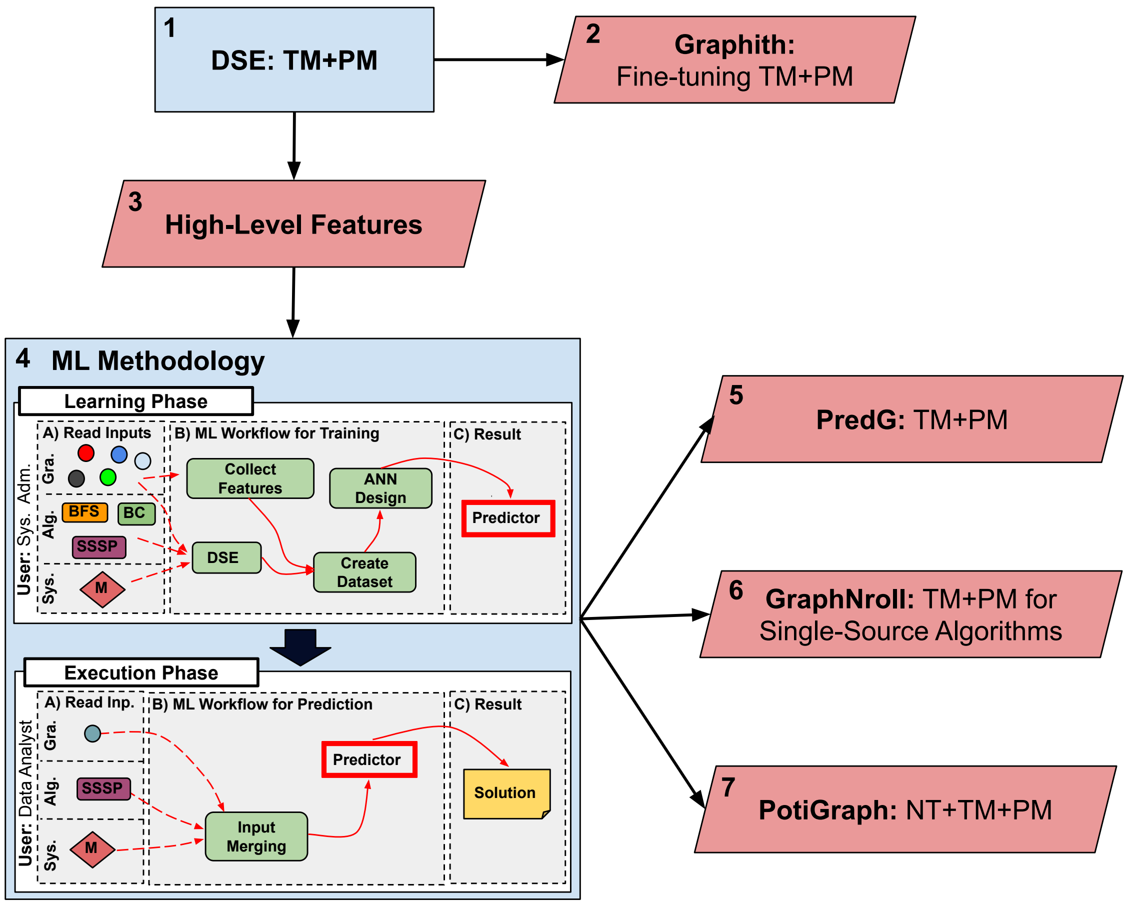
Source: The author.

particular solution will deliver the best result for all input graphs, graph processing algorithms, and NUMA machines.

From this experiment, we developed ***Graphith***, an optimization strategy for tuning the TM and PM of graph applications on NUMA systems (see **2** in Fig. 1.5). *Graphith* is a framework to boost the graph processing algorithm performance by fine-tuning the thread-to-core allocation and varying the data mapping policies, considering the NUMA system, graph algorithm, and input graph at hand. Although *Graphith* can improve the execution of graph applications, it is inflexible and has a high optimization overhead since every time the input graph changes, one must re-execute its expensive optimization process. This fact pressured us to search for alternative solutions with low overhead while providing high-performance execution for graph applications.

From the issue above, we observed that graph data have structural attributes not

Figure 1.5: Thesis' proposal overview.



Source: The author.

found in generic parallel applications that can be succinctly described by **High-Level Features** (see **3** in Fig. 1.5) such as the number of triangles, global clustering coefficient, and average degree, already available in most of the graph data sources (ROSSI; AHMED, 2015; LESKOVEC; KREVL, 2014), which reflect their different topologies (better explained in chapter 2). Since similar topologies often exhibit analogous high-level features, this similarity can be harnessed to propose new strategies to optimize the graph algorithm executions while providing adaptability with minimal run-time overhead.

We conducted additional experiments to assess the suitability of using the input graph high-level features to predict the best TM and PM for graph application execution. Our experiments also considered analyzing the applications' scalability, evaluating the best number of threads (NT) for each TM and PM combination. Based on that, we proposed a Machine Learning (ML) methodology used as a foundation for different approaches. This ML methodology leverages the input graphs' high-level features to predict different system-level variables (e.g., NT, TM, PM) to improve the execution of graph applications (see **4** in Fig. 1.5). The methodology consists of two phases, as described as follows.

Learning Phase. This phase aims to create a *Predictor* that will be used in the next phase (*Execution Phase*) to predict the best system-level variables (e.g., NT, TM, PM) for a given graph application. For that, the methodology receives as input (*Gra.*) a set of representative input graphs and (*Alg.*) algorithms as well as (*Sys.*) the NUMA machine where it will be deployed. Then, the methodology will perform an *ML Workflow for Training* based on the given inputs. It will collect the high-level features from the input graphs (Collect Features) and perform a DSE to select the best solutions for all the combinations of graphs and algorithms. Data from these procedures will be merged to generate a dataset (Create Dataset) that will be used to build an Artificial Neural Network (ANN) model (ANN Design) – our *Predictor* – to be applied in the next phase;

Execution Phase. This phase applies the *Predictor* to find the best system-level configuration (NT, TM, or PM) when a new incoming graph has to be executed. For that, the methodology receives (*Gra.*) the input graph along with its high-level features, (*Alg.*) the algorithm that will process the graph (that should have been considered during the *Learning Phase*), and (*Sys.*) the current NUMA machine. The methodology performs a *ML Workflow for Prediction* by merging the given input into the ANN input format and performing the prediction. Then, it fires the application execution with the system-level configuration set in the target NUMA system.

Using the above-described ML methodology as a foundation, we proposed three offline strategies that adapt to new input graphs or source vertices without any need for application profiling by adjusting: TM and PM simultaneously (TM+PM) for generic graph applications and also for the specific cases of single-source algorithms; and NT, TM, and PM simultaneously (NT+TM+PM) for generic graph applications.

- **PredG.** It is an ML tool that improves the graph analytics execution time by predicting the ideal TM+PM policies configuration as ***new input graphs*** need to be processed;
- **GraphNroll.** It is highly based on the investigation carried out specifically for single-source graph algorithms. *GraphNroll* framework enhances the single-source graph algorithms' execution time by predicting the ideal TM+PM poli-

cies configuration as the ***source vertices change*** for any input graph, algorithm, and NUMA machine;

- **PotiGraph.** It is a framework that exploits the scalability of graph applications by adjusting the NT along with the TM and PM (NT+TM+PM), finding the best configuration as ***new input graphs*** need to be processed.

1.4 Document Organization

We organized this work as follows. Chapter 2 introduces the basics of graph processing and provides a background on the NUMA systems, presenting the search space for NT along with the TM and PM techniques evaluated in this thesis. Chapter 3 discusses the related work that applies NT, TM, or PM for optimizing generic parallel applications and also for the specific case of graph algorithms. This Chapter also highlights our main contributions concerning the state-of-the-art. Chapter 4 presents some preliminary experiments that have led us to converge to this proposal. It discusses some optimization potentials we found by performing a DSE and *Graphith*, our strategy to provide high-quality TM+PM solutions on NUMA machines. Chapter 5 shows how we can use the input graphs' high-level features to propose the graph processing framework *PredG*. Chapter 6 presents *GraphNroll*, a graph processing framework for single-source algorithms. Chapter 7 describes our last proposal, *PotiGraph*, which extends from *PredG*, but now considering the optimization on NT, TM, and PM altogether. Finally, Chapter 8 concludes this thesis with some final considerations.

2 BACKGROUND

In this section, we introduce the concepts used throughout this thesis, such as input graphs, graph algorithms, NUMA machines, and thread and data mapping.

2.1 Graphs Modeling Real-World Problems

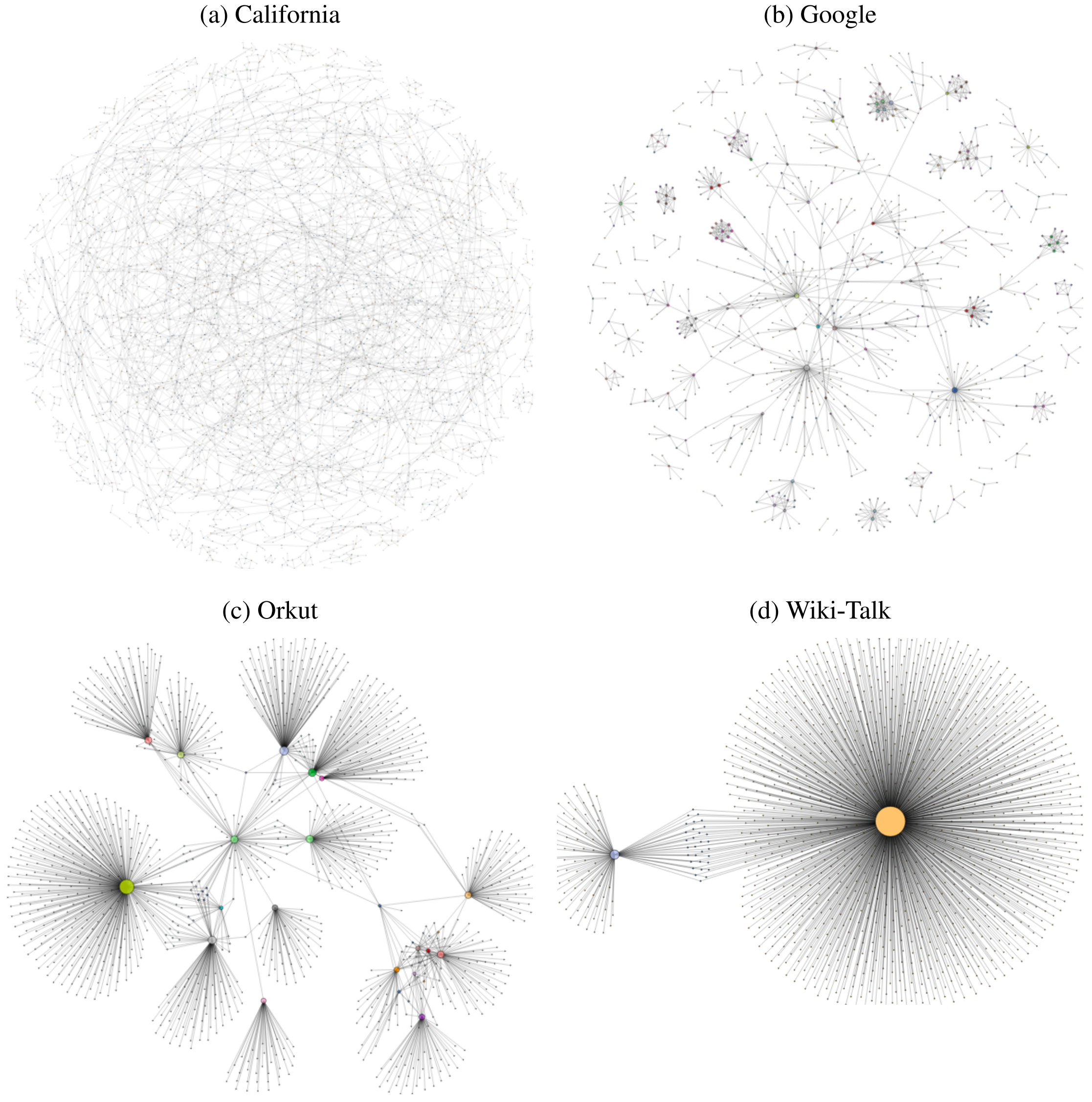
A graph is a data structure that accurately models abstract elements and their relationship. A graph is formally defined as $G = (V, E)$ where V represents the set of vertices (i.e., elements) and E is the set of edges (i.e., the relationship among the elements of V). Each $(v, u) \in E$ in a directed graph strictly indicates an edge pointing from v to u. On the other hand, for an undirected graph, each $(v, u) \in E$ represents a mutual relationship (i.e., there is an edge pointing from v to u and vice-versa) (GUI et al., 2019; LU et al., 2021).

Although graph presents a simple notation, it is a powerful data structure that can model problems related to diverse fields, such as social networks graphs (RAPOPORT; HORVATH, 1961), WWW (HUBERMAN, 2003), citation graphs for academic papers (MITCHELL, 1974), airline routes (AMARAL et al., 2000), and biological graphs to represent neural networks (WHITE et al., 1986) and protein interaction (JEONG et al., 2001). Real-world graphs (i.e., those extracted from actual sources) comprise a huge amount of data and may present different topological structures. A survey conducted by Sahu et al. (2017) considering graph software developers, users, a review of the mailing lists, source repositories, and white papers shows that the largest graphs comprise *billions* of edges.

Regarding their topologies, we show in Figure 2.1 the diversity in the structure of the graphs extracted from actual data sources. The depicted graphs are: (a) A road network of California[1] where vertices represent endpoints and road's intersections and undirected edges represent the roads; (b) A crawl of the Google web pages[2] where vertices represent web pages and edges represent hyperlinks between them; (c) A friendship social network and user-defined communities extracted from Orkut[3] where vertices represent users and edges represent their friendship; (d) Network of talk pages changes of

[1]California road network (Accessed on September 26th, 2023).
[2]Google web graph (Accessed on September 26th, 2023).
[3]Orkut social network and ground-truth communities (Accessed on September 26th, 2023).

Figure 2.1: Diversity in the topologies of the graph extracted from actual data sources.



Source: Extracted from the Network Repository (ROSSI; AHMED, 2016).

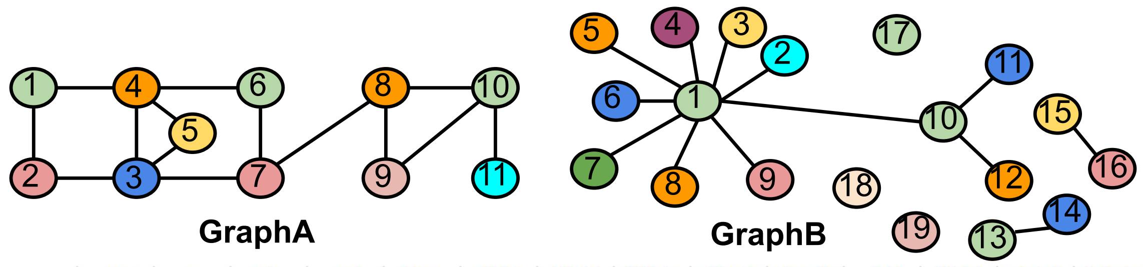
Wikipedia[4] where vertices represent users and a directed edge from vertex v to u represents that user v at least once edited a talk page of user u.

As shown in Figure 2.1, actual graphs present large diversity in their structure. For example, some show significant sparsity (e.g., California and Google) and density (e.g., Orkut and Wiki-Talk). In addition to the visual/graphical representation, we can identify these graph diversity by analyzing their high-level features, such as the *diameter* and *averaged degree*. In Fig 2.2, we illustrate some important features extracted from two different toy input graphs (*GraphA* and *GraphB*). The features are described as follows.

- *Number of Vertices (V)*: It is the number of vertices in the graph;

[4]Wikipedia Talk network (Accessed on September 26th, 2023).

Figure 2.2: Example of the high-level features values extracted from 2 synthetic graphs (GraphA and GraphB).



	V	E	Dia	IV	Den	LCC	GCC	MinD	MaxD	AvgD	DA	NCC	SLCC	PLCC
GraphA	11	14	6	0	0.25	0	0.27	1	4	2.54	-0.29	1	11	100
GraphB	19	13	3	3	0.07	0	0	0	9	1.4	0.22	6	12	63.16

Source: The author.

- *Number of Edges (E)*: It is the number of edges in the graph;
- *Vertex Degree (VD)*: For undirected graphs, it refers to the number of edges incident to a vertex. In the case of directed graphs, it is the number of incoming (in-) and outgoing (out-) edges (also called in-degree and out-degree, respectively);
- *Diameter (Dia)*: It is the largest number of vertices that must be traversed to travel from one vertex to another of the graph. For example, in Fig 2.2, the Dia of GraphA is 6, which is the distance from vertex 1 to 11;
- *Isolated Vertices (IV)*: It is the number of vertices without connection with any other vertices. For example, while GraphA has no IV, the GraphB has 3 IV;
- *Density (Den)*: It represents the ratio between the number of edges in a graph and the maximum possible number of edges that this graph can contain. In other words, it provides an idea of how dense a graph is regarding edge connectivity. For example, for a complete (fully dense) directed or undirected graph, the Den is always 1. On the other hand, a graph with all IV has a Den of 0;
- *Local Clustering Coefficient (LCC)*: It is the clustering coefficient of a vertex in a graph. The LCC quantifies how close the neighbors of a given vertex are to be a complete subgraph, i.e., a clique – a subgraph in which every pair of vertices are adjacent (LUCE; PERRY, 1949). In Fig 2.2, the LCC for both graphs is 0 when considering their vertex 1;
- *Global Clustering Coefficient (GCC)*: The GCC is based on the concept of triplets of vertices: 3 vertices that are connected by either 2 (open triplet) or 3 (closed triplet) undirected edges. Thus, it is computed by the number of closed triplets divided by the total number of triplets (both open and closed) (LUCE; PERRY, 1949). An alternative way to obtain the GCC is by computing the overall level of clustering in

a graph (WATTS; STROGATZ, 1998). It can be obtained by computing the average of the LCC of all the vertices;

- *Minimum Degree (MinD)*: It is the vertex degree with the minimum number of edges. In the example of Fig 2.2, MinD of GraphA and GraphB are 1 (vertex 11) and 0 (vertices 17, 18, and 19), respectively;
- *Maximum Degree (MaxD)*: It is the degree of the vertex with the maximum number of edges. In the example of Fig 2.2, *MaxD* of GraphA and GraphB are 5 (vertices 3 and 4) and 9 (vertex 1), respectively;
- *Averaged Degree (AvgD)*: It is the average vertices' degrees;
- *Degree Assortativity (DA)*: It is the tendency for vertices with high degrees in a graph to be connected to high-degree vertices (it is the same for low-degree vertices);
- *Number of Connected Components (NCC)*: A connected component (CC) is a set of vertices in which every pair of vertices is linked by path (i.e., a sequence of edges that joins a sequence of distinct vertices). Thus, the NCC is the number of CCs in the graph. Notice that while GraphA has only 1 CC (all vertices are connected), GraphB has 6 CC, including the IVs;
- *Size of the Largest Connected Components (SLCC)*: It is the number of vertices in the graph's largest CC;
- *Percentage of the Largest Connected Components (PLCC)*: It is the percentage of vertices in the largest CC of the graph. Notice that only 63.16% of GraphB's vertices belong to its largest CC;
- *Degree Distribution (DD)*: It is the probability that defines the number of edges connecting the graph's vertices. The DD of a graph can be defined as: *normal* when it follows a normal distribution; *power-law* when it follows an exponential distribution (a few vertices connected by most of the graph's edges). In this case, we can define the fraction of vertices with k edges to other vertices as $P(k) \sim k^{-\gamma}$, where γ is a parameter usually in the range $2 < \gamma < 3$ (BEAMER; ASANOVIC; PATTERSON, 2015); and *bounded*, which usually follows any distribution, but the degree of any vertex is limited to a specific value. It is worth noticing that the DD is directly related to the above features. For example, while a *power-law* graph may present a low Dia, a lot of low-degree vertices (small value for MinD) and a few high-degree vertices (very large value for MaxD), the *bounded* are identified

Table 2.1: Input graphs' high-level features

	V	E	Dia	IV	Den	LCC	GCC	MinD	MaxD	AvgD	DA	NCC	SLCC	PLCC
urand	1.34e+08	2.15e+09	7	0.00e+00	0.00e+00	0.00	0.00	6.00	6.80e+01	32.00	0.00	1.00e+00	1.34e+08	100.00
kron	1.34e+08	2.11e+09	6	7.11e+07	0.00e+00	0.01	0.04	0.00	1.57e+06	31.47	-0.04	7.12e+07	6.30e+07	46.96
twitter	6.16e+07	1.20e+09	14	1.99e+07	1.00e-06	0.05	0.08	0.00	3.00e+06	39.06	-0.04	1.99e+07	4.17e+07	67.64
web	5.06e+07	1.81e+09	135	9.20e+01	1.00e-06	0.63	0.68	0.00	8.56e+06	71.49	-0.02	1.23e+02	5.06e+07	100.00
road	2.39e+07	2.89e+07	6304	0.00e+00	0.00e+00	0.02	0.02	1.00	9.00e+00	2.41	0.08	1.00e+00	2.39e+07	100.00
cit-patents	6.01e+06	1.65e+07	22	2.23e+06	1.00e-06	0.12	0.09	0.00	7.93e+02	5.50	0.18	2.24e+06	3.76e+06	62.64
orkut	3.07e+06	1.17e+08	9	1.86e+02	2.50e-05	0.17	0.17	0.00	3.33e+04	76.28	0.08	1.87e+02	3.07e+06	99.99
wikitalk	2.39e+06	4.66e+06	9	0.00e+00	2.00e-06	0.05	0.20	1.00	1.00e+05	3.89	-0.04	2.56e+03	2.39e+06	99.77
california	1.97e+06	2.77e+06	849	6.08e+03	1.00e-06	0.05	0.05	0.00	1.20e+01	2.81	0.19	8.71e+03	1.96e+06	99.28
texas	1.39e+06	1.92e+06	1054	1.35e+04	2.00e-06	0.05	0.06	0.00	1.20e+01	2.76	0.19	1.39e+04	1.35e+06	96.97
youtube	1.16e+06	2.99e+06	20	2.29e+04	4.00e-06	0.08	0.18	0.00	2.88e+04	5.16	-0.03	2.29e+04	1.13e+06	98.02
pennsylvania	1.09e+06	1.54e+06	786	2.83e+03	3.00e-06	0.05	0.06	0.00	9.00e+00	2.83	0.18	3.03e+03	1.09e+06	99.69
google	9.16e+05	4.32e+06	21	4.07e+04	1.00e-05	0.49	0.62	0.00	6.33e+03	9.43	-0.05	4.35e+04	8.56e+05	93.38
berkley	6.85e+05	6.65e+06	514	1.00e+00	2.80e-05	0.60	0.63	0.00	8.42e+04	19.41	-0.11	6.77e+02	6.55e+05	95.56
amazon	5.49e+05	9.26e+05	44	2.14e+05	6.00e-06	0.24	0.43	0.00	5.49e+02	3.38	-0.06	2.14e+05	3.35e+05	61.04

Source: The author.

by high Dia with a relatively small value to MaxD (usually close to MinD).

Several other graph features can be analyzed (BONDY; MURTY et al., 1976), however, by using only the above-described ones, we can accurately identify the structure of a specific graph. In Table 2.1, we show examples of some high-level features extracted from 15 different input graphs available in the GAP Benchmark Suite (GAPBS) (BEAMER; ASANOVIĆ; PATTERSON, 2015) (e.g., urand, kron, twitter, web, and road) and in the Stanford Network Analysis Project (SNAP) (LESKOVEC; KREVL, 2014) (e.g., cit-patents, orkut, wikitalk, california, texas, youtube, pennsylvania, google, berkley, and amazon).

Although there are different ways to classify the topology of graphs (SAHU et al., 2017; SHAO; LI, 2018), according to Beamer, Asanovic and Patterson (2012), we can simplify the analysis by focusing solely on the Dia and DD of real-world graphs. This approach allows us to classify them into two broad categories named after their emblematic members: *meshes* and *social networks*.

- *Meshes.* They are usually derived from physically spatial sources, such as road maps, infrastructure networks, and finite-element mesh applications. Thus, it is straightforward to partition mesh graphs into a few spatial dimensions. They are usually identified by having a high Dia and following a bounded and low DD. For instance, analyzing the Figure 2.1 and Table 2.1, the *california*, with a high Dia and the lower MaxD and AvgD (e.g., MaxD is bounded by 12 edges), represents a mesh;
- *Social Networks.* These networks come from non-spatial sources, so they are hard

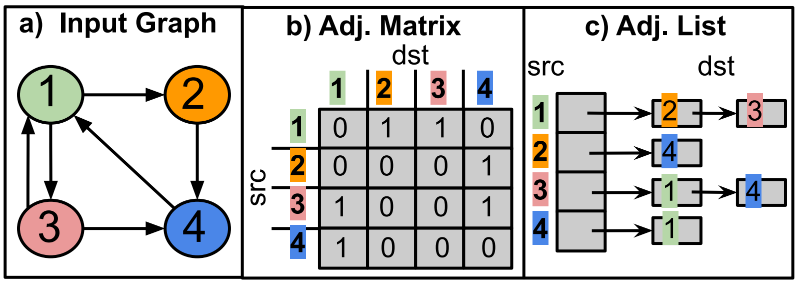
to partition into a reasonable number of dimensions. Social network graphs can represent WWW hyperlinks, social network websites, and social media apps, among many other applications. They are usually identified by having a low Dia and following a power-law DD. For example, the *orkut* graph, with its low Dia and high MaxD and AvgD, represents a social network graph (e.g., MaxD is 33,313). Additionally, such graphs are also known as "*small-world*" because of their low Dia, which means that most vertices are not neighbors of others, but most of them can be reached from every vertex by a small number of hops (WATTS; STROGATZ, 1998).

2.1.1 Graphs's In-Memory Representation

There are different ways to represent graphs as a data structure, such as edge list, adjacency matrix, adjacency list, etc., each with specific particularities. However, the top computer science textbooks mainly focus on adjacency matrices and adjacency lists (CORMEN et al., 2022; KLEINBERG; TARDOS, 2006). Next, we give an overview of such representations.

- ***Adjacency matrix.*** In the *adjacency matrix*, graphs are represented as a 2D matrix (V × V) of vertices, where the indexes of rows/columns indicate the vertices. The matrix positions consist of 0's and 1's. Thus, if an edge (v, u) connects two vertices v and u, it is indicated by 1. Otherwise, it is 0. For example, in Fig. 2.3b, we show the adjacency matrix representation of the example graph shown in Fig. 2.3a. Notice that while a directed edge connects vertices 1 and 2, there are no edges between 2 and 1. Thus, their respective matrix positions are 1 and 0;

Figure 2.3: Illustration of **a)** an example input graph and its representation in **b)** adjacency matrix and **c)** adjacency list.



src \ dst	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	1	0	0	1
4	1	0	0	0

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 \\
2 \arrow[r] & 4 & \\
3 \arrow[r] & 1 \arrow[r] & 4 \\
4 \arrow[r] & 1 &
\end{tikzcd}$$

Source: The author.

- ***Adjacency list.*** In the *adjacency list*, graphs are represented as an array of linked lists. The vertices are stored as indexes of a one-dimensional array, and the edges are stored as a linked list, representing the neighbors of each vertex. For example, in Fig. 2.3c, also considering the example graph shown in Fig. 2.3a, vertex 1 has directed edges to vertices 2 and 3.

An *adjacency matrix* is efficient in some operations, such as verifying if an edge connects a given pair of vertices in constant time. However, the major disadvantage is that it requires $O(|V|^2)$ memory space to store an input graph, even for sparse graphs (which are very common) in which edges do not connect most pairs of vertices – there will be many 0's in the matrix positions. Sparse graphs are common in practice, so using an adjacency matrix for large sparse graphs is not worth using since it wastes too much memory on zero entries. On the other hand, *adjacency lists* are a more flexible and compact way to represent graphs, so most textbooks recommend *adjacency lists* to represent sparse graphs (KLEINBERG; TARDOS, 2006; CORMEN et al., 2022).

Although an adjacency list provides an easy way to insert and remove vertices in the graph and a memory space that fits better to the number of vertices and edges of the input graph ($O(|V| \times |E|)$), it incurs overheads for space allocation and memory accesses: (*i*) the space allocation overhead comes from the need for a "next" pointer for every list' node, and each node will carry allocations overheads to create them (e.g., using `malloc` for C implementation); (*ii*) The overhead of memory accesses arises from chasing pointers across the address space, occurring when an algorithm traverses the adjacency list. In this scenario, the algorithm generates many random memory accesses, which are typically less efficient and slower than sequential memory accesses (ZHANG; CHEN; CHEN, 2015).

Compressed Sparse Row (CSR) format. To overcome the downsides mentioned above, most graph processing works use a compact and memory-hierarchy-friendly representation, called Compressed Sparse Row (CSR), to represent large graphs. CSR originates from high-performance scientific computing representing large sparse matrices comprised mostly of zeros. Its idea is to pack the column indices of non-zero entries into a dense array. The advantage of CSR is that it is more compact and fits more continuously in memory, eliminating any space overhead while reducing random memory accesses. However, CSR has reduced flexibility, so it is more suitable for input graphs with a fixed structure. In Fig. 2.4, we illustrate the CSR representation of the example graph presented in Fig. 2.3a, and we detail it below:

- The CSR format splits the input graph into two basic arrays: the *Vertices* array represents the vertices, with indices indicating the vertices' IDs, and the contents store the position in the *Edges* array; The *Edges* array stores the neighbors of the vertices, representing the edges of the graph.

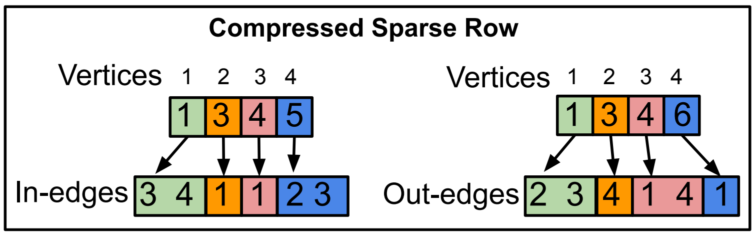
The above definition is used to represent undirected graphs. However, for directed graphs, as the one illustrated in Fig. 2.3a, two *Vertices* and *Edges* arrays are required to represent in- and out-edges. In this case, the input graphs are represented as follows:

– ***To represent in-edges:*** All vertices are partitioned by their target vertex (*Vertices* array), storing the position of source vertices (*In-edges* array). The *Vertices* array represents the vertices, where indexes indicate the vertices' ID and the contents store the position in *In-edges* array where the in-edges of specific vertices can be accessed. The *In-edges* array stores the IDs of the in-edges neighbors of each vertex. For example, to access the in-edges neighbors of vertex 4, an algorithm must access the position 4 of *Vertices* array to get the position to access its neighbors in *In-edge* array (in this case, position 5), to then access each neighbor sequentially (vertices 2 and 3);

– ***To represent out-edges:*** It is similar to the previous one, but in this case, the *Vertices* array stores the position in *Out-edges* array where the out-edge neighbors of specific vertices can be accessed. Similarly, the *Out-edges* array stores the IDs of the out-edge neighbors of each vertex. For example, to access the out-edge neighbor of vertex 4, an algorithm must access the position 4 of *Vertices* array to get the position to access its neighbor in *Out-edge* array (position 6), to then access the neighbor (vertex 1).

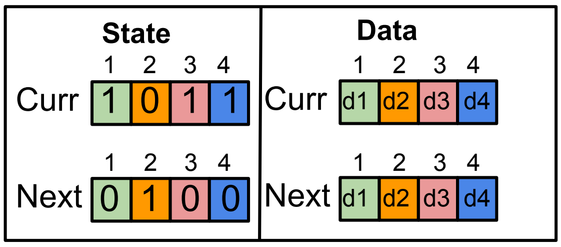
Most graph processing works mainly propose algorithm implementations that process the input graphs iteratively, so not all the vertices will be processed at a specific iteration (as will be explained later). Therefore, in addition to *Vertices* and *Edges* arrays (and

Figure 2.4: CSR representation of the input graph showed in Fig. 2.3a.



Source: The author.

Figure 2.5: Additional arrays used to represent **a)** the graph runtime state and **b)** the working data in iterative graph algorithm computation. This example is based on the example graph showed in Fig. 2.3a.



Source: The author.

the respective In-/Out-edges for in- and out-edges) of the CSR representation, iterative graph processing algorithms use two other kinds of arrays to represent the computation state (*State*) and the data being processed (Data):

- ***State.*** It represents the input graph runtime states, indicating the vertices being processed at a specific iteration (i.e., a frontier of active vertices). For that, it uses an array where the indexes indicate the vertices' ID, and the contents indicate if the vertices are active (1) or not (0). For example, Fig. 2.5 shows two arrays, *Curr* and *Next*, indicating the frontier of active vertices of the current and next iterations, respectively. In this example, vertices 1, 3, and 4 are active in the current iteration, and vertex 2 will be active in the next one;
- ***Data.*** It represents the application-defined data, thus, it depends on the kind of computation and amount of data computed/generated for each vertex. Similarly to *State*, it can be represented by arrays where indexes indicate the vertices' ID, and the contents are the working data of each vertex. The application-defined edge data can be handled similarly if necessary. In Fig. 2.5, we also illustrate two versions of the Data arrays, *Curr* and *Next*, which store the working data of the vertices in the current and next active frontiers, respectively.

2.2 Graph Processing Frameworks and Algorithms

Graphs extracted from actual data sources (e.g., communications, recommendation systems, and logistics) are continuously growing in size and complexity, so implementing algorithms to perform scalable analysis over such complex networks for information retrieval and data mining becomes a significant bottleneck (DHULIPALA; BLELLOCH; SHUN, 2021). To face that, since 2008, several programming paradigms, models, libraries, and frameworks focused on development productivity or speeding up graph analysis have been proposed in the literature (DOEKEMEIJER; VARBANESCU, 2014). However, such graph processing systems strive to find the best trade-off between productivity-enhancing front-ends (i.e., simple and user-friendly) and high-performance back-ends (i.e., optimized algorithms implementations), which means none of them will present neither the best way to express computation nor the best performance for all kinds of graph computation (DOEKEMEIJER; VARBANESCU, 2014).

The concepts of framework and library differ because of the notion of "inversion of control" (HUBERMAN, 2003): while libraries only offer a collection of objects and functions, leaving the coordination to the user, frameworks also take care of the control flow. Thus, used-defined methods are called by the framework itself, meaning it serves as an "extensible skeleton". Graph processing frameworks usually offer Domain-Specific Languages (DSL) as the front-end (i.e., programming interfaces) and compile to a lower-level execution. For example, graph DSLs such as OptiML (SUJEETH et al., 2011) and Green-Marl (HONG et al., 2012) target graph processing frameworks, offering a more natural programming interface for users familiar with the graph domain but not with programming in general.

The most used frameworks usually have off-the-shelf implementations of graph algorithms that are basic kernels of other complex applications, such as the Betweenness Centrality (BC), Breadth-First Search (BFS), Connected Components (CC), PageRank (PR), and Single-Source Shortest Paths (SSSP) – the ones we evaluate in this work. However, they differ in how the algorithms are implemented regarding the programming, computing models, and access to graph data structures. Next, we give an overview of some implementation decisions used on most graph processing frameworks.

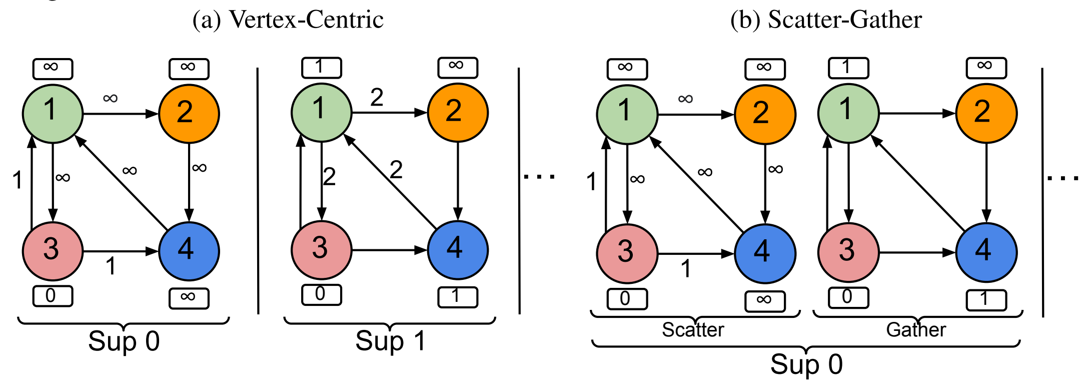
Graph programming models. In the context of graph computation, graph programming models refer to the basic data abstraction to analyze large-scale networks (MCCUNE; WENINGER; MADEY, 2015). There are different manners in graph processing frameworks to abstract the graph data, but the most common ones are: vertex-, edge-, and partition-centric.

- *Vertex-centric.* This model follows the *Think-Like-A-Vertex* (TLAV) principle, where the graph algorithm is expressed by a user-defined computation performed on each vertex (MCCUNE; WENINGER; MADEY, 2015). Thus, each vertex contains its own information and of all its outgoing edges. The computation is expressed in terms of a single vertex. Pregel (MALEWICZ et al., 2010) and GraphLab (LOW et al., 2014) are representative examples of vertex-centric graph processing frameworks. In both frameworks, the computation for a vertex involves receiving messages from other vertices and then updating the states and data of the vertex and its neighbors;

- *Edge-centric.* This model considers edges as the fundamental unit for graph computation, so the graph is partitioned by its edges, i.e., each edge is a single partition (HEIDARI et al., 2018). The partitioning can be done by scanning the entire edge list of the graphs. X-Stream is a representative example of an edge-centric graph processing framework (ROY; MIHAILOVIC; ZWAENEPOEL, 2013);
- *Partition-centric.* This model considers subgraphs (i.e., partitions) as the fundamental unit for graph computation. Partitions can be defined as disjoint sets of contiguously labeled vertices. For example, the work of Lakhotia et al. (2020) uses a lightweight index scheme for partitioning vertices into k disjoint sets of equal size: the set of vertices V_p of a partition p comprises all the vertices with indices in the interval $[p\frac{V}{k}, (p+1)\frac{V}{k}]$, while E_p denotes all the edges in p.

Although the vertex-centric model is easy to program and has been proven to be useful for many algorithms (MCCUNE; WENINGER; MADEY, 2015) – it suits well for traversing along all the vertices in a graph –, it is very short-sighted: it has the information of only the immediate vertex's neighbors, so the information is propagated slowly throughout the graph. Consequently, it takes many computation steps to propagate information from a source to a destination vertex, which makes it unsuitable for subgraph-centric perspective algorithms such as local clustering coefficient and triangle counting (KALAVRI; VLASSOV; HARIDI, 2017). Moreover, it incurs heavy communication overhead with highly irregular memory accesses through indices or pointers, which conventional memory controllers can not efficiently handle. On the other hand, the edge-centric model traverses the graph in a streaming fashion, eliminating the random memory accesses to the edges. With that, it has shown superior performance than the vertex-centric model for those graphs whose number of edges is much larger than the number of vertices (ROY; MIHAILOVIC; ZWAENEPOEL, 2013). In the case of the partition-centric, it leverages vital information about subgraphs (e.g., their connections) to provide exclusive partition ownership to one thread, avoiding unnecessary synchronization and achieving good scalability. For example, the proposal of Lakhotia et al. (2020) partitions the graph into cacheable vertex subsets, providing cache efficiency and high bandwidth for sequential memory accesses.

Figure 2.6: Example of SSSP execution with a) Vertex-centric and b) Scatter-Gather computing models.



Source: The author.

Graph computing models. In general, graph processing algorithms need to traverse the input graph in some way to retrieve information, so they naturally perform iterative computation (BATARFI et al., 2015). Based on that, the graph processing frameworks have used different iterative computation models, such as the *Vertex-Centric iterative processing* and *Scatter-Gather*, explained in the following.

- *Vertex-Centric iterative processing.* Introduced in the work of Malewicz et al. (2010), this computing model follows the TLAV principle. Similar to its correspondent vertex-centric programming model detailed above, this model focuses on the information of a vertex and those of all its one-hop neighbors. The computation process occurs in synchronized iterations, called *supersteps*. In each *superstep*, the framework processes all the active vertices in parallel by calling a user-defined function which will perform the computation over the vertices' data in a single *superstep*. Each vertex has a unique ID and can communicate with any other vertices in the graph through messages. As the *supersteps* are executed synchronously, the messages sent during one *superstep* are guaranteed to be delivered at the beginning of the next *superstep* (i.e., they will be available to the vertex function of the vertex which is the receiver of the message).

  As an illustrative example, in Fig. 2.6a, we present the execution of the SSSP algorithm over the example graph depicted in Fig. 2.3a with the vertex-centric computing model. We illustrate only two *supersteps* (*Sup 0* and *Sup 1*), but the remaining ones perform similarly. Initially, all the vertices are set to an infinite distance value (∞), except the source vertex (vertex 3), which is set to zero. In *Sup 0*, the source vertices propagate distances to their neighbors. For example, vertex 3 sends messages to vertices 1 and 4, which is the sum of its current distance with the edge cost (assuming that all the edges have cost 1). During the following *supersteps*, each vertex checks its received messages and chooses the minimum distance among them. If a value is smaller than the current one, it updates its value and produces messages to its neighbors. For instance, in *Sup 1*, while the vertex 3 is not updated, so it does not send messages, the vertex 1 is updated and then sends messages to its neighbors (vertices 3 and 2). The algorithm converges when it does not update vertices or reaches the maximum number of *supersteps*.

- *Scatter-Gather.* This computing model also divides the computation in synchronized *supersteps*, but each *superstep* computes a *Scatter* phase followed by a *Gather* phase. In the *Scatter* phase, each of the active vertices executes a user-defined function to send messages along out-going edges, and during the *Gather* phase, they collect messages from neighbors and execute another user-defined function to update the vertices' states based on the received messages. It is worth noticing that, unlike the Vertex-centric model, *Scatter-Gather* sends and receives messages in the same *superstep*.

  To illustrate this computing model, consider again the execution of the SSSP algorithm over the example graph depicted in Fig. 2.3a, but now in Fig. 2.6b. We illustrate only the first *superstep* (*Sup 0*), but the remaining *supersteps* work in the same way. In the *Scatter* phase, each vertex sends a candidate distance message to all its neighbors. For example, the source vertex (vertex 3) sends messages to vertices 1 and 4, which is the sum of its current distance with the edge cost. In the *Gather* phase, with the received candidate distances messages, each vertex calculates the minimum distance, and if a shorter path has been discovered, it updates its value. If a vertex does not change its value during a *superstep*, it does not produce messages for its neighbors for the next *superstep*. The algorithm converges when there are no more updates.

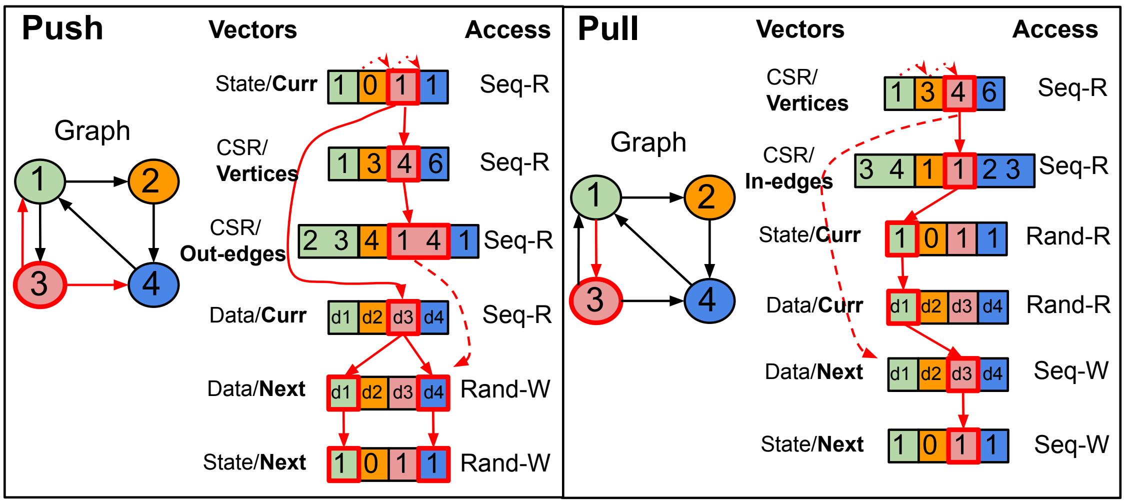
The Vertex-centric computing model is the most used to express a broad set of iterative graph algorithms. This model fits well for computation that can be expressed as a local vertex function that only needs to access data on adjacent vertices and edges, such as the PageRank algorithm. However, it is not trivial to compute graph transformations and single-pass graph computation (i.e., non-iterative computation) in this model. Because of that, the Vertex-centric model is not a good fit for algorithms like triangle counting. In the case of the *Scatter-Gather* model, it is a computing abstraction that can express different graph algorithms concisely and elegantly. As it decouples the logic of producing messages to the logic of updating vertex values based on the received messages, programs written in this model are usually easy to follow and maintain. Like the Vertex-centric model, this model is also a good fit for iterative, value-propagation algorithms such as the PageRank algorithm.

Access to graph data structures. In addition to the programming and computation models described above, the way the graph application accesses the CSR data structure (described in section 2.1.1) and how the data is propagated throughout the vertices/edges also significantly impacts on its performance. This is done by two basic modes (*push* and *pull*), applied independently of the programming and computation models used. Next, we explain both modes, giving an illustrative example in Fig. 2.7 considering the ordering and the types of accesses to the data structure shown in section 2.1.1 – CRS (*Vertices* and *In-/Out-edges*), *State*, and *Data* arrays. The types of accesses are sequential (*Seq*) or random (*Rand*), and for reading (*R*) or writing (*W*) data.

- *Push direction.* In this mode, at each iteration, the algorithm first evaluates the current vertices frontier (i.e., the *State/Curr* array in Fig. 2.5) to ***push***/propagate the data of the active vertices to their outgoing neighbors. For that, it performs the following steps (see Fig. 2.7 – left side):

  (i) The worker thread first scans the *State/Curr* state array (Sequentially for Reading data, Seq-R) to identify active vertices in the *CSR/Vertices* array (Seq-R) and then obtains its outgoing neighbors through the *CSR/Out-edges* array

Figure 2.7: Push and Pull accessing modes to the graph data structure.



Source: The author.

(Seq-R);

(ii) Thus, the worker thread *pushes* the value of the active vertex in the *Data/Curr* array (Seq-R) to its neighbors in the *Data/Next* data array (Randomly for Writing data, Rand-W) and sets the *State/Next* array (Rand-W).

For instance, in Fig.2.7 (left side), the value of vertex 3 will be *pushed* to its neighboring vertices 1 and 4 along out-edges, which we highlighted in red arrows.

- *Pull direction.* In this mode, at each iteration, the algorithm evaluates the entire vertices array (i.e., the *CSR/Vertices* array in Fig. 2.5 – right side) to ***pull***/gather the data of their active incoming neighbors to update their own data. For that, it performs the following steps:

  (i) For each vertex in the *Vertices* array (Seq-R), the worker thread first obtains its active neighbors through the *CSR/In-edges* array (Seq-R) and *State/Curr* state arrays (Rand-R);

  (ii) Thus, the worker thread *pulls* the value from the active neighbors in the *Data/Curr* array (Rand-R) to update the vertex's data in the *Data/Next* array (Seq-W) and sets the *State/Next* array (Seq-W).

  For instance, vertex 3 will *pull* the value of its neighbor vertex 1 along the in-edge to update its own value.

As mentioned, the graph processing algorithms perform differently when implemented with the above modes. For instance, the work of Beamer, Asanović and Patterson (2017) showed that the pull-based implementation of the PageRank algorithm is often more efficient than the push-based one since it removes the need for atomic operations to add each neighbor's rank score. Beamer, Asanovic and Patterson (2012) have also shown that a hybrid push-pull implementation benefits the Breath-First Search (BFS) algorithm. The observation is that the *push* mode is better when the frontier of active vertices is small (i.e., a few 1's in the *State/Curr* array). Otherwise, the *pull* mode will yield speedups when the active frontier is a substantial fraction of the total graph, commonly occurring in small-world graphs such as social networks. Later, this idea was generalized by the work of Shun and Blelloch (2013), which implemented different graph algorithms that switch between push and pull modes depending on the size of the frontier of active vertices.

As a summary, although different frameworks combine some of the techniques mentioned above to provide efficient execution for various algorithms and input graphs,

Table 2.2: Graph algorithms' features.

	single-source /whole-graph	**(Un)Weighted**	**Vertex per iteration**	**push /pull**	**traversal/ compute**
BC	single-source	Unweighted	Part	push	compute
BFS	single-source	Unweighted	Part	push e pull	traversal
CC	whole-graph	Unweighted	Part	push	traversal
PR	whole-graph	Unweighted	All	pull	compute
SSSP	single-source	Weighted	Part	push	traversal

Source: The author.

each graph algorithm has its characteristics that need to be considered to achieve high performance, such as: whether it requires a starting vertex or not; if it considers the weight of the graphs; the number of vertices processed in each iteration; and also if the algorithm focuses on to traverse the graph's vertices (e.g., search for a vertex using BFS) or to compute properties of the graph's vertices (e.g., compute the rank score of a vertex using the PR algorithm).

2.2.1 Graph Algorithms

This thesis mainly evaluates five distinct graph algorithms, which represent many applications within social network analysis, engineering, and science (SAHU et al., 2020). We considered algorithms that focus the optimization in both graph traversal and compute the vertices' properties. Next, we describe them and summarize their characteristics in Table 2.2.

- ***PageRank (PR).*** Although it is the core of the Google search engine (ROGERS, 2002), the PR algorithm has been applied in many other applications, such as to quantify the scientific impact of researchers and in the analysis of protein networks (SENANAYAKE; PIRAVEENAN; ZOMAYA, 2015; IVÁN; GROLMUSZ, 2011). PR algorithm iteratively calculates the rank scores for all vertices in the graph. For that, it computes the rank score for each vertex using the following formula:

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N^-(v)} \frac{PR(u)}{|N^+(u)|},$$

where d is the dumping factor, V is the set of vertices in the graph, $N^-(v)$ is the set of vertices that v has an in-going edge to them, and similarly, $N^+(u)$ is the set of vertices that u has an out-going edge to them. Therefore, the score of a vertex v is based on the score of the vertices that point to v ($u \in N^{-}(v)$). The dumping factor d is a click-through probability, which prevents vertices with no outgoing edges (sinks) from absorbing the ranks of those vertices connected to the sinks. Thus, $d = 0$ means that all edges are random and uniformly distributed throughout the vertices (the $\frac{1}{|V|}$ coefficient in the first term) by definition. A common value for this parameter is 0.85 (FU; LIN; TSAI, 2006).

- ***Connected Components (CC).*** This algorithm is used in social network applications to identify groups of people who are friends of each other or who have common interests (BEAMER; ASANOVIĆ; PATTERSON, 2015). A CC is a set of vertices $V_c \subseteq V$ in which an undirected path exists between any pair of vertices belonging to V_c. The algorithm assigns a unique label to each CC and all its vertices. Vertices with zero degrees (isolated vertices) are considered CC and receive labels. When the input graph has direct edges, the algorithm only requires weakly connected components. Thus, in this case, a CC is interpreted as a set of vertices $V_c \subseteq V$ where for each pair of vertices $(v, u) \in E$ there exists a path from v to u or from u to v.
- ***Breadth-First Search (BFS).*** One can find BFS in the literature as a kernel of several other graph processing algorithms, such as the connected components and the topological ordering (BEAMER; ASANOVIC; PATTERSON, 2012; MCCUNE; WENINGER; MADEY, 2015). It has wide applicability in recommendation systems, such as the one used by Alibaba's Website (SAHU et al., 2020). BFS algorithm traverses the graph from a source vertex, visiting all vertices of the same depth (distance from the source) before advancing to the next one. For that, it keeps track of parent vertices. However, for any vertex reached from the source, there is often more than one possible parent vertex since any incoming vertex in a depth one less than the reached vertex could be its parent. Thus, different parent vertices cause more than one correct solution to BFS. Because of that, a correct solution for the BFS starting from a source vertex gives a final parent array satisfying the following rules: Let $source, v, u, \in V$, where $source$ is the source vertex, and *parent[v]* and *depth[v]* are the parent and depth of a vertex v, respectively.
  - $parent[source] = source$;
  - $parent[v] = -1$ if v is unreachable from $source$;

  - if v is reachable from $source$ and $parent[v] = u$, there exists an edge from u to v;
  - if v is reachable from $source$ and $parent[v] = u$, $depth[v] = depth[u] + 1$.

- ***Single-Source Shortest Paths (SSSP).*** This algorithm has been widely applied in Global Positioning System (GPS) applications for path mapping purposes (SAHU et al., 2020). However, it also has applicability in plant and facility layout, robotics, and VLSI design (CHEN, 1996). SSSP calculates all the shortest reachable paths from a given source vertex. The distance between a pair of vertices is defined as the sum of edge weights along the minimum path connecting them. Unlike BFS, SSSP does not keep track of the parent vertices since the final solution is the distances and not the shortest paths. Although there may be more than one shortest path connecting a pair of vertices, all of them will have the same distance. Therefore, a correct SSSP solution is the one that satisfies the following rules: Let $source, v, u, \in V$, where $source$ is the source vertex, and *distance[v]* being the $distance$ from the source vertex.

  - $distance[source] = 0$;
  - $distance[v] = \infty$ (or some known sentinel value) if v is unreachable from $source$;
  - If v is reachable from $source$, there is a path of combined weight $distance[v]$ from the $source$ to v;
  - If v is reachable from $source$, there is no path of combined weight less than $distance[v]$ from the $source$ to v.

- ***Betweenness Centrality (BC).*** One can find practical BC applications in social network analysis (STOLZ; SCHLERETH, 2021), and the analysis of the topological complexity of river networks as well as their use in maritime trade (SARKER et al., 2019). BC approximates the BC score of the graph by a subset of vertices, in which the score for a vertex v is the ratio between the shortest paths that go through v and all the shortest paths of the considered set. BC updates the vertices scores using the following formula:

$$BC(v) = \sum_{s,t \in V, s \neq v \neq t} \frac{\sigma_{st(v)}}{\sigma_{st}},$$

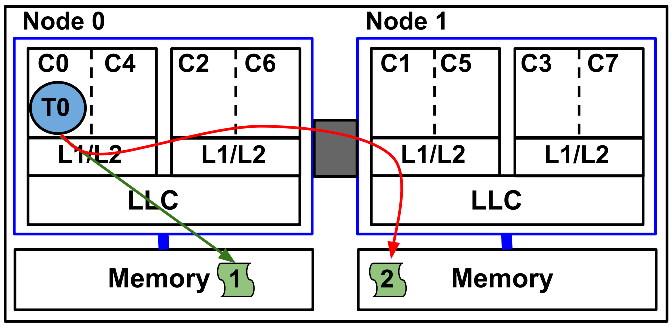
where σ_{st} is the number of shortest paths between vertices s and t, and $\sigma_{st(v)}$ is the number of those shortest paths that pass through v.

2.3 NUMA Systems

The computing power and memory storage capacity of the High-Performance Computing (HPC) servers have been increasing on a large scale in recent years, pushing forward applications that extract useful information from massive data (DHULIPALA; BLELLOCH; SHUN, 2021). Modern HPC servers comprise several shared memory multiprocessors based on Non-Uniform Memory Access (NUMA) architecture. NUMA machines present better scalability than Uniform Memory Access (UMA) ones as the number of cores increases, supporting higher available bandwidth. Because of that, such systems have become commonly used in top-end HPC servers (for example, the Fugaku Supercomputer, which led the TOP500 in November 2020[5]).

NUMA systems comprise several NUMA nodes. A NUMA node consists of a multicore processor with its own memory controller and memory attached. NUMA nodes are connected by fast interconnect links (e.g., QuickPath Interconnect and HyperTransport Interconnect on Intel and AMD computers, respectively). Figure 2.8 illustrates a NUMA system composed of 2 nodes (Node 0 and Node 1). Each node has 2 cores with 2-way Simultaneous Multi-Threading (SMT), resulting in a total of 8 hardware threads, e.g., 4 physical cores (C0, C1, C2, and C3) and 4 logical cores (C4, C5, C6, and C7). As a real example, the Fugaku Supercomputer mentioned above comprises many A64FX processors, which are Arm-based HPC processors composed of 4 NUMA nodes with 12 cores and 48 hardware threads.

Figure 2.8: Representation of a NUMA system.



Source: The author.

Although NUMA systems have more than one physical memory, they usually

[5]Top500 - Supercomputer Fugaku.

share a single physical address space, with consistency being kept by the use of advanced cache coherence mechanisms. Therefore, threads running on any core can access data locally (in the same NUMA nodes' memory) or remotely (in any available system's memories). However, as the interconnect link is slower than the local node's memory bus, remote memory accesses present higher latency than the local ones. Similarly, data accesses resolved by remote caches also impose a higher latency (DIENER et al., 2015).

Therefore, where threads and data are located across the available resources plays an essential role in the performance: when threads request data from a memory region located in another node (remote access), the access latency will be higher than if it was accessing data from its node's memory (local access). Hence, reducing the number of accesses to remote memory will likely improve application performance (TAM; AZIMI; STUMM, 2007; DIENER et al., 2015). Such performance benefits also come when the parallel application uses only the necessary processing resources obtained by setting a number of threads compatible with the application's scalability. Next, we describe how to adjust the thread and data mapping as well as the number of threads on NUMA machines.

2.4 Thread/Data Mapping and Adjusting Number of Threads on NUMA Systems

2.4.1 Thread Mapping

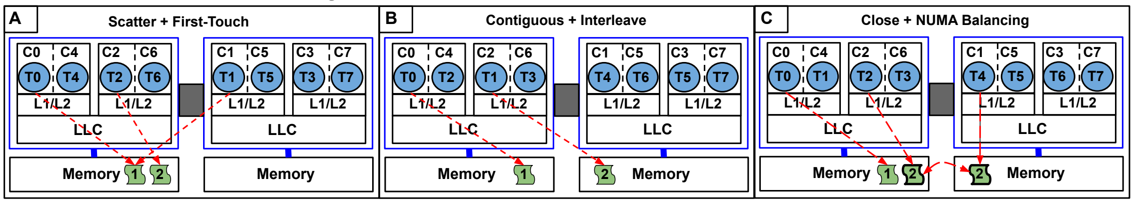
Also called thread placement, thread mapping (TM) is the process of assigning threads to cores (DIENER et al., 2016). It can specify a fine-grain mapping, defining a thread-to-core allocation, or a coarse-grain mapping, indicating only the NUMA node where the threads must run (i.e., threads can run in any core of that node). Taking again the Fig 2.8 as an example, it shows a single thread **T0** that is mapped to core **C0** of the example NUMA system.

The Linux Operating System (OS) uses the *Completely Fair Scheduler* (CFS) to perform thread mapping, which aims at fairness (to share running time to all tasks) and resource usage balance (LINUX, 2021). The CFS allocates threads in a round-robin way across the NUMA nodes (similar to the *Scatter* policy explained next), which may not be the best one for many applications since it can assign threads that likely share data far from each other. Moreover, to save power, CFS can map a group of idle threads to the same core, and when they become active, CFS migrates them to different cores. Even though it may be beneficial w.r.t. system utilization, migrating threads among the cores requires context copying, which may become very expensive when performed several times in a time slice. As an alternative, other thread mapping policies in the literature consider the execution of several parallel applications to perform the placements (DIENER et al., 2015; SCHWARZROCK et al., 2020). In Figure 2.9, we illustrate the thread-to-core assignment generated by 3 different thread mapping policies (*Scatter*, *Contiguous*, and *Close*) when executing an example application comprised of 8 threads (blue circles, where T# refers to thread #, for example) on the NUMA system illustrated in Figure 2.8. We describe each thread mapping policy below.

- ***Scatter.*** This thread mapping policy focuses on load balancing and reducing cache contention. It allocates the threads in a round-robin way, distributing them across the nodes as evenly as possible, first in the physical cores and then the logical cores. As shown in Fig. 2.9A, while the first 4 threads are allocated to physical cores threads (T0 → C0, T1→ C1, T2 → C2, T3 → C3), the remaining threads are assigned to the logical ones (T4 → C4, T5 → C5, T6 → C6, T7 → C7);
- ***Contiguous.*** It aims to improve the locality of memory accesses. Threads are placed continuously across a node's cores by placing neighboring threads (considering the threads' ID) on neighboring cores (i.e., first in the physical cores and after in the logical cores). As shown in Fig. 2.9B, while the first 4 threads are allocated to the cores in the first NUMA node (e.g., T0 → C0, T1→ C2, T2 → C4, T3 → C6), the remaining threads are assigned to the second NUMA node (e.g., T4 → C1, T5 → C3, T6 → C5, T7 → C7);
- ***Close.*** This policy aims to improve cache utilization. The neighboring threads are allocated on the same core, i.e., the physical and the logical cores. For example, in Fig. 2.9C), each pair of consecutive thread tend to share some level of cache memory (e.g., T0 and T1 shares L1/L2 and T0 and T2 shares LLC).

Controlling Thread Mapping. Configuring the thread mapping policies on the

Figure 2.9: Examples of A) *Scatter* + *First-Touch*, B) *Contiguous* + *Interleave*, and C) *Close* + *NUMA Balancing*.



Source: The author.

Linux OS can be done in different ways. This section explains tools capable of setting any of the above standard thread mapping policies or another user-defined mapping configuration (i.e., a solution based on the user expertise of a specific parallel application) – we will explore both in this work.

For OpenMP applications, the OpenMP runtime library provides some options to set TM policies, as explained above, through environment variables. One can set standard thread mapping policies through the variable `OMP_PROC_BIND`, such as *Close* and *Scatter*. The same policies can be set inside the parallel code through the `proc_bind` clause. On top of that, in the GNU implementation of the OpenMP, the `GOMP_CPU_AFFINITY` environment variable can be used to set the thread-to-core affinities. Thus, one can set the policies by specifying the placement according to the system topology. Notice that other user-defined thread mappings can also be defined through the `GOMP_CPU_AFFINITY` environment variable. Similarly, Intel's implementation of the OpenMP offers this functionality through `KMP_AFFINITY`.

For parallel applications implemented with other shared memory libraries (e.g., Pthreads implementations), one can use the "Linux-portable" methods *taskset* or *numactl* tool (KLEEN, 2004), as explained below.

- ***taskset.*** As an example, one can launch an `app.x` as `taskset -c 0,2,4,6 app.x` specifying that the application must run on cores 0,2,4,6 (any core of Node 0 in Fig. 2.8). However, it is not a strict thread-to-core assignment, so the threads can execute on any core and even share the same cores, leaving other cores idle. To take control of the thread-to-core allocations, one must identify the threads' ID (using `ps` or `top` commands after firing the application execution) and use the `taskset` command to set the affinities specifying the ID of the thread and the core to assign it to. For example, the command `taskset -p -c 0 1234` assigns a thread with ID 1234 to core 0;
- ***numactl.*** Since the numactl tool is aware of the processor topology and how the CPU cores map to CPU sockets, one is capable of setting the affinity on a coarser-grained basis (i.e., CPU sockets rather than individual CPU cores) than *taskset* (only CPU cores). To specify the same set of cores as shown in the *taskset* example above (i.e., to specify the Node 0 of the NUMA system of Fig. 2.8), one can use `numactl -cpunodebind=0 app.x`. However, the same problem to set the thread-to-core affinity happens here. Thus, one also must identify the threads' ID (using `ps` or `top` commands during the application execution) and use the `numactl` command. For

example, using `numactl -physcpubind=0 1234` to map a thread with ID 1234 to core 0.

2.4.2 Data Mapping

Also called memory page placement, data mapping (PM – page mapping) is the assignment of pages to memory controllers, i.e., it defines which NUMA node each memory page will be allocated. For example, Fig 2.8 shows the placement of two pages (1 and 2 in green) in two different NUMA nodes. The literature has different mapping policies, each differing in their optimization objectives (DIENER et al., 2016). *First-Touch* is the default page mapping policy in many OS, including Linux. Next, we describe such a policy and two other widely used policies available on the Linux OS. In addition to thread mapping policies, we also show in Figure 2.9 the data mapping policies, where the numbered green figures represent the application's data/pages and the red arrows indicate which threads are accessing them (and in which memory).

- ***First-Touch.*** As mentioned before, it is the default data mapping policy on the Linux OS. This policy places the data on the NUMA node where the thread that performs the first access to the data is running (DIENER et al., 2016). *First-Touch* assumes that the first thread to access a data will perform the most accesses to it, as shown in Figure 2.9A;
- ***Interleave.*** This policy aims at maximizing load balance by distributing the pages as evenly as possible among the nodes, balancing the load on the memory controllers, and increasing memory system throughput. As shown in Figure 2.9B, even though page 1 and page 2 are accessed by threads running on the same NUMA node, they are mapped to different memories;
- ***NUMA Balancing.*** The above mapping policies are static, so the pages are allocated and remain in the same NUMA node until the execution ends, not considering the applications' dynamic access behavior. On the other hand, Linux's *NUMA Balancing* tool is dynamic since it adapts the data mapping as the application executes (CORBET, 2012). The NUMA Balancing policy attempts to adapt the page mapping to changes in the memory access behavior by migrating memory pages to the node where the thread accessing that page is running whenever a page fault occurs. For example, as shown in Figure 2.9C, page 2 was initially placed on one memory

accessed by T2, but it had to change to another memory when T4 accessed it.

Controlling Data Mapping. Configuring the above standard data mapping policies is performed at the OS level, regardless of which shared memory library the application implements (e.g., OpenMP and Pthreads). As mentioned, the *First-Touch* is the default data mapping policy in Linux OS. To set the *Interleave* policy, one must use the `numactl` tool specifying the parameter `-interleave=all` (KLEEN, 2004). Also, one can enable the *NUMA Balancing* feature by configuring a kernel parameter, specifying Linux's file `/proc/sys/kernel/numa_balancing` to 0 or 1 to disable and enable *NUMA Balancing*, respectively.

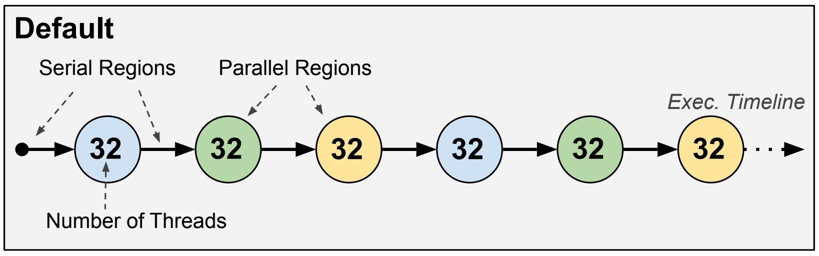
2.4.3 Thread Throttling

Thread Throttling is a technique to optimize the parallel application execution by adjusting the number of threads (NT). It is an effective technique for many parallel applications that have sub-linear scalability (CHADHA; MAHLKE; NARAYANASAMY, 2012; LORENZON et al., 2018) so that reducing the number of threads to execute such applications brings performance benefits over the default parallel execution, which uses the maximum number of threads (SULEMAN; QURESHI; PATT, 2008). The parallel scalability limitations are related to both software and hardware issues (LORENZON et al., 2018), such as:

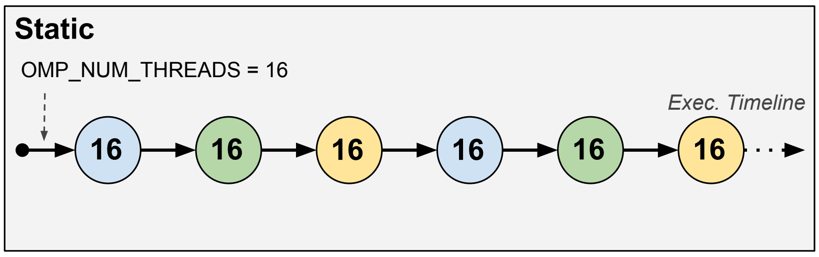
- *Saturation of functional units.* Although SMT can benefit applications with low instruction level parallelism (ILP), it may result in extra idle cycles if an individual thread presents enough ILP to issue instructions in most of the core's functional units;
- *Off-chip bus saturation.* The off-chip bus bandwidth is limited by the number of I/O pins, which does not increase as the number of cores increases. Thus, when it saturates, there are no further improvements by increasing the number of threads;
- *Overhead of data synchronization.* The overhead of threads' communication through shared regions on caches and main memories may also become a bottleneck;
- *Number of accesses of shared memory.* As a single thread must execute a critical section (region of code) at once when the number of threads increases, more threads must be serialized inside the critical sections, increasing the synchronization time.

Figure 2.10: Timelines of an parallel application running on a system with 32 hardware threads. The application has three parallel regions (identified by circles with different colors). Each number inside the circles indicates the number of threads used to execute the parallel region.

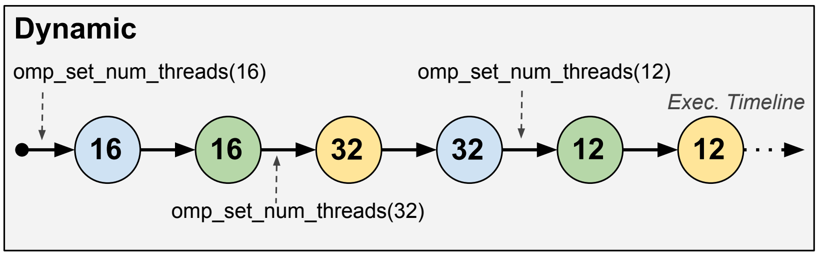
(a) Default execution of a parallel application.



(b) Static thread throttling.



(c) Dynamic thread throttling.



Source: The author.

We illustrate in Fig. 2.10 different timelines representing different ways to set the number of threads of a parallel application executing on a multicore with 32 hardware threads, where lines and circles represent serial and parallel regions, respectively. The application is represented by three different parallel regions, each with a different color. The number inside each circle indicates the number of threads used to execute that specific parallel region. Fig. 2.9a depicts the *Default* parallel execution, where every parallel region uses the maximum number of threads available on the system.

Adjusting the number of threads can be applied Statically or Dynamically. Static thread throttling, as illustrated in Fig. 2.9b, consists in setting the specific number of threads at the beginning of the application's execution and keeping the same value during the entire execution, i.e., it uses the same number of threads for all parallel regions.

Dynamic thread throttling, on the other hand, adapts the number of threads based on the behavior of each parallel region, so it adjusts the number of threads for each of them as the application executes, as illustrated in Fig. 2.9c.

Controlling the Number of Threads. One can control the number of threads of a parallel application through an application programming interface (API). In the case of OpenMP applications, Static thread throttling can be defined by setting the `OMP_NUM_THREADS` environment variable before the application execution so all parallel regions will execute with the same number of threads. However, for the Dynamic thread throttling, as OpenMP does not allow changing the number of threads using environment variables during execution, one has to use the `omp_set_num_threads()` routine or the `num_thread` clause of a parallel directive inside the application source code. When no value is set for such a routine or clause, the parallel region is executed with the maximum number of threads (the default configuration).

3 RELATED WORK

In this chapter, we present the related work concerning *(i)* the works that enhance generic parallel applications by optimizing thread and data mapping on NUMA systems, *(ii)* the works that adjust the number of threads of generic parallel applications, and *(iii)* the graph processing frameworks that focus their optimization on NUMA systems.

3.1 Thread and Data Mapping on NUMA Systems

Thread Mapping. Some approaches in the literature place threads according to applications' communication behavior. However, there is no direct and easy way to identify the communication among the application's threads on shared memory systems since they communicate implicitly, i.e., the communication happens when threads share data at the same time slice. For that, most of them collect statistics from the application execution, taking optimization decisions either offline (before firing the application execution) or online (during the application execution). Such statistics-based thread mapping mechanisms work by (*i*) detecting the threads' communication behavior and (*ii*) computing the thread-to-core placement using the communication behavior and the target hardware topology. After that, such approaches place or migrate the threads accordingly.

In this context, the work of Tam, Azimi and Stumm (2007) proposed an online scheme to cluster threads based on their data-sharing pattern and place each cluster into the same NUMA node. Their proposal monitors stall cycles and data cache misses to determine the communication pattern. Cruz, Diener and Navaux (2012) proposed a mechanism to detect threads' communication by monitoring information from the Translation Lookaside Buffer (TLB). The proposed mechanism outputs a communication matrix, which is used to compute the thread mapping through a heuristic method based on an algorithm for the graph-matching problem. A later work (CRUZ; DIENER; NAVAUX, 2015) inherited their idea to propose an online mechanism that migrates pages using the mapping strategy they have proposed before. Similarly, Diener, Cruz and Navaux (2013) proposed an online mechanism that allocates threads with a high amount of communication between them to neighbor cores. For that, it detects the communication between threads by monitoring page table accesses and uses a heuristic based on an algorithm for the graph-matching problem to define the thread-to-core allocation.

Along with the threads' communication behavior, the works of Jeannot, Mercier and Tessier (2013) and Cruz et al. (2015) also considered information on the target hardware topology to propose algorithms to compute the thread-to-core placement. The algorithms aim to maximize the data access locality by placing threads that share data into cores closer to each other. Jeannot, Mercier and Tessier (2013) and Cruz et al. (2015) have evaluated their proposal when employing it in an offline mechanism, but it can also be applied to online approaches.

There are also works that used the traditional thread mapping policies (as described in section 2) to improve the performance of parallel applications. For example, Papadimitriou, Chatzidimitriou and Gizopoulos (2019) have used such policies along with voltage and clock frequency adjustment to save energy on ARMv8 servers.

Data Mapping. Similar to the above-presented works, there are also statistics-based data mapping proposals in the literature, either offline or online. The offline strategies collect the memory access behavior and compute the page mapping before application execution, and then apply the data mapping when the application starts (MARATHE; THAKKAR; MUELLER, 2010; DIENER; CRUZ; NAVAUX, 2015). The online ones perform the mapping and data migration during the application execution (DASHTI et al., 2013; GUREYA et al., 2020a).

Marathe, Thakkar and Mueller (2010) propose a method to maximize local memory accesses by allocating memory pages to the node with the most accesses. The work of Diener, Cruz and Navaux (2015) proposed two new data mapping policies: *Balance* and *Mixed*. While the *Balance* leads to load-balance between memory controllers by considering the number of access to each page, *Mixed* focuses on combining memory access locality and page interleaving. Dashti et al. (2013) proposed *Carrefour*, an online data mapping algorithm that replicates or migrates data to maximize memory bandwidth utilization. It decides the data placement dynamically based on global observations of traffic congestion and access patterns of individual pages. Based on that, *Carrefour* places some data to maximize locality and interleaves the others. The work of Gureya et al. (2020a) proposed an asymmetric data placement mechanism that interleaves the pages based on a weighted distribution to maximize bandwidth. The proposal incrementally migrates data until an appropriate page distribution for the application is found during the application execution. Although it is an online approach, calibrating the target system topology bandwidth model requires an offline phase.

Thread and Data Mapping Together. The above-presented works assume the thread affinity knowledge as prior information and place the data according to it (DASHTI et al., 2013; DIENER; CRUZ; NAVAUX, 2015). However, the benefits of data mapping, especially for those statistic-based approaches, depend on thread mapping since the best local for memory pages depends on where the threads that access them the most are running. In this context, some works have proposed memory tracer tools for analyzing the memory access behavior of parallel applications, such as *Numalize* (DIENER et al., 2015), *Tabarnac* (BENIAMINE et al., 2015), and *NumaMMA* (TRAHAY et al., 2018). These tools support offline optimizations by providing information that helps with thread placement and data placement decisions.

Numalize (DIENER et al., 2015) is an offline tool that provides information about threads' communication through a communication matrix that gives the amount of communication between each pair of threads. For that, it considers, as a communication event, each access to the same cache line within the same time slice. Since *Numalize* collects the memory trace by a fine-grained level, it incurs an overhead of about $10\times$ the application execution time. It also provides page usage statistics – the number of accesses each page received from each thread. The other two tools, *Tabarnac* (BENIAMINE et al., 2015) and *NumaMMA* (TRAHAY et al., 2018), offer a graphical environment that allows the developer to understand the memory access patterns and, based on that, fix performance issues.

Based on the communication matrix generated as output of the *Numalize* tool, Diener et al. (2015) proposed an offline approach to place threads that share data on cores that shared caches and pages on the NUMA node with the most access to them. It also performs threads and data migrations. For that, it uses *Numalize* to collect information at each time slice, and with that, it can provide the mappings for each application phase.

Other statistic-based approaches have also been proposed to provide online thread and data mapping/migrations for parallel application execution. In this context, Broquedis et al. (2010a) and Broquedis et al. (2010b) proposed *ForestGOMP*, an extension to the OpenMP runtime environment. *ForestGOMP* decides thread and page migration based on memory affinities (given by the programmer) and hardware counters. With that, it schedules threads to cores to maximize cache memory reuse and maps the data to reduce remote memory accesses. The work of Diener et al. (2014) proposed kMAF, an OS-level tool that aims to improve data locality by adjusting thread and page mapping. kMAF analyzes page faults to identify memory access behavior. Based on that, it decides the optimized mappings and migrates threads and memory pages. Finally, Lepers, Quéma and Fedorova (2015) proposed *AsymSched*, a runtime algorithm that migrates threads and pages to maximize memory bandwidth. *AsymSched* considers asymmetric interconnect present in some NUMA systems to make migration decisions.

3.2 Thread Throttling

Static Approaches. A few studies have proposed to find a single number of threads that optimizes the entire application execution. The methods introduced by Pusukuri, Gupta and Bhuyan (2011) and by Sensi (2016) serve as examples of such approaches. Thread Reinforcer (PUSUKURI; GUPTA; BHUYAN, 2011) entails running the application multiple times over a short interval, varying the number of threads to identify the optimal configuration. It relies on OS-level data to guide this exploration. The approach detailed by Sensi (2016) employs a multiple linear regression model to predict the ideal combination of thread count and CPU frequency level for achieving optimal performance and power efficiency.

Some static methods opt for a particular configuration for each parallel region, considering their different requirements. An example of such an approach is NuCore, a model to predict the ideal number of threads for each parallel region to maximize bandwidth usage while minimizing the load on cores of different NUMA nodes. The model considers scalability concerns related to bandwidth saturation but does not address the performance drawbacks resulting from cache contention and data synchronization in parallel applications.

Dynamic Approaches. The strategies described here are dynamic, operating entirely during runtime. As they make thread count decisions during the application execution, they focus on optimizing each application's phase individually. These approaches mainly vary in their learning methods or determining the optimal number of threads: whether they predict a solution or evaluate different solutions to converge to the best one.

The following studies employ prediction and estimation techniques. In Suleman, Qureshi and Patt (2008), the FDT framework is introduced, which dynamically predicts the optimal number of threads at runtime, considering data synchronization and bus bandwidth saturation issues. Varuna (SRIDHARAN; GUPTA; SOHI, 2014) consists of an analytical engine that continually monitors system changes to determine the optimal level of parallelism, along with a manager that oversees the execution to align with the predefined level of parallelism. In the work by Shafik et al. (2015), an adaptive and scalable energy-saving model for OpenMP programs is proposed. This model leverages DCT and dynamic voltage and frequency scaling (DVFS) and involves two steps: (*i*) the programmer inserts code notations in the code to enable energy saving with specified performance requirements, and (*ii*) the runtime system reads these requirements and uses the information to guide energy saving. Finally, in Sensi, Torquati and Danelutto (2016), Nornir is introduced, which is an algorithm aimed at determining a configuration encompassing the number of threads, CPU frequency, and thread placement policy to meet specific bounds, either on performance or power consumption.

The runtime systems introduced by Chadha, Mahlke and Narayanasamy (2012) and Porterfield et al. (2013) implement thread count adjustments based on runtime observations. In the case of the LIMO runtime system proposed by Chadha, Mahlke and Narayanasamy (2012), it reduces the number of active threads to prevent cache saturation or when a thread is suspended. Additionally, LIMO employs DVFS to increase the frequency of active cores when a few threads are executing. The runtime system outlined by Porterfield et al. (2013) incorporates dynamic cache throttling (DCT) and CPU duty-cycle techniques to conserve energy. By monitoring hardware performance counters, this system decides when to modify the configuration, and such changes are implemented during the next available opportunity, typically in the subsequent parallel region execution.

The following studies employ methods to evaluate multiple solutions during application execution to converge to an optimal solution: Conductor by Marathe et al. (2015) is a runtime system designed to enhance performance within power constraints for OpenMP/MPI applications. It requires code modifications to insert functions into the applications. Conductor parallelizes the search by distributing various thread counts and CPU frequency configurations among MPI processes to accelerate the online search. In the work of Li and Martinez (2006), a hill-climbing heuristic and a linear search are used to identify the thread count and DVFS level configuration that optimizes power while meeting performance constraints. Various libraries proposed in Lorenzon, Souza and Beck (2017), Schwarzrock et al. (2017), Lorenzon et al. (2018), Alessi et al. (2015) employ a hill-climbing-based algorithm to determine specific configurations for each parallel region during application execution. The focus of these libraries varies, with some directed at Intel and AMD processors (LORENZON et al., 2018) and others targeting ARM processors (SCHWARZROCK et al., 2017). Alessi's library also fine-tunes the CPU operating frequency and the number of threads. The framework introduced by Bari et al. (2016) optimizes OpenMP parallel applications within specified power constraints by adjusting the number of threads, scheduling policies, and chunk sizes for each parallel region. The framework identifies configurations using a Nelder-Mead search algorithm. As proposed by Oliveira (2019), Odin utilizes a Fibonacci-based algorithm to reduce the search space and discover the best configuration for the number of threads and CPU frequency during runtime.

Hybrid Approaches. Some approaches follow a hybrid learning strategy so that they determine the optimal thread count during application execution but require prior knowledge of the application or undergoing an offline training phase specific to the target system: In the work by Jung et al. (2005), a performance model for SMT multiprocessor architectures is introduced to identify the optimal number of threads for each parallel loop. This technique includes a compiler time phase for identifying parallel regions with sub-optimal scalability. Thread Tailor, proposed by Lee et al. (2010), is a dynamic compiler that reduces active threads by combining them to minimize communication and synchronization. It incorporates an offline phase to analyze the inter-thread relationships within each application. The runtime system developed by Curtis-Maury et al. (2006, 2008) predicts the optimal configuration for various dimensions of parallelism, such as the number of multicore processors, cores within processors, and threads within cores. It adapts this configuration for each parallel region. Their approach features an offline phase for constructing prediction models and calibrating coefficients, a process performed once for the specific target system. In the work by Li et al. (2010), an instrumentation library for hybrid MPI/OpenMP applications is proposed to optimize individual OpenMP regions. The library includes an offline phase for training a model used at runtime to estimate the suitable thread count and CPU frequency configuration.

Most of the works discussed in this section ignore the influence of thread and page placement on application performance, particularly when running on NUMA systems. However, a few proposals address thread placement in addition to adapting the number of threads: Nornir by Sensi, Torquati and Danelutto (2016) considers thread placement and deploys threads based on traditional placement policies; NuCore by Wang, Davidson and Soffa (2016) defines the number of active cores per NUMA node, which can be considered a coarse-grain form of thread placement. However, neither of the works mentioned address data mapping.

In summary, the works discussed so far typically focus on either thread and data mapping or thread throttling individually. The following section will describe works that couple thread and page mapping with thread throttling.

3.3 Thread Throttling along with Thread Mapping and Page Mapping

Only the studies proposed by Popov, Jimborean and Black-Schaffer (2019) and Schwarzrock et al. (2020) have explored the potential of fine-tuning NT+TM+PM for generic applications. Both are offline strategies: one employs a comprehensive search across all possible NT+TM+PM combinations (POPOV; JIMBOREAN; BLACK-SCHAFFER, 2019), while the other optimizes each parameter separately along defined sequences (SCHWARZROCK et al., 2020).

Popov, Jimborean and Black-Schaffer (2019) is the first attempt to integrate the optimization of thread and page mapping with thread throttling. They introduce an offline approach for finding the best configuration through an exhaustive search that evaluates all possible combinations of such variables. To accelerate the search process, they employ a technique called *codeletes*, which involves running representative portions of the application's execution to assess each configuration. However, the search procedure can still be laborious despite the efficiency gains of using *codeletes*. This is due to the inclusion of data mapping policies that require information of the application's page access behavior for computing the page placement. To gather the necessary data for this purpose, the authors profile each application using the Numalize tool, which comes with a significant execution time overhead, approximately 10×, that causes a high overhead for the developer or user since each possible thread count requires a new profile.

Schwarzrock et al. (2020) evaluated the search space of tuning thread mapping (TM), page mapping (PM), and number of threads (NT) to identify optimization paths able to significantly reduce the search space (in at least 70% and up to 86%) while delivering results 5% different from the optimal one. Their evaluation found that the best path depends on the application being optimized, but optimizing NT→TM→PM (e.i., NT, TM, and PM, in this sequence), TM→NT→PM, and TM→PM→NT are the most promising paths.

3.4 Graph Analytics Optimization

Overview of Graph Optimization Proposals. Since 2004, more than 80 graph processing works have been proposed from both academia and industry (BATARFI et al., 2015; YAN et al., 2017; HEIDARI et al., 2018), including specific graph algorithms targeting specific platforms (HARISH; NARAYANAN, 2007; HONG et al., 2011), Graph DataBase Management Systems (HOLZSCHUHER; PEINL, 2013; MCCOLL et al., 2014), domain-specific languages (DSL) (SUJEETH et al., 2011; HONG et al., 2012), libraries (GREGOR; LUMSDAINE, 2005; BULUÇ; GILBERT, 2011), and frameworks (STUTZ; BERNSTEIN; COHEN, 2010; KYROLA; BLELLOCH; GUESTRIN, 2012; LOW et al., 2014), being the graph processing frameworks the most proposed ones. From 2010 onward, graph processing frameworks, such as Pregel introduced by Google (MALEWICZ et al., 2010), became very popular. They usually offer DSLs as the front-end (i.e., programming interfaces) and compile to a lower-level execution. For example, graph DSLs such as OptiML (SUJEETH et al., 2011) and Green-Marl (HONG et al., 2012) target graph processing frameworks, offering a more natural programming interface for users familiar with the graph domain but not with programming in general.

Signal/Collect (STUTZ; BERNSTEIN; COHEN, 2010), GraphChi (KYROLA; BLELLOCH; GUESTRIN, 2012), and GraphLab (LOW et al., 2014) were the first graph processing frameworks to be implemented for shared memory environments, enabling graph processing on consumer computers. Since then, several other graph processing frameworks have been proposed over the past years (SHUN; BLELLOCH, 2013; NGUYEN; LENHARTH; PINGALI, 2013; ZHANG; CHEN; CHEN, 2015; ZHU et al., 2016; ZHANG et al., 2018). They mainly focus on increasing the data locality by applying some sort of preprocessing in the graph layout or changing the algorithm computation during its execution. Moreover, they usually offer programming interfaces allowing users to write the algorithms' computation while automatically leveraging hardware properties such as data locality and system efficiency.

Graph Processing on NUMA Machines. Despite many graph processing frameworks in the literature, most are NUMA-oblivious, so their optimizations may not be effective for modern NUMA-based machines. Next, we explain the existing works.

The work of Agarwal et al. (2010) investigates the execution of the Breadth-First-Search (BFS) on NUMA systems to propose a new BFS implementation for large-scale graph processing. In the proposed BFS, the authors organize the computation around work queues spread over multiple sockets. Moreover, they use efficient spinning locks and lock-free channels to synchronize threads, and they also introduce peephole optimizations, e.g., to avoid atomic operations by verifying if they will fail. With that, the authors combine a high-level BFS design that captures the machine-independent aspects (ensuring portability with performance to next-generation processors) with an implementation that embeds processor-specific optimizations.

Frasca, Madduri and Raghavan (2012) proposed a NUMA-aware Betweenness Centrality (BC) implementation. The proposed BC uses *(i)* an Adaptive Data Layout (ADL) that observes parallel memory access behavior and dynamically reorders the graph to improve data locality and *(ii)* a NUMA-aware task scheduler that encourages better task to thread assignment while considering the NUMA distances between threads. The authors show that their dynamic design adapts to hardware topology and dramatically improves energy and performance.

The work of Zhang, Chen and Chen (2015) investigated the NUMA characteristics and their impact on the efficiency of graph analytics. Although the conventional wisdom is that remote memory accesses have higher latency and lower throughput than local ones, Zhang, Chen and Chen (2015) have shown that sequential remote accesses have much higher bandwidth than both random local and random remote ones (2.92$\times$ and 6.85$\times$). Further, they also showed that either interleaved or centralized allocation of graph data on existing graph analytics frameworks causes poor data locality and limited parallelism. Based on the above observation, Zhang, Chen and Chen (2015) proposed Polymer, a NUMA-aware graph-analytics framework that aims to minimize both random and remote memory accesses by optimizing graph data layout and access strategies. For that, Polymer *(i)* differentially allocates and places topology data, application-defined data, and mutable runtime states of a graph system according to their access patterns to minimize remote accesses; and *(ii)* for some remaining random accesses, Polymer carefully converts random remote accesses into sequential remote accesses, by using lightweight replication of vertices across NUMA nodes.

Sun, Vandierendonck and Nikolopoulos (2017) investigated the adverse effects of NUMA-based graph partitions on the performance of graph analytics, including load imbalance, increase in work performed per vertex, and reduction in the density of the graph connections. They showed that these issues limit the applications' scalability for large partition counts (i.e., increasing the number of partitions reaches a tipping point, after which overheads quickly dominate performance gains). Based on that, Sun, Vandierendonck and Nikolopoulos (2017) proposed GraphGrind, a NUMA-aware graph analytics framework that addresses the limitations incurred by graph partitioning by providing a fair graph partition strategy and changing from different graph representation (e.g., CSR and CSC) during the application execution. GraphGrind is based on the Ligra API and extends from the Cilk programming language, enabling NUMA-aware scheduling and work stealing.

Krause et al. (2019) proposes NeMeSys, a NUMA-aware graph pattern processing engine that aims to improve graph algorithm execution by leveraging different partitioning strategies and applying Bloom filter-based messaging optimization. NeMeSys uses the concept of Near Memory Processing to limit the scope of each worker thread to memory domains, which are directly connected to their socket, precluding them from performing expensive remote accesses. NeMeSys focuses its optimization on graph pattern matching computation, which usually launches several queries from different source vertices. Thus, the NeMeSys mechanisms for graph partitioning consider an outgoing edge table that partitions the graphs based on the source vertices.

Inspired by the shared-nothing scale-out designs of distributed graph processing frameworks, the works of Aasawat, Reza and Ripeanu (2018) and Aasawat et al. (2020) proposed the Hybrid Graph processing engine for NUMA (HYGN): a graph processing engine that exploits the characteristics of the synchronous and asynchronous processing modes. Their solution partitions the graph and binds each partition to a NUMA node to maximize locality. Then, considering the algorithm, phase of execution, and graph topology, HYGN shifts from one processing mode to another during the application execution, harnessing conditions where each mode holds unique advantages. For example, when the frontier of active vertices is larger, HYGN takes advantage of remote traffic aggregation and fast sequential remote access, addressing the limitation of the asynchronous mode design (e.g., a large amount of expensive remote random access during computation).

Considering the context of NUMA-based machines with hybrid main memories, the work of Liu et al. (2021) proposes HNGraph, a graph processing framework for NUMA systems comprised of volatile (DRAMs) and non-volatile memories (NVM). Since accessing DRAM memories is cheaper than accessing NVM memories (even remote access to DRAM is cheaper than local access to NVM), HNGraph mainly focuses on improving performance by reducing random access to NVM nodes. For that, it performs a degree-aware partitioning strategy, which distributes high-degree and low-degree vertices to DRAM and NVM nodes, respectively. For the algorithm execution, HNGraph exploits two communication primitives: in DRAM nodes, HNGraph aggregates several accesses to NVM nodes and uses a message-passing communication to perform a single remote random NVM update; and in NVM nodes, HNGraph uses shared memory primitives to access remote DRAM directly. With that, HNGraph can mitigate the impact of high-latency NVM accesses on the performance of graph processing.

3.5 Contribution of This Thesis

While the works presented in sections 3.1, 3.2, and 3.3 have already shown the benefits of adjusting the thread/data mapping and the number of threads for generic parallel applications on NUMA systems, most depend entirely on the threads' communication behavior. Since graph processing algorithm execution experiences highly irregular communication patterns, such techniques may not benefit graph applications significantly. Because of that, in this section, we aim to highlight our main contributions concerning works that aim *to optimize graph execution on NUMA machines*. As we have already shown in section 3.4, in this niche of graph computation, only a few works are NUMA-aware (AGARWAL et al., 2010; FRASCA; MADDURI; RAGHAVAN, 2012; ZHANG; CHEN; CHEN, 2015; SUN; VANDIERENDONCK; NIKOLOPOULOS, 2017; KRAUSE et al., 2019; AASAWAT et al., 2020; LIU et al., 2021).

We summarize in Table 3.1, the works described in sections 3.1, 3.2, and 3.3, which optimize the thread/data mapping and the number of threads for generic parallel applications. We also summarize in Table 3.2, the works that optimize the execution of graph applications on NUMA machines, described in the end of section 3.4 (*Graph Processing on NUMA machines*). Both tables describe information of their tuning parameters, adaptability, if they require changing the source code, and the target APIs. In addition, Table 3.2 also includes the input graph information used for optimization. Next, we detail what each column refers to.

Parameters. It indicates what is optimized: *Thread/Data mapping* and/or *Number of Threads*;

Information *(only in Table 3.2).* It specifies the information gathered and used for decision-making during the optimization process. As one of our proposals (*Graph-Nroll*) considers the source vertex variation, we have a specific column (*Src Vertex*) to indicate those works that consider, in some way, which vertex the graph algorithms will start executing;

Adaptability. In this column, the field *Learning* indicates when the approaches learn the optimized solution: *Offline (OFF)* when they learn the solution before application execution, which requires profiling/monitoring previously the entire application to find the best solution; *Online (ON)* when they learn without any previous information, i.e., they have the entire learning process done during application execution using only data gathered during the execution; and *Hybrid (HY)* when they

Table 3.1: Characteristics of the strategies proposed for generic applications *We mark the values as "?" when the authors have given no (or unclear) information about that.*

Proposal	Parameters			Adaptability		No Code Changing	APIs
	Thread Mapping	Data Mapping	Number of Threads	Learning	Setting		
Tam, Azimi and Stumm (2007)	x			ON	DY	x	Java multithreading
Cruz, Diener and Navaux (2012, 2015)	x			ON	DY	x	OpenMP, Pthreads
Diener, Cruz and Navaux (2013)	x			ON	DY	x	Any
Jeannot, Mercier and Tessier (2013)	x			OFF	ST	x	MPI
Cruz et al. (2015)	x			OFF	DY	?	Any
Papadimitriou et al. (2019)	x			ON	DY	x	Multi-workload
Marathe, Thakkar and Mueller (2010)		x		OFF	ST		OpenMP
Dashti et al. (2013)		x		ON	DY	x	Any
Diener, Cruz and Navaux (2015)		x		OFF	ST		OpenMP, Pthreads
Gureya et al. (2020a)		x		HY	DY		Any
Broquedis et al. (2010a, 2010b)	x	x		ON	DY		OpenMP
Diener et al. (2014)	x	x		ON	DY	x	Any
Lepers, Quéma and Fedorova (2015)	x	x		ON	DY	x	Any
Diener et al. (2015)	x	x		OFF	DY		Any
Beniamine et al. (2015)	x	x		OFF	ST		?
Trahay et al. (2018)	x	x		OFF	ST		?
Pusukuri, Gupta and Bhuyan (2011)			x	OFF	ST	x	OpenMP, PThreads
Sensi (2016)			x	OFF	ST	x	OpenMP, PThreads
Jung et al. (2005)			x	HY	DY	x	OpenMP
Lee et al. (2010)			x	HY	DY		PThreads, MPI
Curtis-Maury et al. (2006, 2008)			x	HY	DY	x	OpenMP
Li et al. (2010)			x	HY	DY		MPI+OpenMP
Suleman, Qureshi and Patt (2008)			x	ON	DY	x	OpenMP
Sridharan, Gupta and Sohi (2014)			x	ON	DY		PThreads, TBB
Shafik et al. (2015)			x	ON	DY		OpenMP
Chadha et al. (2012)			x	ON	DY	x	OpenMP, PThreads
Porterfield et al. (2013)			x	ON	DY	x	OpenMP
Li and Martinez (2006)			x	ON	DY		OpenMP
Marathe et al. (2015)			x	ON	DY		MPI+OpenMP
Bari et al. (2016)			x	ON	DY	x	OpenMP
Alessi et al. (2015)			x	ON	DY		OpenMP
Lorenzon, Souza and Beck (2017)			x	ON	DY		OpenMP
Schwarzrock et al. (2017)			x	ON	DY		OpenMP
Lorenzon et al. (2018)			x	ON	DY	x	OpenMP
Oliveira (2019)			x	ON	DY	x	OpenMP
Wang, Davidson and Soffa (2016)	x		x	OFF	DY	?	Any shared mem.
Sensi, Torquati and Danelutto (2016)	x		x	ON	DY		OpenMP, PThreads
Popov et al. (2019)	x	x	x	OFF	DY	?	OpenMP
Schwarzrock et al. (2020)	x		x	OFF	ST	x	**OpenMP**
Graphith	**x**	**x**		**OFF**	**ST**	**x**	**OpenMP***
PredG	**x**	**x**		**OFF**	**ST**	**x**	**OpenMP***
GraphNroll	**x**	**x**		**OFF**	**ST**	**x**	**OpenMP***
PotiGraph	**x**	**x**	**x**	**OFF**	**ST**	**x**	**OpenMP***

Source: The author.

Table 3.2: Graph processing frameworks targeting NUMA systems. *We mark the values as "?" when the authors have given no (or unclear) information about that.*

Proposal	Parameters			Information		Adaptability		No Code Changing	APIs
	Thread Mapping	Data Mapping	Number of Threads	Decision-Making	Src Vertex	Learning	Setting		
Agarwal et al. (2010)				Number of NUMA nodes		OFF	ST	x	?
Frasca, Madduri and Raghavan (2012)				NUMA topology		HY	DY	x	OpenMP
Zhang, Chen and Chen (2015)		x		Partitioning: number of edges		HY	DY		Pthread, Cilk, OpenMP
Sun, Vandierendonck and Nikolopoulos (2017)		x		Execution modes: frontier size		HY	DY		Cilk
Krause et al. (2019)				Source vertices	x	OFF	ST	?	?
Aasawat et al. (2020)		x		Partitioning: vertices' degree Execution modes: frontier size		HY	DY	X	OpenMP
Liu et al. (2021)		x		Partitioning: vertices' degree		OFF	ST	?	?
Graphith	**x**	**x**		**Previous executions**		**OFF**	**ST**	**x**	**OpenMP***
PredG	**x**	**x**		**Graphs' features**		**OFF**	**ST**	**x**	**OpenMP***
GraphNroll	**x**	**x**		**Graphs' features**	**x**	**OFF**	**ST**	**x**	**OpenMP***
PotiGraph	**x**	**x**	**x**	**Graphs' features**		**OFF**	**ST**	**x**	**OpenMP***

Source: The author.

learn as the application executes but rely on some offline training or analysis;

The field *Setting* indicates when the approaches apply the optimized solution: *Static (ST)* when they set the optimized solution at the beginning of the execution, keeping the same configuration during the entire execution; and *Dynamic (DY)* when they set the solution while the application executes, which may change at run-time;

No Code Changing. In this column indicates the approaches that do not demand the software developer to apply any change in the source code;

APIs. The column *API* shows the parallel libraries supported by each referred work.

Our proposals focus on optimizing graph processing algorithms, which is not covered by any of the works in Table 3.1. Although our proposals learn and set the best solution before the application's execution (OFF learning and ST setting), *PredG*, *GraphNroll*, and *PotiGraph* leverage the characteristics of the input data to make decisions, providing adaptability for new input graphs or source vertices (in the case of *GraphNroll*). Therefore, our proposals do not incur any learning overhead during the application's execution after being deployed on the target systems. Moreover, they also cover the simultaneous adjustment of TM+PM (*Graphith*, *PredG*, and *GraphNroll*) and NT+TM+PM (*PotiGraph*), which is addressed by only a fraction of such works.

Regarding the works that optimize graph processing on NUMA machines (Table 3.2), they are:

(i) Not aware of the benefits that simultaneously optimizing NT+TM+PM for graph execution on NUMA system may bring to their proposals (AGARWAL et al., 2010; FRASCA; MADDURI; RAGHAVAN, 2012; ZHANG; CHEN; CHEN, 2015; SUN; VANDIERENDONCK; NIKOLOPOULOS, 2017; KRAUSE et al., 2019; AASAWAT et al., 2020; LIU et al., 2021);

(*ii*) Not suitable for different graph algorithms since they optimize specific algorithms, leveraging NUMA topology information that benefits each case (FRASCA; MADDURI; RAGHAVAN, 2012; AGARWAL et al., 2010);

(*iii*) Time-consuming as they require reordering or partitioning the graph's vertices before (AASAWAT et al., 2020; ZHANG; CHEN; CHEN, 2015; SUN; VANDIERENDONCK; NIKOLOPOULOS, 2017; KRAUSE et al., 2019) or during the application execution (FRASCA; MADDURI; RAGHAVAN, 2012) (which slowdowns execution) for each different graph being processed;

(*iv*) Requires the user to write or change the application code (ZHANG; CHEN; CHEN, 2015; SUN; VANDIERENDONCK; NIKOLOPOULOS, 2017);

(*v*) Based on improving the data locality by using offline (AASAWAT et al., 2020; ZHANG; CHEN; CHEN, 2015; SUN; VANDIERENDONCK; NIKOLOPOULOS, 2017; KRAUSE et al., 2019) or online (FRASCA; MADDURI; RAGHAVAN, 2012) strategies that suffer learning overheads before or during the application execution.

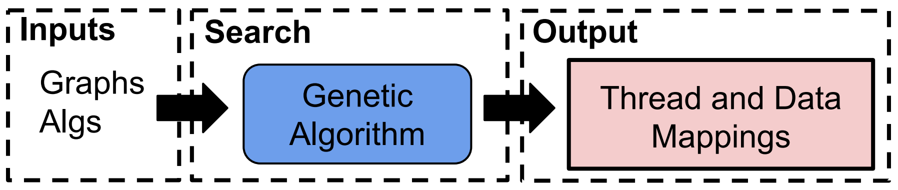
Contributions. To our knowledge, we are the first to propose strategies that exploit the input graphs' high-level characteristics to optimize execution (other than the number of vertices and edges, as shown in Table 3.2). With that, we can use a strategy similar to the offline ones but without any sort of profiling; and still adapt to different input data before execution, offering significant levels of adaptability without incurring any penalties presented by online methods. Therefore, by using a Machine Learning methodology to process the high-level graph features, we can accurately apply the right thread/data mapping policies and the best number of threads in NUMA systems when processing a new input graph, without any extra application execution, profiling, or the need for changing the application source code or the input graph structure. On top of that, our proposal is orthogonal to any graph processing framework presented in section 3.4 since it improves graph processing performance by selecting an optimized thread/data and/or number of threads for the execution without requiring any algorithm changes.

4 GRAPHITH: OPTIMIZING GRAPH EXECUTION ON NUMA MACHINES

The execution of large real-world graphs, such as web searches and social networks, has been boosted by modern HPC systems. However, their irregular communication patterns and poor data locality impose many challenges, mainly when executed on NUMA systems. As Fig. 1.2 of chapter 1 showed, adjusting the thread and data placements effectively improves the parallel graph application's performance on NUMA systems. In this chapter, *(i)* we perform a Design Space Exploration (DSE) to analyze the potential of thread and data mapping for graph applications. Based on that, we show that there is no one-fits-all solution for all input graphs, algorithms, and NUMA systems, i.e., no single combination of thread and data mapping that achieves the best performance if the input graph, algorithm, or NUMA system changes. *(ii)* We also show the complexity of finding the ideal solution for our tackled problem by presenting its relation to a well-known *NP*-Hard problem, the Quadratic Assignment Problem (QAP).

On top of the no-one-fits-all behavior, we also argue that there is still room for improvements when it comes to thread mapping: the traditional mapping policies available on Linux OS apply only specific deterministic rules (e.g., to locate the threads/data in a round-robin fashion or to locate neighboring threads in neighboring cores – see chapter 2). It limits the graph algorithms for further improvements since only a small part of the search space is evaluated. To overcome that, *(iii)* we propose ***Graphith***: a framework that improves graph processing performance by adapting its execution considering the NUMA system, graph algorithm, and a particular input graph. As illustrated in Fig 4.1, *Graphith (Inputs*) receives the algorithm and input graph to be optimized, *(Search)* it performs fine-tuning in the thread-to-core allocation, which evaluates the entire problem's search space, and *(Output)* gives the best solution of thread and data mapping found. With that, *Graphith* converges to high-quality solutions, further improving the existing TM and PM policies. In summary, the main contributions presented in this chapter are:

Figure 4.1: Graphith framework overview.



Source: The author.

- A DSE considering the combinations of several standard thread and data mapping policies;
- The correlation of the tackled problem with a well-known *NP*-Hard (the QAP), showing its complexity in finding the ideal thread and data mapping solution;
- *Graphith*: a framework to boost the graph processing algorithm performance on NUMA systems.

Results on two real NUMA systems composed of 2 and 4 nodes show that *Graphith* outperforms, on average, 21% and 7% the default execution and the best combination of TM+PM policies found through an exhaustive search of such policies.

4.1 Design Space Exploration

This section describes the DSE performed to analyze the potential for TM and PM on NUMA systems. Next, we present the considered input graphs, algorithms, machines, and the evaluated mapping policies.

Input Graphs. For the input graphs, we evaluated five representative real-world and synthetic graphs, which cover the two comprehensive classes of topologies (meshes and social networks) and different sizes (BEAMER; ASANOVIĆ; PATTERSON, 2015). The graphs are urand, kron, twitter, web, and road – the five largest ones presented in Table 2.1 of chapter 2.

Graph Algorithms. We used the GAP Benchmark Suite (GAPBS) (BEAMER; ASANOVIĆ; PATTERSON, 2015). GAP contains a collection of well-known and representative graph algorithms – as described in chapter 2. The benchmark provides high-performance implementations for each algorithm using the optimization strategy that is the most appropriate for each one. All algorithms are written in C++11 and parallelized with OpenMP. For our experiments, we compiled the applications with GNU g++ 10.1.0 and OpenMP 4.5, with the optimization flag -O3.

Execution Environment. We performed our experiments in two real NUMA systems using the Ubuntu OS with kernel v. 4.19:

- ***Intel32***: 2x 8-core Intel Xeon E5-2640v2 (Ivy Bridge) @2.0GHz, 2-way SMT (2 nodes / 16 cores / 32 threads). Each core has a 32KB L1 cache and 256KB L2 cache; and shares 2x20MB L3 cache and 2x64 GB of main memory;
- ***Intel64***: 4x 8-core Intel Xeon X7550 (Nehalem) @2.0 GHz, 2-way SMT (4 nodes

Table 4.1: Execution time variation by changing the thread and data mappings on *Intel32*, normalized by the **baseline**.

	PM	**Default**			**Interleave**				**NUMA Balancing**			
	TM	**Clo**	**Con**	**Sca**	**Def**	**Clo**	**Con**	**Sca**	**Def**	**Clo**	**Con**	**Sca**
BC	**kron**	0.98	1.00	**0.97**	0.98	1.00	1.00	0.99	1.01	1.04	0.99	0.98
	road	**0.99**	1.01	1.04	1.00	1.04	1.04	1.01	1.04	1.07	1.05	1.04
	twitter	1.04	1.01	0.97	1.03	1.03	1.03	0.97	1.01	1.03	1.02	**0.96**
	urand	1.02	1.01	**0.95**	1.01	1.02	1.03	0.99	1.02	1.05	1.01	1.00
	web	**0.98**	1.01	1.00	1.00	1.03	1.01	1.02	1.02	1.02	1.02	0.99
BFS	**kron**	1.03	1.01	1.01	1.01	1.03	1.06	1.03	1.12	1.12	1.07	**0.99**
	road	1.04	1.02	1.14	1.01	**0.93**	1.02	0.94	1.14	1.19	1.13	**0.93**
	twitter	0.96	0.97	0.99	0.98	0.96	**0.95**	0.98	1.09	1.19	1.14	0.97
	urand	**0.98**	**0.98**	1.04	0.99	1.04	1.09	1.10	1.08	1.08	1.05	1.04
	web	0.98	0.99	0.98	1.00	**0.90**	**0.90**	0.91	1.10	1.04	1.15	1.01
CC	**kron**	0.99	0.99	1.02	1.00	0.97	0.98	**0.96**	1.02	1.00	1.01	1.02
	road	1.06	1.06	1.02	1.01	1.01	1.01	1.01	1.01	1.11	1.07	1.01
	twitter	0.99	0.99	1.03	0.99	**0.92**	0.93	0.98	1.03	1.04	1.02	1.03
	urand	0.97	0.97	1.02	0.97	**0.96**	**0.96**	**0.96**	0.98	1.00	1.02	1.03
	web	1.00	1.01	1.02	0.98	**0.87**	**0.87**	0.98	1.01	1.03	1.03	1.00
PR	**kron**	1.00	1.00	1.01	0.99	0.92	**0.91**	0.93	1.02	1.47	1.04	1.02
	road	1.05	1.05	1.01	1.02	1.05	1.05	1.05	1.00	1.09	1.05	1.01
	twitter	**0.91**	0.92	1.02	1.00	**0.91**	**0.91**	**0.91**	1.06	0.94	0.94	1.02
	urand	0.87	0.88	0.89	0.88	**0.86**	0.89	0.91	0.99	0.99	0.90	0.99
	web	1.01	1.01	0.95	1.02	**0.85**	0.89	**0.85**	1.03	0.96	0.97	0.95
SSSP	**kron**	1.00	1.00	1.04	1.02	**0.86**	0.91	**0.86**	1.02	1.04	1.06	1.04
	road	1.00	1.00	1.01	1.01	1.01	1.00	1.01	1.01	1.05	1.02	1.01
	twitter	1.02	1.01	1.02	1.01	**0.96**	**0.96**	0.98	1.04	1.05	1.05	1.03
	urand	1.00	1.01	1.03	**0.91**	**0.91**	0.94	**0.91**	1.05	1.06	1.06	1.04
	web	1.00	1.00	0.99	1.00	0.97	0.98	**0.94**	1.05	1.05	1.05	0.99

Source: The author.

/ 32 cores / 64 threads). Each core has a 32KB L1 cache and 256KB L2 cache and shares 4x18MB L3 cache and 4x32 GB of main memory.

Thread and Data Mapping Policies. Our experiments considered the execution of the TM and PM policies explained in Chapter 2: *Linux's Default solution* (Def), *Close* (Clo), *Contiguous* (Con), and *Scatter* (Sca) for TM; and *First-Touch* (Def), *Interleave* (Int), and *NUMA Balancing* (NUM) for PM.

Results. In Tables 4.1 and 4.2, we show the execution time of the **TM** and **PM** policies combinations evaluated in this work (columns). Results are normalized to the baseline (so the lower, the better - for instance, 0.98 means that a given metric is better than the baseline in 2%) and grouped by graph algorithms and input graphs (rows). Also, we highlight the best combination for each row in **bold**.

By analyzing the results, we observed that the best solution changes according to:

- *Input graph.* For example, on *Intel64*, while BC-urand is better executed with the combination of *Default* TM and *NUMA Balancing* PM (Def/NUM with 31% improvement), the BC-kron has its best performance using *Contiguous* and *NUMA Balancing* (Con/NUM with 43% improvement);

Table 4.2: Execution time variation by changing the thread and data mappings on *Intel64*, normalized by the **baseline**.

	PM	**Default**			**Interleave**				**NUMA Balancing**			
	TM	**Clo**	**Con**	**Sca**	**Def**	**Clo**	**Con**	**Sca**	**Def**	**Clo**	**Con**	**Sca**
BC	**kron**	0.98	1.00	0.70	0.75	0.96	0.71	0.69	0.60	0.59	**0.57**	0.59
	road	1.07	1.06	1.04	1.03	1.06	1.06	1.07	1.05	1.12	1.08	1.08
	twitter	0.99	**0.97**	1.01	1.73	1.48	1.67	1.65	1.11	1.02	0.98	1.00
	urand	1.16	1.15	1.06	0.83	1.52	0.90	0.88	**0.69**	0.70	0.74	**0.69**
	web	0.97	1.01	1.03	1.09	0.99	1.16	1.08	**0.94**	0.97	0.95	0.98
BFS	**kron**	0.95	1.05	**0.82**	1.14	1.06	1.18	1.07	0.91	0.90	1.08	0.88
	road	1.32	1.33	1.38	1.40	1.24	1.06	1.36	1.30	1.16	1.04	1.35
	twitter	**0.91**	0.96	0.95	0.98	1.02	0.99	1.06	1.14	1.35	1.12	1.08
	urand	1.19	1.07	1.02	1.35	1.29	1.39	1.35	1.06	1.03	1.19	1.06
	web	0.91	0.98	0.95	1.03	**0.82**	1.14	0.98	1.05	1.01	1.07	1.01
CC	**kron**	1.02	1.01	1.00	0.91	**0.78**	1.03	0.88	0.98	1.02	1.05	0.97
	road	1.38	1.38	1.11	1.20	**0.90**	1.23	1.08	1.22	1.42	1.32	1.00
	twitter	0.98	1.00	0.95	0.71	**0.62**	0.75	0.63	1.04	0.96	1.04	0.98
	urand	1.03	1.05	1.00	0.86	**0.83**	0.87	0.84	1.01	1.06	1.08	0.99
	web	0.96	0.99	0.98	0.59	**0.47**	0.56	0.59	0.97	0.98	0.95	1.00
PR	**kron**	1.04	1.03	1.39	1.43	1.05	1.37	1.45	0.75	0.66	**0.59**	0.66
	road	1.06	1.09	1.01	1.35	1.16	1.21	1.28	1.09	1.13	1.19	1.06
	twitter	1.08	1.00	1.02	2.12	2.13	2.09	2.00	1.05	1.79	1.20	1.07
	urand	0.76	0.43	0.55	1.01	0.83	1.03	1.03	0.44	**0.38**	0.49	0.40
	web	0.99	1.00	0.95	0.93	**0.81**	1.06	0.90	0.94	1.00	0.87	0.93
SSSP	**kron**	0.93	0.99	1.23	1.00	0.95	**0.90**	0.91	0.95	0.95	0.95	0.97
	road	0.97	0.96	0.97	0.97	**0.94**	**0.94**	0.95	1.07	1.06	0.99	0.99
	twitter	1.04	1.05	1.01	**0.97**	**0.97**	0.99	**0.97**	1.05	1.09	1.12	1.05
	urand	**0.97**	**0.97**	1.02	1.15	1.06	**0.97**	1.02	1.03	1.01	1.02	1.02
	web	0.97	1.01	1.40	1.02	0.96	**0.94**	**0.94**	1.23	1.28	1.34	1.13

Source: The author.

- *Graph algorithm.* For example, also on *Intel64*, the BFS-kron is better executed with *Scatter* TM and *Default* PM (Sca/Def with 18% improvement), while CC-kron presets its best execution with *Close* TM and *Interleave* PM (Clo/Int);
- *NUMA machine.* For instance, when running the PR-urand on *Intel64*, the combination of *Close* TM and the *NUMA Balancing* PM (Clo/NUM) improves performance by 62% over the baseline (Def/Def). However, such improvement does not exist when changing the NUMA system (only 1% improvement on *Intel32*).

The observation above reinforces our discussion in the introduction that no ideal combination of TM and PM exists for all applications and graph structures since each graph algorithm performs a distinct computation. Some focus on the computation of the vertices' properties (BC and PR), while others traverse the graph vertices (BFS, CC, and SSSP). Additionally, several other characteristics of the algorithms and input graphs (see Tables 2.2 and 2.1) impose even more challenges in optimizing the performance of graph algorithms by adjusting the TM and PM policies.

4.2 Thread Mapping Problem as a QAP's Instance

So far, this thesis has addressed a practical problem of improving the graph processing algorithms execution by tuning the thread-to-core allocation and data placement. However, defining only the thread-to-core affinity is already a challenging problem since it can be seen as an instance of the Quadratic Assignment Problem (QAP), an NP-Hard problem (KNOWLES; CORNE, 2003; GAREY; JOHNSON, 2002). To highlight the theoretical complexity, we present the relationship between the **thread-to-core** allocation and the QAP. The QAP entails the assignment of n *facilities* to n *locations* aiming to minimize the *flow*×*distance product* given by Equation 4.1. It is given the problem size n, the flow $a_{i,j}$ exchanged between facilities i and j, and the distance b_{π_i,π_j} between locations π_i and π_j. π_i represents the location in which the facility i is allocated in a specific permutation $\pi \in P_n$. P_n is the set of all possible permutation of $\{1, 2, \dots, n\}$ (KNOWLES; CORNE, 2003). *Then, the objective of the problem is to find a permutation π in the set of all possible permutations of size n, P_n, that minimizes the objective function $minC(n)$*

$$minC(n) = min_{\pi \in P_n} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} \times b_{\pi_i,\pi_j} \tag{4.1}$$

To define a relationship between the QAP and the thread-to-core allocation, we need to correlate each element of them:

- ***Threads*** $\rightarrow$ ***Facilities*****:** Although each thread executes a set of graph's vertices, we can see them as individual entities. In this sense, each thread performs its own computation and eventually communicates with other threads to exchange data;
- ***Communication Between Threads*** $\rightarrow$ ***Flows*****:** Data exchanging between threads is done through shared memory accesses. The communication pattern highly depends on the graph's structure, the computation performed by the graph algorithms, and the vertices distribution throughout the threads;
- ***System's Cores*** $\rightarrow$ ***Locations*****:** The system's cores are the locations where the threads are assigned to execute. Hence, we can consider all resources available to execute the application's threads as the set of locations in our problem;
- ***NUMA Topology*** $\rightarrow$ ***Distances*****:** The distance between each core in a NUMA system is given by its node location.

Hence, theoretically, the thread-to-core allocation search space is the same as the QAP, requiring the evaluation of $n!$ solutions to perform an exact procedure. However, the problem addressed in this work may be even more challenging because we also consider data mapping optimization. Furthermore, TM and PM are highly correlated: if one changes the PM policy, the best TM policy may change, and vice versa.

Some of our experiments consider all the search space of thread-to-core allocation and data mapping policies variation in this work, requiring $n! \times d$ evaluations, where n is the number of threads/cores, and d is the number of data mapping policies considered. However, evaluating all the search space for a larger n is impractical, e.g., finding an exact solution for the QAP with a size greater than 30 is already impractical (RAMKUMAR et al., 2008). Therefore, metaheuristics such as Genetic Algorithm (GA) emerge as a suitable approach to dealing with the thread-to-core allocation and data mapping problem. The next section will show our proposal to tackle this problem.

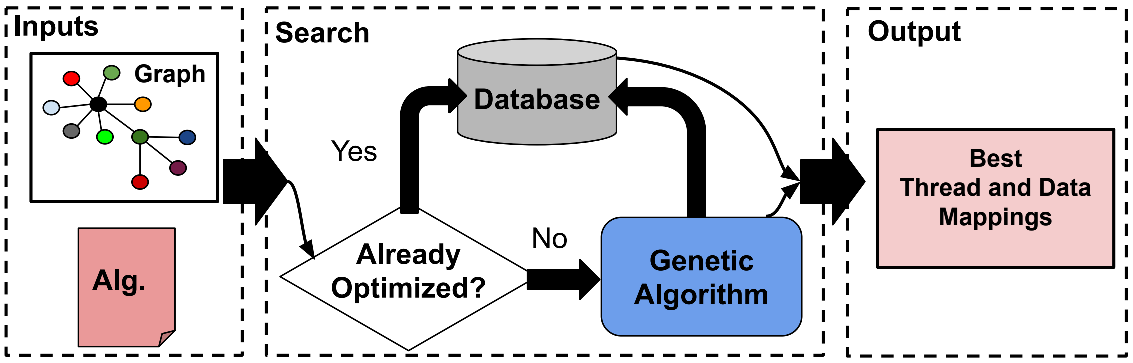
4.3 Graphith

To handle the above-presented problem, in this section, we present the ***Graphith*** framework, which consists of an offline method for finding a configuration of thread and data mappings to optimize the execution of a given graph processing algorithm running a specific input graph. *Graphith* framework is depicted in Fig. 4.2 and works as follows:

(i) The user provides the input graph and graph algorithm to be optimized;

(ii) Then, the *Graphith* applies a search for the best configuration of the thread and data mappings. For that, it verifies whether the given algorithm running the input graph was already optimized or not. If it is *true*, *Graphith* recovers the configuration stored in the **Database** and starts execution. Otherwise, *Graphith* applies a GA to perform a fine-tuning in the thread-to-core allocation and vary the data mapping

Figure 4.2: Graphith's optimization flow.



Source: The author.

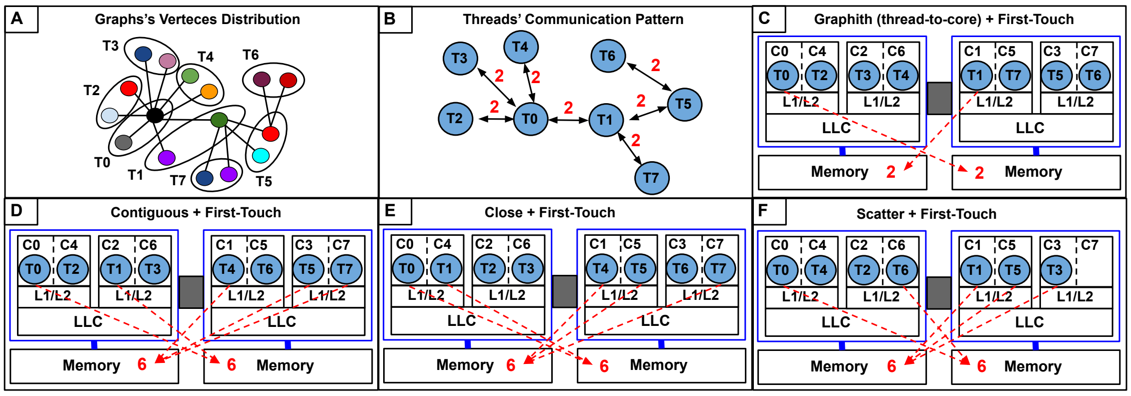
policies (it will be explained in section 4.3.1). Finally, *Graphith* stores the obtained solution in the Database for future reuse. Although *Graphith* uses a GA, this search part can be easily modified to support any other search strategies; and

(iii) The *Graphith*'s output is the best configuration found for that graph input/algorithm and will be used whenever that specific occurrence is called again.

How Graphith Goes Beyond OS Mappings. It is worth noticing that the standard policies limit further performance improvements as only a small part of the entire search space is evaluated: a total of 12 combinations of thread mapping and data mapping. As discussed in chapter 2, they apply only specific deterministic rules (e.g., mapping the threads/data in a round-robin fashion or placing neighboring threads in neighboring cores). Given that, *Graphith* is totally flexible w.r.t. thread-to-core allocations: it may potentially test any combination without following any specific rule in opposition to the TM policies available in the OS.

Therefore, *Graphith* extends the default design space by applying fine-tuning in thread-to-core allocations (considering all possible allocations of any thread to any core) and varying the PM policies available in the OS (see Figure 4.5 for a solution structure used in the *Graphith*'s optimization process, which we will explain later). Figure 4.3 compares a possible *Graphith* solution (Fig. 4.3C) with other TM policies (*Contiguous*, *Close*, and *Scatter*) combined with *First-Touch* PM. This figure considers the input graph in Fig. 4.3A being processed by an application with the communication pattern among its threads as illustrated in Fig. 4.3B, where numbers in red indicate the number of edges connecting the set of vertices processed by each thread. By fine-tuning the placement

Figure 4.3: Examples of A) graph's vertices distribution across threads, B) threads communication, C) *Graphith*'s solution, D) *Contiguous* + *First-Touch*, E) *Close* + *First-Touch*, and F) *Scatter* + *First-Touch*.



Source: The author.

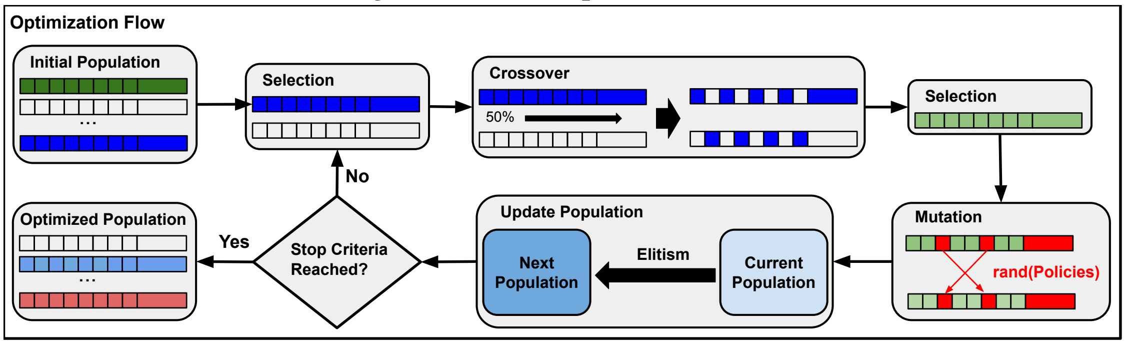
of threads, *Graphith* considers all possible thread-to-core combinations and can find a solution with only 4 remote memory accesses (arrows and numbers in red) against 12 presented by the other TM policies available in the OS.

4.3.1 Genetic Algorithm (GA)

As mentioned, *Graphith* uses a GA for learning the best TM and PM configuration. GA is an evolutionary strategy that uses the concept of population, i.e., a set of individuals (given by chromosomes that represent the tackled problem's solutions) that can evolve to an optimum solution through generations (GOLDBERG, 2000). In a GA, the best individuals (parents) are selected for reproduction to generate new individuals (offspring). Three basic operators are used to process the population (described later in this section): selection, crossover, and mutation. These operators, the fitness function (which indicates each individual's quality), and the algorithm's stop criterion must be suitable to solve the tackled problem. Next, we describe our proposed GA, as illustrated in Fig. 4.4.

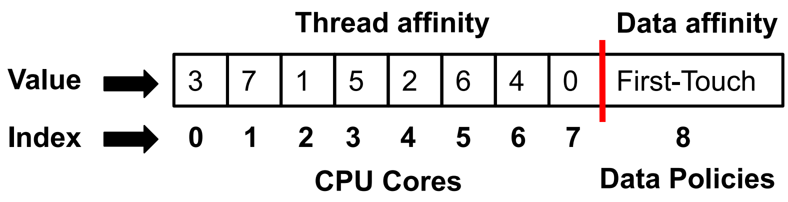
Optimization Flow. Our GA creates N random solutions (individuals) to compose the initial population. In each iteration (generation), the GA selects two parent solutions to perform an information exchange through the crossover operator (according to a probability ϕ, called crossover rate). The main idea of this step is to exchange the chromosomes' characteristics that can generate even better offspring. It can also be seen as an intensification procedure (or exploitation), combining two good solutions to generate an even better solution. Later, according to a probability θ (mutation rate), the GA selects a solution to perform the mutation operator. Mutation usually prevents the search from

Figure 4.4: GA's optimization flow.



Source: The author.

Figure 4.5: Chromosome representation.



Source: The author.

being stuck in a region of a local optimum. They are not performed if their respective probabilities are not satisfied in both crossover and mutation. At the end of each generation, the GA selects individuals to compose the next generation by an elitist procedure. Finally, if the algorithm reaches the stop criterion (we chose 50 generations, which is a good trade-off between the time and the algorithm's convergence), the optimized population is given as a result. Otherwise, the algorithm performs the next iteration.

Chromosome Representation. We represent the chromosome structure as a vector of n elements, as shown in Fig. 4.5. A chromosome has two pieces of information: the thread mapping and the data mapping policy. For the thread mapping part, indexes $[0 : n - 2]$ represent the application's threads, and the $[0 : n - 2]$ values represent the system's cores. For the data mapping part, the last index $(n - 1)$ value indicates the PM policy, e.g., *First-Touch*, *Interleave*, or *NUMA Balancing*.

We set the thread-to-core mapping with the *GOMP_CPU_AFFINITY* OpenMP environmental variable. For the data placement, the *First-Touch* and *NUMA Balancing* policies are available in Linux systems (DIENER et al., 2016). On the other hand, to interleave the data among the system's memories (applying the *Interleave* policy), we have used the Linux *numaclt* tool (KLEEN, 2005).

Initial Population. We generated the initial population randomly according to a uniform distribution. The number of generated solutions is given as the algorithm's parameter N, which is kept unchanged throughout the optimization process.

Selection. We used the roulette wheel selection. This selection operator assigns probabilities to each chromosome based on their fitness function values, then randomly selects some for the next stage. Using this operator, the proposed GA keeps a high proportion of better solutions moving forward to the next generations. This procedure speeds up GA's convergence since the probability of searching in nonpromising regions is very low.

Crossover. Our GA uses the binary crossover. This operator combines two selected parents' genetic codes, generating two new solutions (offspring). This operator replicates the parents' genetic code through a complete sequential evaluation of their chromosomes, applying a 50% probability of choosing a genetic code from one of them, i.e., we chose 50% to keep a fair balancing of both selected solutions. When this procedure generates an invalid solution, i.e., a solution with two or more threads assigned to a specific core, the algorithm fixes it by performing a random relocation in the empty cores.

Mutation. This operator acts in both chromosome parts: thread mapping and data mapping policy. For the thread mapping part, it randomly chooses two threads to perform a swap (changing their respective positions). While for the data mapping part, the operator randomly chooses one of the evaluated data mapping policies (*First-Touch*, *Interleave*, or *NUMA Balancing*) to replace the current one.

Update Population. Our GA uses the elitist criterion to insert new individuals into the population and to select individuals to move forward to the next generation. The algorithm keeps the population's elements sorted by their fitness values in increasing order during the optimization process. For every new individual p generated by crossover or mutation operator, the algorithm tests if it is suitable to enter the population P. Then, the algorithm removes the worst individual in the set $P \cup \{p\}$. At the end of each generation, the GA selects all inserted offspring and the current generation's best individuals to move forward to the next generation.

Fitness Function. Our GA aims to optimize the application execution time. To assign each solution to its fitness value, we performed three executions of the application and averaged the execution times.

Parameterization. For our GA, we considered a population size of 20 solutions, a crossover rate of 80%, and a mutation rate of 5%. We chose these values experimentally by running the GA algorithm several times, changing its parameter values deterministically, and selecting those that resulted in the best performance. By doing that, we get the GA's parameterization that matches that of other problems from the GA's literature (GOLDBERG, 2000).

4.4 Results

This section presents the *Graphith* results regarding its performance over the ***baseline*** and its convergence behavior. Moreover, we also compare the *Graphith* against all the evaluated TM and PM combinations.

Executions settings. Due to the non-deterministic behavior of Graphith's search part, which uses a GA, we performed 15 executions for each particular set of algorithms and input graphs. Results show the average of these executions (our experiments showed a coefficient of variation less than 1). We normalized the results by the **baseline**: *the regular OpenMP execution, which uses the thread placement defined by the Linux's scheduler and the First-Touch data mapping.* We executed the experiments on the same NUMA machines used in the DSE (*Intel32* and *Intel64* – see section 4.1 for details).

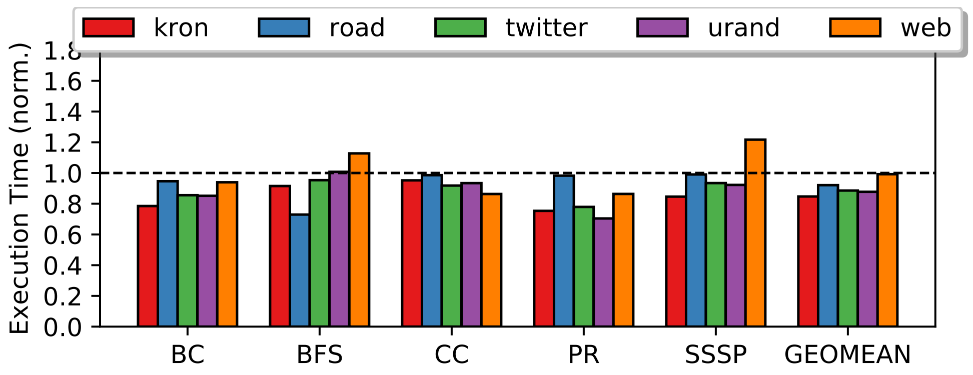
4.4.1 Graphith's Performance

To demonstrate the *Graphith* effectivity, we show in Fig. 4.6 the averaged results normalized by the **baseline** (y-axis). Each bar represents a particular input executed by a graph algorithm (x-axis). *Graphith* presents better results in most cases, showing, on average (geometric mean), 10% and 21% improvements in performance on the *Intel32* and *Intel64*, respectively.

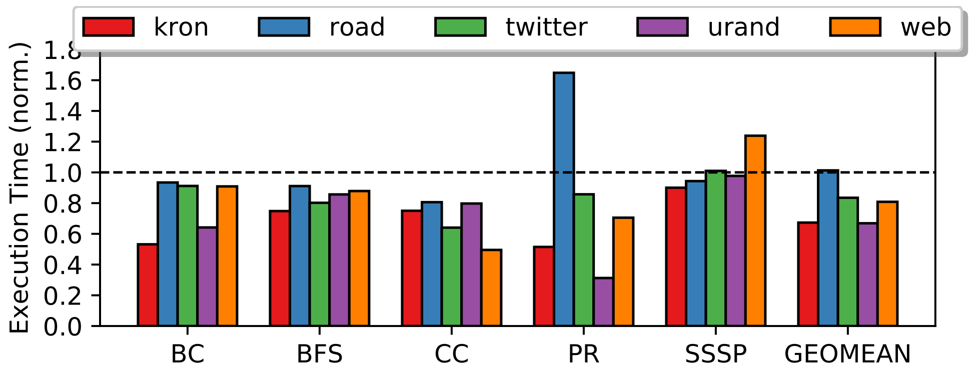
While *Graphith* outperforms the **baseline** in most cases, the computations per-

Figure 4.6: Graphith's performance normalized by the **baseline**.

(a) *Intel32*



(b) *Intel64*



Source: The author.

Table 4.3: Graphith's best thread mapping solution for PR on *Intel32* compared to the standard policies (last four rows).

Solution	**Thread Mapping**																														
Thr.id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
kron	10	14	9	22	2	5	31	30	15	24	20	13	21	19	1	18	0	29	8	12	17	6	7	25	28	3	16	4	11	26	23	27
road	13	26	18	15	28	9	23	16	4	19	25	14	30	31	3	24	21	2	5	12	11	10	22	20	1	8	7	29	6	0	27	17
twitter	8	18	22	15	17	14	9	13	23	16	0	20	1	19	29	7	4	26	11	21	24	3	30	28	25	10	6	5	31	12	27	2
urand	30	19	17	4	21	0	10	29	27	3	2	6	25	28	18	20	31	23	13	12	1	26	16	5	15	8	9	24	14	7	22	11
web	27	17	25	20	26	10	3	11	23	29	9	5	0	18	16	2	21	1	6	19	8	12	7	30	31	14	13	22	28	4	24	15
Default	0	16	1	17	2	18	3	19	4	20	5	21	6	22	7	23	8	24	9	25	10	26	11	27	12	28	13	29	14	30	15	31
Close	0	16	2	18	4	20	6	22	8	24	10	26	12	28	14	30	1	17	3	19	5	21	7	23	9	25	11	27	13	29	15	31
Contig.	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Scatter	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Source: The author.

formed by the GA's operators are intrinsically random. As a result, it may converge to low-quality solutions in a few cases. Examples of this behavior can be seen in Figure 4.6, particularly with the PR running on the *road* graph on the *Intel64* system and the BFS and SSSP running on the *web* graph on both systems. Despite *Graphith*'s control over GA's parameters and its operators' implementation, the performance at execution time and the search space region the algorithm will exploit can still exhibit significant randomness.

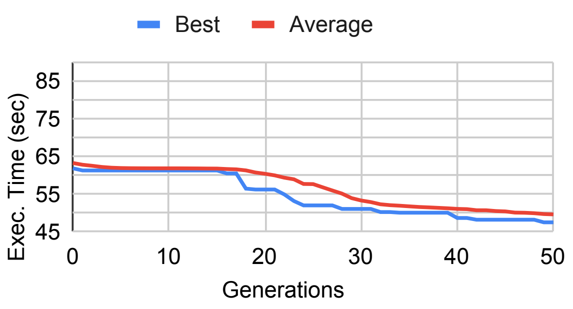
Finally, as another representative example, in Table 4.3, we compare *Graphith*'s solutions with the SO standard ones when optimizing the PR algorithm on the *Intel32*. This table depicts the thread-to-core placement. For example: in the *Scatter* policy, threads 0, 1, and 2 are allocated to cores 0, 1, and 2, respectively; and in the *Close* policy, threads 0, 1, and 2 are allocated to cores 0, 16, and 2. Unlike the standard policies, *Graphith*'s solutions do not follow specific rules (highlighted in the last four rows). Throughout the generations, *Graphith* adjusts the solution according to the graph algorithm behavior and the structure of the input graph. With that, *Graphith* converges to solutions that outperform all the TM+PM combinations (up to 18% for PR-twitter on *Intel64*).

4.4.2 Graphith's Convergence

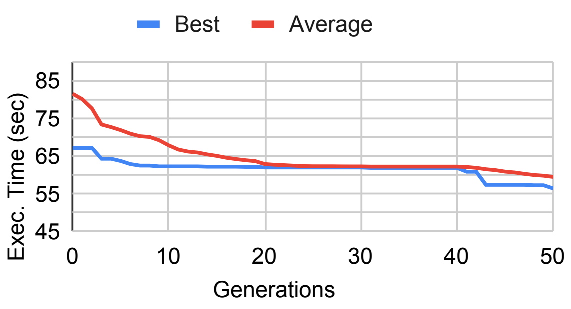
To demonstrate the convergence of our framework, we plotted in Fig. 4.7 how it evolves throughout the generations (x-axis), considering the fitness value of the GA population best's solution (Best in blue) and the average fitness values of all the solutions in the population (Average in red) when the GA optimizes the BC running the *kron* graph on both *Intel32* and *Intel64* – the other graph algorithms present similar convergence

Figure 4.7: *Graphith*'s convergence when optimizing the *BC* algorithm running (a) the *kron* graph on *Intel32* and (b) the *urand* graph on *Intel64*.

(a) BC-kron on *Intel32*.



(b) BC-urand on *Intel64*.



Source: The author.

behavior.

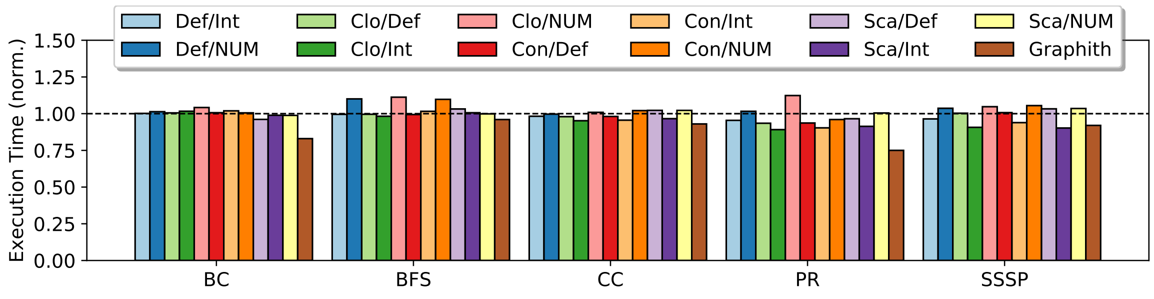
The population keeps evolving even when the best solution does not present a significant convergence pattern in *Graphith*'s optimization. Note that the average line (red) continuously evolves and sometimes gives a light touch to the best one (blue). The sudden improvements shown in Fig. 4.7, mainly in the best solution's fitness, are due to the mutation operator. Besides changing some thread places, this operator also changes the PM policies and can drastically change the algorithms' performance.

4.4.3 Comparison Against Traditional Policies

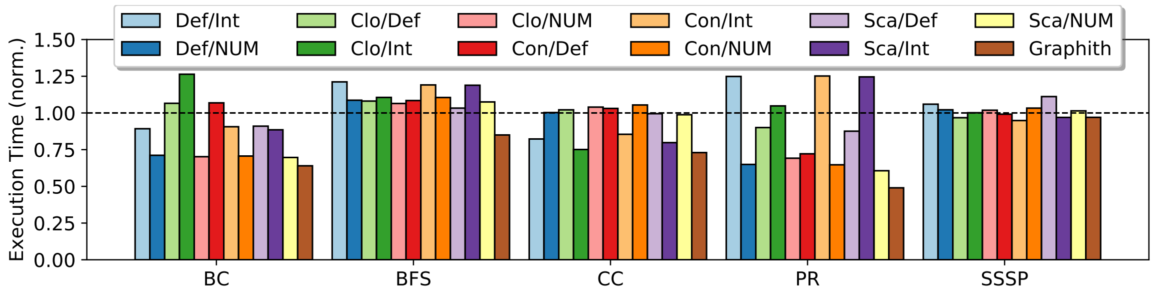
In Figure 4.8, we present the results obtained with *Graphith* in comparison to all of the TM+PM policy combinations evaluated in our DSE (see section 4.1) on both *Intel32* and *Intel64* systems. Each chart shows the execution times for each combination of TM+PM (y-axis, normalized by the **baseline**). We averaged the results for each graph algorithm executing all the evaluated input graphs (x-axis). As one can see, in most cases, *Graphith* outperforms any existing mapping combinations. **Overall**, *Graphith*'s performance is, in most cases, better than the baseline and the exhaustive search's best solution (which considers only the TM and PM policies available on the OS). The *Graphith* efficiency comes from its automatic GA that allows for a larger search space exploration - bringing significant gains without any code modification. While performing an exhaustive search with the available policies may provide performance improvements of up to 39% over the baseline (e.g., Sca/NUM in PR on the *Intel64*), *Graphith* goes beyond that and is capable of optimizing over the best possible solution available by the OS: 51% for

Figure 4.8: *Graphith* performance compared with all the combinations of the standard thread and data policies evaluated. Results are normalizes by the **baseline**. **Legend:** the <thread>/<data> refers to the combination of the thread and data mapping policies.

(a) *Intel32*



(b) *Intel64*



Source: The author.

the same graph algorithm and system.

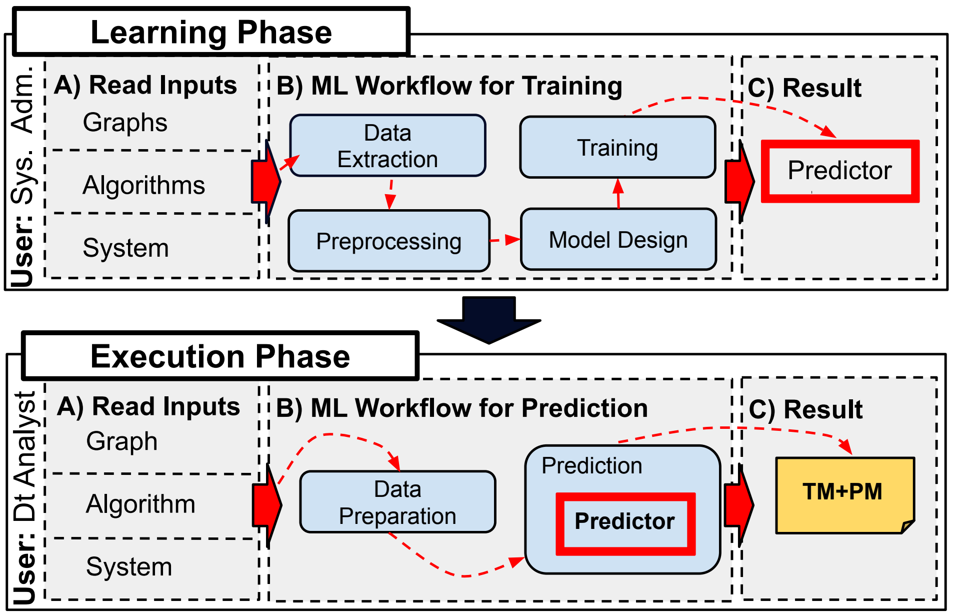
Although *Graphith* significantly improves graph processing performance, it is an inflexible strategy with a high learning overhead. It is worth noting that *Graphith* aims to find the best thread-to-core and PM solution for each algorithm when executing a specific input graph. Therefore, the entire search process must be re-executed if the graph changes. The next chapter introduces another approach to overcome this limitation.

5 PREDG: EXPLOITING THE GRAPHS' HIGH-LEVEL FEATURES FOR ADAPTIVE GRAPH PROCESSING

This chapter introduces the *PredG*, a framework that uses a machine learning (ML) methodology to optimize the execution of graph applications on NUMA machines by adjusting TM and PM policies. In Fig. 5.1, we provide an overview of *PredG*, which operates in two phases (further details will be provided later): in the *Learning Phase*, it performs an ML workflow to train a Predictor. In the *Execution Phase*, it carries out another ML workflow using the Predictor to find the best TM and PM policies combination (defined as TM+PM) for executing a new input graph.

The motivation to design *PradG* is that, although *Graphith* framework (presented in the previous chapter) is capable of improving the performance of graph applications significantly, it lacks adaptability. As already mentioned, *Graphith* fully optimizes the execution of each algorithm executing a specific input graph, but if the input graph changes, the entire search process needs to be re-executed. Similarly, most of the current graph analytic frameworks present the same drawbacks of *Graphith* since they implement graph partitioning and hybrid computation/communication strategies (ZHANG; CHEN; CHEN, 2015; SUN; VANDIERENDONCK; NIKOLOPOULOS, 2017; SHUN; BLELLOCH, 2013; AASAWAT et al., 2020). Because of that, they are far from reaching near-optimal solutions for many input graphs due to the huge amount of data and different topological

Figure 5.1: *PredG* framework overview.



Source: The author.

Table 5.1: **Graphs' characteristics:** Number of Vertices (V), Number of Edges (E), Diameter (Dia), Global Clustering Coefficient (GCC), Maximum Degree (MaxD), Averaged Degree (AvgD), and Degree of Assortativity (DA).

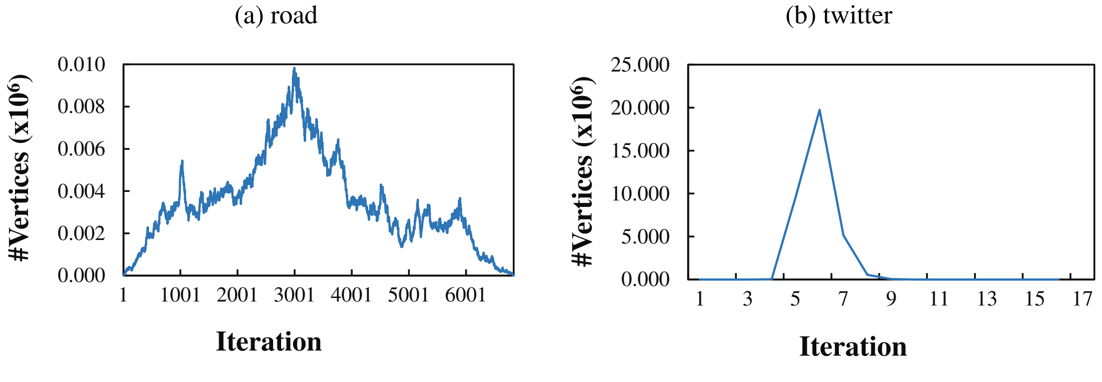
	V	E	Dia	GCC	MaxD	AvgD	DA
twitter	6.16e+07	1.20e+09	14	0.08	3.00e+06	39.06	-0.04
road	2.39e+07	2.89e+07	6304	0.02	9.00e+00	2.41	0.08

Source: The author.

structures (e.g., meshes and social networks). The reason is that the algorithm performance depends not only on the algorithm itself or the optimization strategy performed on it but also on the structure of the input graph to be executed.

To discuss the variation brought by changing the input graph, let us consider two graphs that present distinct topologies (see Table 5.1): *road* with the highest Diameter (Dia) and the lowest Maximum and Average Degree (MaxD and AvgD), representing a mesh; and *twitter* with a low Dia and high MaxD and AvgD, representing a social network graph. We show in Fig. 5.2 the iterations of the Breadth-First Search (BFS) algorithm executing over both input graphs, where each number in the x-axis represents one iteration, while the y-axis is the number of vertices executed at each iteration. As observed in Fig. 5.2, while bfs-road takes 6835 iterations to finish, processing a significant number of vertices throughout its execution (on average 3.503 vertices per iteration of a total of 2.39e+07), the bfs-twitter execution takes only 16 iterations to finish, with 98% of all the vertices (6.16e+07 in total) processed in only 3 iterations (5-7). This difference in the inputs affects the total execution time even if the algorithm is the same (BFS): the bfs-road takes 10% longer to run than the bfs-twitter, even though the road is 41.68× smaller (in the number of edges) than twitter (see Table 5.1).

Figure 5.2: The number of evaluated vertices (*y-axis*) over the BFS iterations (*x-axis*).



Source: The author

Fortunately, the graph structures that impact the execution time and cause the above execution variation may somehow be inferred by evaluating some graphs' high-level features, such as the Dia and the *Global Clustering Coefficient (GCC)* (see others in Table 5.1). These features are already available along with the graph data sources (LESKOVEC; KREVL, 2014) and can be exploited before each graph execution for optimization, precluding any profiling. Therefore, our main challenge is correlating such high-level features with low-level decisions (i.e., mapping) through a neural network.

Therefore, to design *PredG*, we exploited the input graph's high-level features not found in generic parallel applications to develop a novel offline strategy that adapts to new input graphs without needing application profiling. In other words, by taking advantage of these specific features only found in graphs, we can offer significant levels of adaptability in an offline method without incurring any penalties presented by online methods; and still cover NUMA systems, which is not adequately done by most specific graph processing frameworks (SHUN; BLELLOCH, 2013; ROY; MIHAILOVIC; ZWAENEPOEL, 2013; ZHANG et al., 2018; NGUYEN; LENHARTH; PINGALI, 2013).

In this scenario, the contribution of this chapter is:

- The proposal of *PredG*: a Machine Learning (ML) framework that considers only the high-level features of a given graph (see Table 5.1) to predict the best TM and PM policies for any graph execution. *PredG* works in two phases: in the ***Learning Phase***, *PredG* builds and trains an Artificial Neural Network (ANN) model to learn from representative graphs by considering their high-level features and execution of several algorithms on NUMA systems; and in the ***Execution Phase***, before triggering the execution of a given incoming graph, *PredG* predicts the best TM+PM by only using the trained ANN with the already available high-level features.

By evaluating *PredG* with widely used graph analysis algorithms processing real-world data on three different NUMA machines, we show that *PredG* finds the best solutions for the most applications (on average 81.33% of accuracy). Performance-wise, it is up to 41% better than the Linux OS *Default* and the *Best Static* mapping configuration – and on average only 2% worse than the *Oracle*. We also show that *PredG* is on average 8% and 16% more energy efficient than the *Default* and the *Best Static* strategies.

5.1 PredG

PredG is divided into two phases: *Learning* and *Execution*, as described next.

5.1.1 Learning Phase

PredG learns the best TM+PM based on the behavior of the given algorithms and system by processing different graphs with different structures (identified by different features). This phase is executed only once in the entire optimization process. For that, *PredG* applies the following steps, as can be seen in Fig. 5.3 and Fig. 5.4:

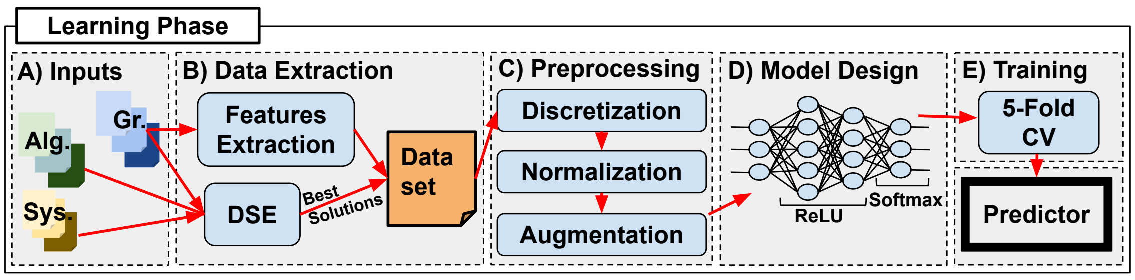
A) Inputs. This step consists of feeding *PredG* with the following inputs: the graphs in edgelist files representation (e.g., orkut.el and texas.el), the binary of the graph algorithms, and the target NUMA systems.

B) Data Extraction. *PredG* creates a Dataset composed of *(i)* the graphs' features and *(ii)* the best TM+PM for each graph, algorithm, and NUMA system. For the former *(i)*, *PredG* extracts the high-level features (e.g., diameter and averaged vertices degree) by using NetworKit, a parallel tool for large-scale network analysis (STAUDT; SAZONOVS; MEYERHENKE, 2016). For the latter *(ii)*, *PredG* performs a Design Space Exploration (DSE) by executing all the combinations of the input graphs and algorithms on the target NUMA system, considering the thread and data mapping policies available on the Linux OS. *PredG* uses the results found by the DSE to know which are the TM+PM solutions that result in the best performance for each evaluated graph and algorithm. Both *(i)* and *(ii)* are merged to compose the Dataset, used in the following steps.

C) Preprocessing. To avoid data issues that can bias the ML model (e.g., underfitting and overfitting), *PredG* preprocesses the Dataset by applying the following steps:

(i) *Discretization.* Because machine learning models require all variables to be numeric, this step encodes each categorical feature (string) to numbers before fitting and evaluating the ML model. For that, *PredG* uses the One-Hot Encoding as all categorical features in the Dataset do not present any sequence order (e.g., algorithm);

Figure 5.3: PredG's Learning Phase.



Source: The author.

(ii) Normalization. PredG normalizes the Dataset to a common scale because different features have different ranges (e.g., for the evaluated graphs in Table 2.1, while diameter (Dia) varies from 6 to 6304, the degree assortativity (DA) ranges from -0.11 to 0.19). *PredG* uses the *Min-Max* normalization strategy, computed by the equation $\frac{x-x_{min}}{x_{max}-x_{min}}$, where x, x_{min}, and x_{max} are the current, minimum, and maximum values of the current column being normalized, respectively. This procedure fits the Dataset to the range of $[0, 1]$; and

(iii) Augmentation. Selecting the best TM+PM solutions for each graph and algorithm (as done in the *B) Data Extraction*) may generate an unbalanced Dataset as specific solutions may present the best results for different graphs and algorithms while others do not. This can lead to overfitting, as the TM+PM solution that appears the most may bias the learning process. Therefore, *PredG* performs the random data augmentation strategy to randomly select, modify, and insert examples in the minority target classes to avoid such a problem, resulting in a balanced Dataset. *PredG* modifies the selected data to avoid identical samples in the Dataset. The modification is mainly done in the graph's high-level features, which slightly changes the graph's structure.

D) Model Design. *PredG* analytically performs the hyperparameter optimization (e.g., number of layers, types of activation functions, and learning rate value) to build an ANN model that best fits the Dataset created in the previous step. For that, it uses the Keras Tuner, a scalable hyperparameter optimization library that automatically searches for the best configuration and parameters of the ANN model (O'MALLEY et al., 2019). The considered ANN configurations are: number of hidden layers (varying in the range of [1, 5]); number of neurons in each hidden layer (8, 16, 32, and 64); activation function (Sigmoid, ReLU, Softmax, and Softplus); learning rate (0.01, 0.03, 0.05, 0.07, and 0.09); momentum (0.1, 0.3, 0.5, 0.7, and 0.9); and the number of epochs (50, 100, 150, 200, and 250).

E) Training. To train the ANN model, *PredG* uses the 5-Fold Cross-Validation strategy (YADAV; SHUKLA, 2016). This strategy randomly divides the Dataset into 5 folds/sections evaluated through the iterations. For example, in the first iteration, the first fold is used to test the model, and the rest are used to train the model. In the second iteration, the second fold is used as the testing set, while the rest serve as the training set. This procedure repeats until all 5 folds have been used as the testing set. As running an ANN is a stochastic procedure and each iteration of the above-described strategy may present a different outcome, *PredG* performs 30 executions of the 5-Fold Cross-Validation strategy and selects the best model to be the *Predictor*.

5.1.2 Execution Phase

This phase is triggered whenever a graph needs to be processed. After training, *PredG* can predict the best TM+PM policies for every graph without further application executions. For that, *PredG* applies the following steps (see Fig. 5.4).

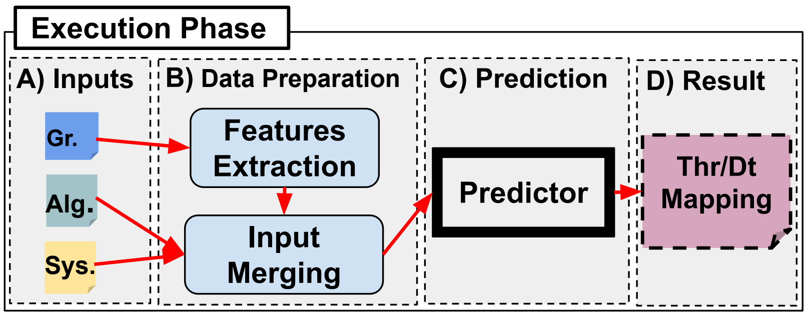
A) Inputs. The user is responsible for providing *PredG* with the following inputs: the new graph required to be processed – in edgelist files representation (e.g., twitter.el and road.el); the binary of the algorithm that was considered during the *Learning Phase*; and a description (*string*) indicating the current NUMA machine in which *GraphNroll* must have been trained in the *Learning Phase*.

B) Data Preparation. *PredG* preprocesses the data to feed the *Predictor*. It collects the same graph's high-level features of the input graph used in the *Learning Phase* using the NetworKit tool. Then, these features are merged with the target algorithm and system descriptions.

C) Prediction. Based on the information obtained in the previous step, *PredG* applies the *Predictor* (the previously trained ANN model) to find the best TM+PM policies for the specified graph, algorithm, and system.

D) Result. Finally, *PredG* fires the application execution with the predicted TM+PM solution set in the system.

Figure 5.4: PredG's Execution Phase.



Source: The author.

5.1.3 Implementation Details

We have implemented *PredG* using Python version 3.8.10. Currently, it is designed to work with OpenMP applications and utilizes the GOMP_CPU_AFFINITY OpenMP environment variable to establish specific thread mapping policies, as detailed in section 2. For data mapping, *PredG* configures the Linux system's file located at $/proc/sys/kernel/numa\_balancing$, setting it to either 0 for *First-Touch* only or 1 for *First-Touch* with *NUMA Balancing*, allowing for data migration as required. The *Interleave* policy is set using the *numactl* tool (KLEEN, 2004). To measure the execution time of each application, *PredG* employs the `time.time()` function provided by Python3's **time** module.

5.2 Methodology

Graphs Algorithms. We considered the same algorithms from GAPBS that were evaluated in the previous chapter (chapter 4), parallelized and compiled in the same way.

Graphs Data Input. We consider 15 representative real-world and synthetic graphs, which cover two comprehensive classes of topologies (i.e., meshes and social networks) and different dimensions (i.e., number of vertices and edges). In Table 2.1 (in chapter 2), we present each graph along with its high-level features that comprise the training Dataset. In the *PredG*'s Learning Phase, we considered the following graphs: the modified versions of urand, kron, twitter, web, and road (see the Preprocessing step in section 5.1.1 – *(iii) Augmentation*); and the original and modified versions of cit-patents, orkut, wikitalk, california, texas, youtube, pennsylvania, google, berkley, and amazon. In the *PredG*'s Execution Phase, we considered only the original versions of the largest graphs: urand, kron, twitter, web, and road – the same we evaluated in the previous chapter (chapter 4).

Execution Environment. We performed the experiments in three NUMA machines (using the Linux kernel v. 4.19): ***Intel32*** and ***Intel64***, as described in the previous chapter (chapter 4), and the following additional machine.

- ***Intel88***: 2x 22-core Intel Xeon E5-2699 v4 (Broadwell) @2.2 GHz, 2-way SMT (2 nodes / 44 cores / 88 threads). Each core has a 32KB L1 cache and 256KB L2 cache, and the system comprises 2x55MB of L3 cache and 2x128GB of main memory.

Evaluated Configurations. Our study considers all the TM and PM policies described in chapter 2: *Linux's Default solution* (De), *Close* (Cl), *Contiguous* (Co), and *Scatter* (Sc) for TM; and *First-Touch* (De), *Interleave* (In), and *NUMA Balancing* (NU) for PM. Based on that, we compared *PredG* with the following three strategies:

- ***Default***, the *regular execution*, which uses the *De/De* thread and data mapping configuration;
- ***Best Static***, the TM+PM solution that delivers the averaged best execution time for all algorithms and graphs evaluated in this chapter (which is the *Scatter/Interleave* on *Intel32* and *Intel88*, and *Scatter/Default* on *Intel64*);
- ***Oracle***, the best TM+PM for each algorithm and input graph considered in the *Execution Phase*, found via an exhaustive search that tries all possible combinations of mapping policies.

Analysis Tools. We used the Intel Performance Counter Monitor (WILLHALM; DEMENTIEV; FAY, 2016) to collect hardware counter information (e.g., memory accesses and interconnect links usage) for the results presented in section 5.3.2. We also collected the energy consumed by the DRAM and core domains, i.e., CPU and cache memories, through the Intel Running Average Power Limit (RAPL) (HÄHNEL et al., 2012) for the results in section 5.3.3.

5.3 Results

5.3.1 Evaluation of the *PredG*'s Phases

In this section, we present *(i)* the ANN model built by *PredG* and its accuracy for training, *(ii)* the *PredG*'s predicted solutions, discussing its adaptability as the input graph changes, and *(iii)* we compare the *PredG* optimization overhead against the *Oracle* strategy.

(i) Learning Phase. Given the data obtained from the combination of 15 graphs, 5 algorithms, and 3 NUMA systems executed on the 12 different TM+PM configurations (resulting in 2700 executions), *PredG* built the ML model for each machine. For all machines (*Intel32*, *Intel64*, and *Intel88*), the models consist of an ANN of four layers with 36 neurons in the input layer, 64/32 neurons in the hidden layer and 12 neurons in the output layer. Also, the following parameters are considered by the ANN: for *Intel32* and *Intel64* the learning rate is 0.01, the momentum is 0.9, and the number of epochs is 150; and for the *Intel88* the learning rate is 0.01, the momentum is 0.7, and the number of epochs is 100.

Once the model has been generated, *PredG* performs 30 executions of the 5-Fold Cross Validation strategy, selecting the best trained model to be the *Predictor*. The chosen model shows an accuracy of 91%, 89%, and 92% on *Intel32*, *Intel64*, and *Intel88*.

(ii) Execution Phase. We consider the evaluation of 5 large-scale input graphs not used in the *PredG*'s Learning Phase with the 5 available algorithms, resulting in a total of 25 different executions (algorithm and input). Table 5.2 shows the *PredG* predicted solutions for the evaluated multicore systems (*Intel32*, *Intel64*, and *Intel88*), where TM+PM indicates the predicted combination of thread and data mapping policies, while the number between parenthesis (x.xx) represents how distant the solution is from the *Oracle*, given in percentage of the execution time. Overall, *PredG* can correctly predict the best TM+PM in 88%, 68%, and 88% of the evaluated applications in *Intel32*, *Intel64*, and *Intel88*, respectively. Considering the design space exploration performed to find the best (*Oracle*) solutions, we found out that the *PredG* solutions are in the top 3 best solutions 92% of the time (while *Default* and *Best Static* reach the top 3 in only 26.67% and 44%

Table 5.2: PredG's predictions: TM+PM (x.xx) indicates the predicted thread and data mapping policies combination followed by the percentage difference from the *Oracle* solution.

		bc	**bfs**	**cc**	**pr**	**sssp**
	kr	Cl-In (0.00)	Cl-In (0.00)	Cl-In (0.00)	Sc-In (0.00)	Cl-In (0.00)
	ro	Cl-In (0.00)	De-In (0.00)	**Co-De (8.14)**	Sc-In (0.00)	Co-De (0.00)
I32	**tw**	Cl-In (0.00)	**De-In (5.74)**	De-In (0.00)	De-In (0.00)	Cl-In (0.00)
	ur	Co-In (0.00)	Sc-De (0.00)	**Sc-De (6.31)**	Sc-In (0.00)	Cl-In (0.00)
	we	Cl-De (0.00)	Cl-In (0.00)	Sc-In (0.00)	Co-In (0.00)	Sc-In (0.00)
	kr	**De-In (3.44)**	De-In (0.00)	**De-In (9.05)**	**De-In (82.49)**	De-In (0.00)
	ro	Co-De (0.00)	De-De (0.00)	De-In (0.00)	**Co-De (11.50)**	Sc-In (0.00)
I64	**tw**	**Cl-De (1.50)**	Cl-De (0.00)	De-In (0.00)	Co-De (0.00)	Sc-In (0.00)
	ur	**Co-In (0.90)**	Co-In (0.00)	Co-In (0.00)	**De-In (4.57)**	De-In (0.00)
	we	**De-De (13.58)**	Cl-In (0.00)	Cl-In (0.00)	Cl-In (0.00)	Co-In (0.00)
	kr	Sc-In (0.00)	Cl-De (0.00)	Sc-NU (0.00)	Sc-In (0.00)	Co-De (0.00)
	ro	De-De (0.00)	De-De (0.00)	De-De (0.00)	Cl-In (0.00)	De-De (0.00)
I88	**tw**	De-De (0.00)	De-De (0.00)	Sc-De (0.00)	Cl-In (0.00)	Cl-De (0.00)
	ur	Sc-In (0.00)	Sc-In (0.00)	Sc-NU (0.00)	Cl-NU (0.00)	Co-De (0.00)
	we	**De-In (9.35)**	De-In (0.00)	De-In (0.00)	**De-In (0.03)**	**De-In (2.66)**

Source: The author.

of the times).

Table 5.2 also depicts that there is *no one-fits-all* solution for all graphs, algorithms, and machines (i.e., it is not possible to find a single TM+PM solution that would present the best result for all particular algorithms and inputs). We can highlight the following examples: (*i*) the BFS algorithm presents different solutions as the input graph changes in both machines; (*ii*) the *road* graph on *Intel64* presents different solutions as the algorithm changes; and (*iii*) the best solutions for bc-road are *Cl/In*, *Co/De*, and *De/De* on *Intel32*, *Intel64*, and *Intel88*, respectively.

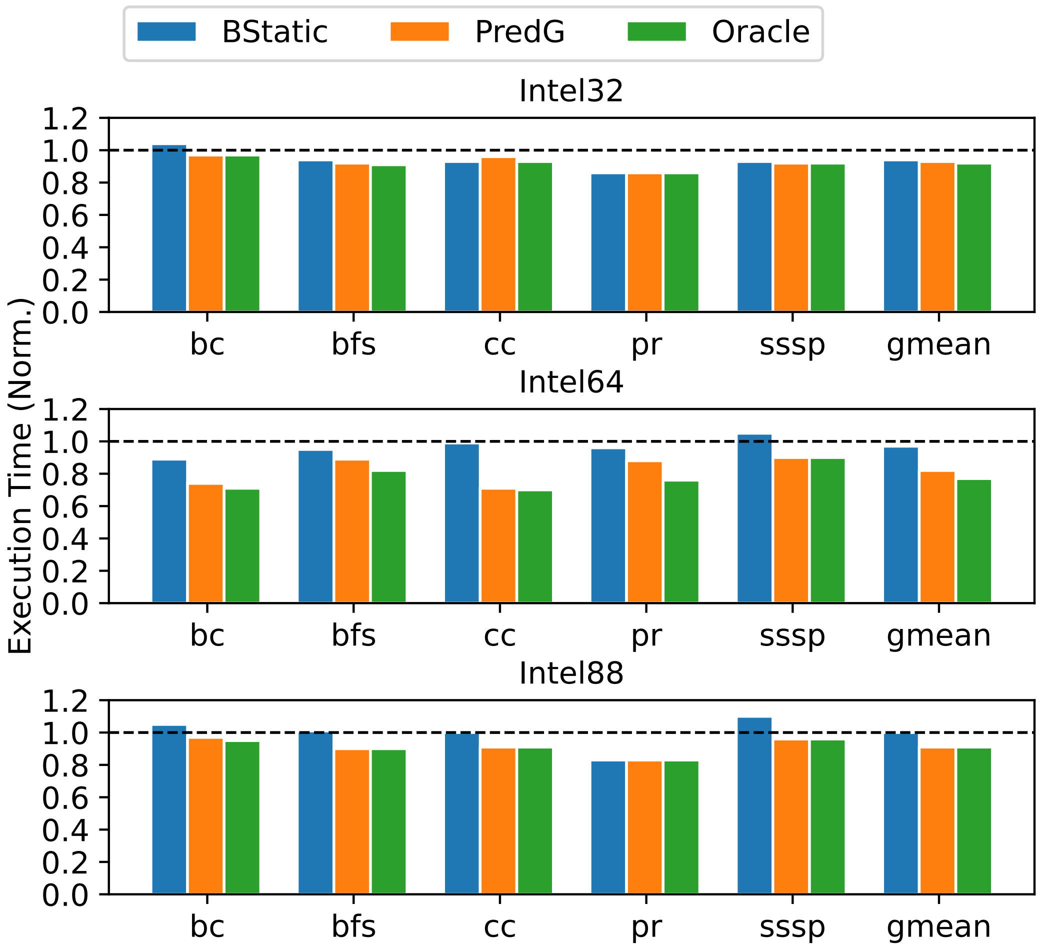
(iii) Costs of the Oracle (Exhaustive Search). If one would use the *Oracle* to find the best TM+PM solution, the exhaustive search to evaluate all possible combinations of algorithms, NUMA systems, and TM+PM configurations would take more than 85 hours, according to our experiments. This process would have to be repeated every time a new graph must be executed. On the other hand, the *PredG*'s Execution Phase takes only 0.08 seconds to perform 30 predictions (with a standard deviation of 0.0521) for all combinations of algorithms, input graphs, and machines.

5.3.2 Performance Evaluation

Let us now compare the performance achieved by *PredG*'s solutions with the *Default*, *Best Static*, and *Default* strategies. For that, Fig. 5.5 shows the execution time (y-axis) of each strategy (bars with different colors) grouped by the evaluated algorithms (the average of all evaluated input graphs) along with the *gmean* (x-axis). Results are normalized by the *Default*, hence, the lower, the better. On average, *PredG* is 8%, 22%, and 10% better than the *Default* and 1%, 18%, and 9% better than the *Best Static* on *Intel32*, *Intel64*, and *Intel88*, respectively.

PredG presented better results on the *Intel64* system as it is a more complex machine. Compared to *Intel32* and *Intel88*, *Intel64* has a more complex memory hierarchy composed of 4 NUMA nodes instead of 2 NUMA nodes in the other machines. To further assess the main sources of improvements obtained by the PredG's solutions in such a machine, we show in Table 5.3 the following metrics normalized by the *Default* strategy (so the lower, the better): *Time* is the application execution time (in seconds); *L2* is the miss rate in the cache level 2; *L3* is the miss rate in the cache level 3; *Rem* is the remote memory access rate; *MRead* and *WRead* are the numbers of bytes read/written from/to main memories; *WRem* is the number of bytes transferred in the most used system's in-

Figure 5.5: Execution time normalized to the *Default*, represented by the black line (↓ values = better execution time).



Source: The author.

terconnect links (the highest value considering all the system's sockets and interconnect links); *WMRead* and *WMWrite* are the number of bytes read/written from/to the most used system's main memory.

Our results show that the predicted solutions reduce the metrics related to the most accessed interconnect link and memory controller (*WRem*, *WMRead*, and *WMWrite*). For instance, in the most significant case for *Time* (cc-web), the reductions of 47%, 50%, and 11% in these respective metrics improve the execution time by 49%.

Considering that the computation performed by each algorithm is the same regardless of the TM+PM policies set in the SO, the number of memory operations is roughly the same. Hence, improvements on *WRem*, *WMRead*, and *WMWrite* indicate that the *PredG*'s solutions evenly distribute the application's data transferred across the interconnecting links (*WRem*), balancing the amount of work done by each memory controller (*WMRead* and *WMWrite*).

Note that, in most cases, there are no significant improvements in the other metrics

Table 5.3: Performance of the PredG solutions on *Intel64* normalized by *Default* (the regular execution).

Alg-Gr	Time	L2	L3	Rem	MRead	MWrite	WRem	WMRead	WMWrite
bc-kr	0.71	1.00	1.00	0.97	1.04	1.00	0.78	0.78	0.58
bc-ro	0.76	1.00	1.01	1.02	0.98	0.98	0.99	0.92	0.94
bc-tw	0.93	1.00	1.00	0.99	1.01	1.00	0.86	0.87	1.01
bc-ur	0.72	1.00	1.00	0.98	1.02	1.00	0.72	0.70	0.66
bc-we	0.86	1.00	1.00	0.95	1.02	1.02	0.65	0.70	0.80
bf-kr	0.83	1.00	1.00	0.96	1.00	1.05	0.65	0.85	0.89
bf-ro	0.75	1.00	0.99	1.04	0.96	0.90	0.86	0.81	0.90
bf-tw	0.72	0.99	0.98	1.22	0.99	0.94	0.56	0.48	0.85
bf-ur	0.68	1.00	1.00	1.00	0.99	0.97	0.92	0.91	0.79
bf-we	0.57	1.00	1.00	1.07	0.99	0.93	0.57	0.55	0.77
cc-kr	0.79	1.00	1.00	1.03	1.03	1.01	0.68	0.63	0.56
cc-ro	0.77	0.97	1.00	1.07	1.01	1.00	0.70	0.63	0.99
cc-tw	0.64	0.98	0.99	1.07	1.02	0.99	0.60	0.56	0.97
cc-ur	0.80	1.00	1.00	1.01	1.02	0.98	0.86	0.72	0.56
cc-we	0.51	1.00	1.01	1.03	1.01	1.03	0.53	0.50	0.89
pr-kr	0.69	1.00	0.99	0.94	1.07	0.95	0.65	0.64	0.68
pr-ro	0.98	1.00	1.00	1.01	1.01	1.00	0.83	0.68	0.94
pr-tw	0.93	1.00	1.00	1.00	1.00	1.00	0.97	0.92	0.95
pr-ur	0.98	1.00	1.00	1.00	1.00	1.03	0.94	0.94	1.03
pr-we	0.81	0.99	1.02	1.03	1.00	1.03	0.93	0.93	0.83
ss-kr	0.89	1.00	1.00	1.00	1.00	1.01	0.88	0.86	0.78
ss-ro	0.94	1.02	1.01	1.03	1.05	1.38	0.82	0.53	1.17
ss-tw	0.95	1.00	1.00	1.02	1.01	1.02	1.06	1.01	0.92
ss-ur	0.90	1.01	1.00	1.02	1.00	1.00	0.88	0.85	0.87
ss-we	0.95	1.00	1.00	1.04	1.00	1.02	0.90	0.84	0.97
gmean	0.79	1.00	1.00	1.02	1.01	1.01	0.78	0.74	0.84

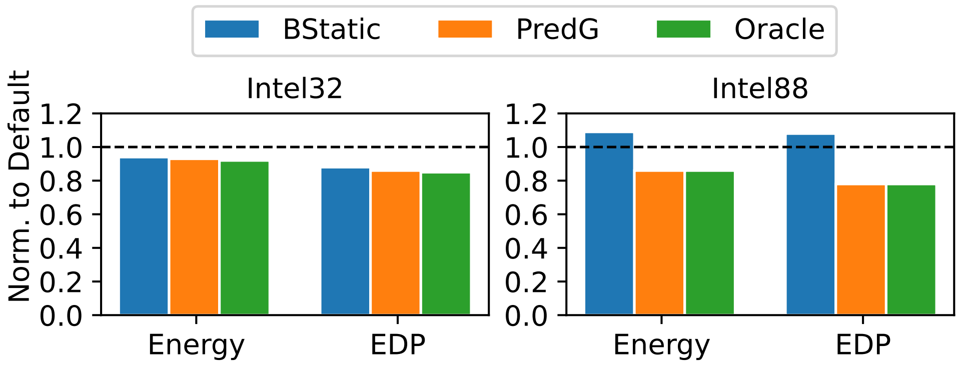
Source: The author.

(e.g., *L2*, *L3*, *MRead*, and *MWrite*) rather than the ones discussed above, which shows the importance of optimizing the memory accesses (i.e., avoiding remote accesses) and therefore the use of a proper thread/data mapping strategy.

5.3.3 The Impact of *PredG* on the Energy and EDP

When proposing techniques to optimize the performance of graph applications in cloud servers and HPC systems, a key challenge is not significantly increasing the energy consumption and the trade-off between both metrics (e.g., EDP) of the applications. Hence, to show that *PredG* can also bring energy and EDP improvements on top of optimizing for performance, we compare the energy and EDP of *PredG* with the previously

Figure 5.6: Energy and EDP normalized to the *Oracle*, represented by the black line (↓ values = better energy and EDP).



Source: The author.

discussed strategies. For that, we illustrate the overall geometric mean results for the *Intel32* and *Intel88* systems[1] in Fig. 5.6, where the energy and EDP of each strategy are normalized by the *Default* configuration (so the lower, the better). As one can observe, because *PredG* is capable of selecting the TM+PM policies that balance the amount of work done by each memory controller, reducing the congestion in the interconnect links and the usage of remote memories, it provides energy savings and EDP improvements: when considering the overall geometric mean, *PredG* reduces the energy consumption in 8% and 16% over the *Default* on the *Intel32* and *Intel88*, respectively. Furthermore, the very same behavior, but at different rates, is observed when the EDP metric is considered. *PredG* delivers results comparable to the *Oracle* (on average for all systems, only 1% and 2% worse than the *Oracle* in energy and EDP).

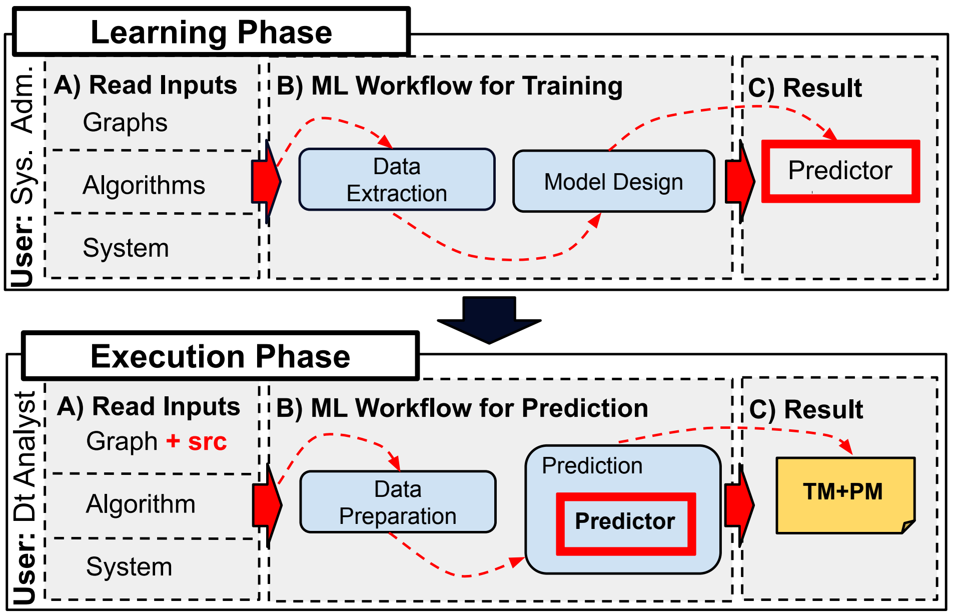
[1]The *Intel64* does not have support for energy counters.

6 GRAPHNROLL: OPTIMIZING SINGLE-SOURCE GRAPH EXECUTIONS

In this chapter, we introduce a new graph processing framework, *GraphNroll*, which fully leverages the optimization of single-source graph algorithms. Fig. 6.1 provides an overview of *GraphNroll*. Notice that it is very similar to *PredG* shown in Fig. 5.1, except that *GraphNroll*'s *Learning Phase* follows ML workflow for training based on different source vertices, and its *Execution Phase* receives a source vertex. Additionally, its *Execution Phase* receives not only the input graph but also the source vertex, enabling it to predict the best TM+PM configuration based on this source.

The motivation to propose *GraphNroll* is that the current graph processing frameworks (including the ones we have proposed in previous chapters) have been designed to extract valuable information from large graphs efficiently, none of them thoroughly analyze the variations that occur when executing single-source graph algorithms from different source vertices. In this kind of graph algorithm, every time a new execution is fired (so the source vertex changes), different parts of the graph with distinct structures and amounts of vertices/edges are processed. In other words, threads will process different data, changing how they communicate and access the memory regions. Significant examples of single-source graph algorithms are BFS and SSSP, widely applied for telecom network routing (PETERSON; DAVIE, 2007), neural image reconstruction (LI et al., 2019; MARRETT et al., 2021), road navigation (GOLDBERG; HARRELSON,

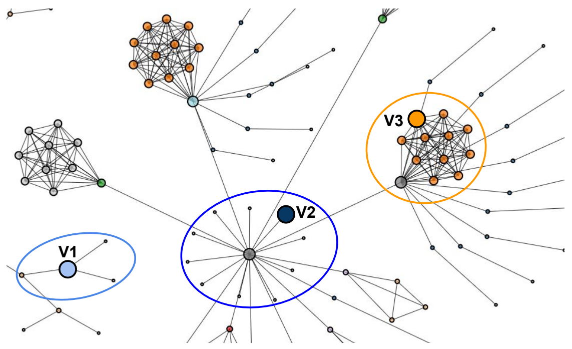
Figure 6.1: *GraphNroll* framework overview.



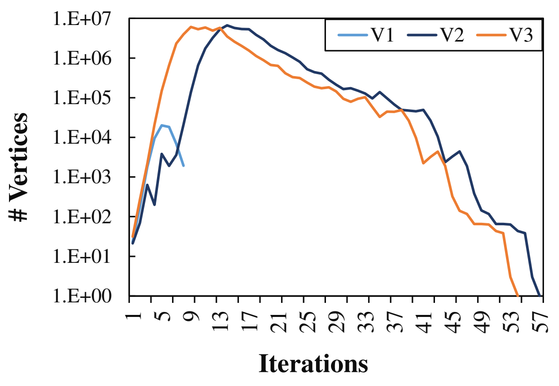
Source: The author.

Figure 6.2: Distinct views of the BFS executing the *web* graph on the *Intel64*. (a) depicts the structure of the sub-graphs to which the vertices *V1*, *V2*, and *V3* belong, (b) shows the number of vertices processed in each iteration, and (c) shows the results of different thread and data mapping combinations. **Legend:** the TM-PM refers to the combination of the thread and data mapping policies.

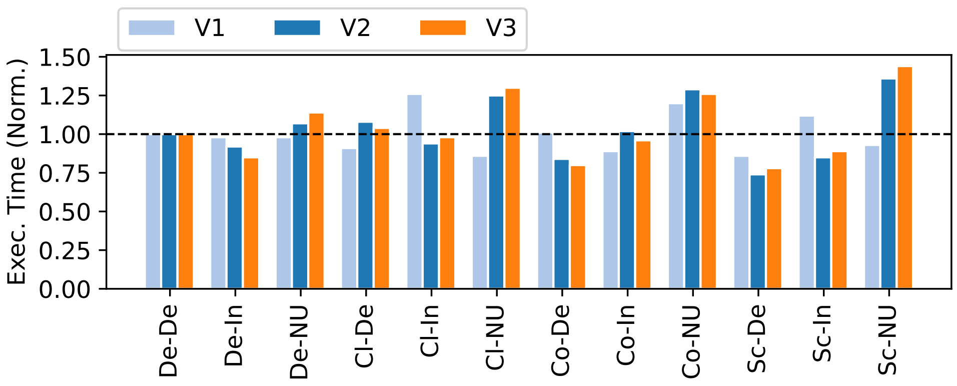
(a) *Graph Visualization*



(b) *Execution Variation*



(c) *Thread and Data Mapping Comparison*



Source: The author.

2005), and social network analysis (BRANDES; PICH, 2007).

To illustrate the discussion above, we present in Fig. 6.2 three different perspectives of the BFS executing the web input graph. Fig. 6.2 (*a*) depicts the source vertices (*V1*, *V2*, and *V3*) for three particular executions. When processing the graph from these different sources, a different number of vertices is processed in each iteration of the BFS, as presented in Fig. 6.2.b. The amount of data processed throughout execution may also vary, as in V1, in which only a small fraction of the graph is processed. The outcome of these variations is that the best TM+PM to execute the algorithm will change according to the source vertex, as illustrated in Fig. 6.2.c. It shows the execution time obtained by 12 different combinations of TM+PM policies normalized to Linux's default solution (De-De). As one can observe, *V1* is better executed with the combination Contiguous-Interleave (Co-In) while *V2* and *V3* achieve better performance with *Scatter-Interleave*

(Sc-In) and *Default-Interleave* (De-In), respectively.

Similar to the context of *PredG* (see chapter 5), it is evident that adaptability is essential here, but at a higher level since the optimization strategy must be resilient across different inputs and distinct source vertices that belong to the same input. With that in mind, we can exploit the same *PredG*'s idea of using specific input graphs' high-level features to train ANNs, but now, considering the features of the subgraph generated from the given source vertices. Fortunately, the sub-graph structure where a vertex belongs in large graph data can be expressed by embeddings. Graph embeddings are transformations of the graph's properties (e.g., topology, vertex-to-vertex relationship, and vertex similarities) to low-dimensional vectors so that they can be used as inputs of Artificial Neural Network (ANN) models. Thus, the embeddings of the source vertices can be exploited to develop a novel strategy to optimize the execution of single-source graph algorithms with almost no runtime overhead but offering significant levels of adaptability, exploiting at the same time the advantages of online and offline techniques, as also presented by *PredG*.

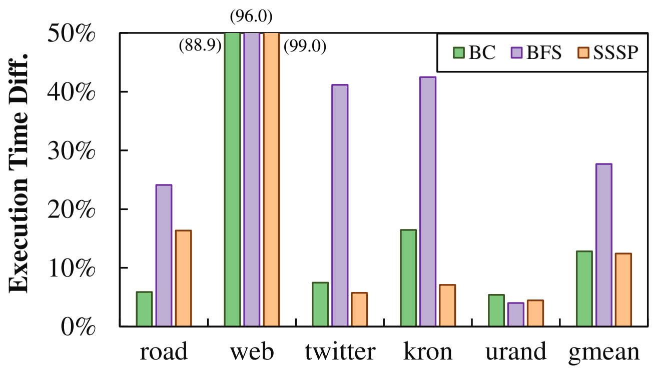
In this context, this chapter presents the following contributions:

- A DSE considering 3 single-source algorithms (BC, BFS, and SSSP) executing 5 real-world inputs graphs (road, web, twitter, kron, and urand), each starting from 15 different source vertices. With that, we evaluated the optimization potential of adjusting TM and PM for executing single-source algorithms;
- *GraphNroll*, a Machine Learning (ML) framework for enhancing the single-source graph algorithms' execution time by predicting the ideal TM+PM policies configuration as the source vertices change for any input graph, algorithm, and NUMA machine.

6.1 Design Space Exploration of Source Vertices

To further highlight the performance variation when executing single-source graph algorithms from different source vertices, we have performed a design space exploration (DSE) considering 3 single-source algorithms (BC, BFS, and SSSP) executing 5 real-world input graphs (road, web, twitter, kron, and urand), each starting from 15 different source vertices. We have used an Intel Xeon E5-2640 processor (*Intel32* - described in section 5.2) and Linux's default thread and data mapping policy (De-De). Based on this DSE, Fig. 6.3 presents, for each algorithm, the execution time difference (in %) between

Figure 6.3: The difference in the execution time (in %) between the vertices with the best and worst outcomes in performance when executing BC, BFS, and CC algorithms on the *Intel32* system with the Linux's default thread and mapping policies. The lower the bar, the lower the difference.



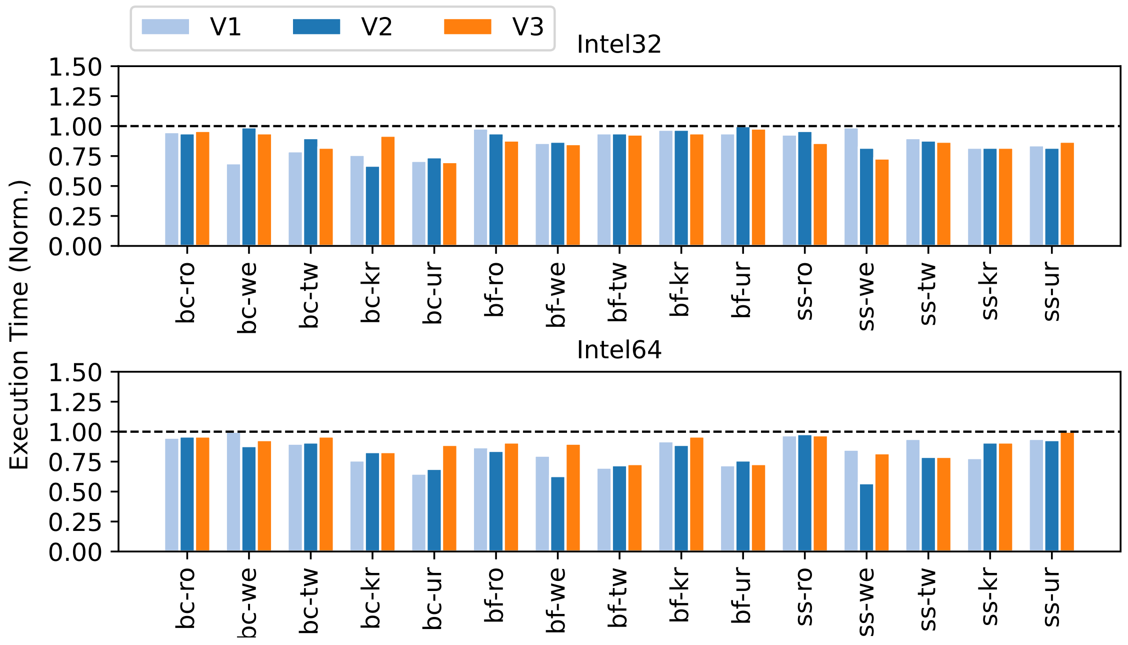
Source: The author.

the vertex that took the longest and the shortest time to execute. For instance, the two corner case vertices of BFS-kron have a 42.47% difference in execution time. This difference is even more evident for the *web* input graph: up to 99% (SSSP algorithm). This huge discrepancy happens because there is a significant variation in the number of processed vertices when the source vertex changes, as illustrated in Fig. 6.2.b. For instance, while BFS-web processes 58,264 vertices when starting from *V1*, it processes 50,539,645 vertices when it begins from *V2*. When considering the geometric mean of different input graphs on each algorithm, the time difference is 12.85%, 27.70%, and 12.43% in the BC, BFS, and SSSP, respectively.

On top of the above-presented results, as we showed in chapter 4, Linux's default thread and data mapping policies combination (*De-De*) is not always the best choice to execute graph applications. We illustrate this scenario for the single-source algorithms in Fig. 6.4 for the execution of the same algorithms and inputs as the previous experiment, but now considering 3 different source vertices (*V1*, *V2*, and *V3* - that do not represent the corner cases in execution time), and 2 distinct NUMA machines (*Intel32* and *Intel64*, described in section 5.2). Each bar in the plot represents the execution time achieved by the best thread and data mapping combination for the respective execution (algorithm + input graph + source vertex) normalized to the Linux's default (*De-De*, represented by the black line). Hence, the lower the value, the greater the reduction in the execution time. We also argue that there is no one-fits-all solution, as shown in Table 6.1, which depicts the best combination for each algorithm, input graph, and source vertex.

Therefore, based on this discussion, we can highlight the following:

Figure 6.4: Execution time of the best thread and data mapping combination when each algorithm (BC, BFS, and SSSP) starts from 3 different source vertices (V1, V2, and V3) with 5 input graphs (road, web, twitter, kron, urand) on the *Intel32* and *Intel64* machines. Results are normalized by the Linux's default thread and data mapping policies combination. The lower the bar, the better.



Source: The author.

Table 6.1: The ideal thread and data mapping policies combination when executing 3 single-source algorithms (BC, BFS, and SSSP) with 5 input graphs (road, web, twitter, kron, and urand) starting from 3 different source vertices (*V1*, *V2*, and *V3*) on 2 NUMA machines (*Intel32* and *Intel64*).

	Intel32			***Intel64***		
	V1	**V2**	**V3**	**V1**	**V2**	**V3**
bc-road	Cl-In	Co-In	Co-De	Co-In	Sc-In	Cl-De
bc-web	De-In	Co-De	De-In	De-De	Co-In	Cl-In
bc-twitter	De-In	De-In	De-In	De-In	Co-In	Sc-In
bc-kron	De-In	De-In	De-In	De-NU	De-In	De-De
bc-urand	De-In	De-In	De-In	De-In	De-In	Sc-De
bf-road	De-NU	De-NU	De-NU	Co-De	Cl-De	Sc-De
bf-web	Co-In	Cl-In	De-In	Co-In	Sc-In	De-In
bf-twitter	Sc-In	Co-Ine	Sc-De	Cl-In	Co-In	Sc-In
bf-kron	Co-In	De-NU	Sc-De	Co-In	Sc-In	Cl-In
bf-urand	Cl-In	De-De	Cl-De	Co-In	Co-In	Sc-In
ss-road	Sc-In	De-NU	Cl-In	Co-In	Cl-In	Sc-In
ss-web	Cl-De	De-In	Cl-De	Co-In	Cl-In	Co-De
ss-twitter	De-In	De-In	De-In	De-In	De-In	Cl-In
ss-kron	De-In	De-In	De-In	Cl-In	De-In	De-In
ss-urand	De-In	De-In	De-In	Sc-In	De-In	De-NU

Source: The author.

- Since single-source algorithms process different amounts of data with different structures, the ideal TM+PM solution for the same input graph and algorithm will very likely change if the source vertex changes. This scenario can be observed for the BC-urand on the *Intel64* machine. When the execution starts at *V1*, the best outcome in the execution time is achieved with the combination *De-In* (35% of improvements over *De-De*). However, when the source vertex is *V3*, the combination *Sc-De* reaches the best execution time, being 11% better than the default configuration;

- Even when the same TM+PM solution yields the best results for multiple source vertices, the rates of performance improvement over Linux's default solution can vary significantly. As a representative example, let us consider the SSSP execution over the *web* input graph (SSSP-web) on the *Intel32* system. In this scenario, when one defines the source vertices *V1* and *V3*, the best outcome reached for both is with the combination *Cl-De*, but with different rates: *V1* and *V3* are 17% and 9% faster than the default configuration, respectively;
- Furthermore, the same observations as those presented in section 4.1 remain with regard to the variation in the input graph, algorithm, and machines. Thus, the best TM+PM solution may also change if any of these variables change.

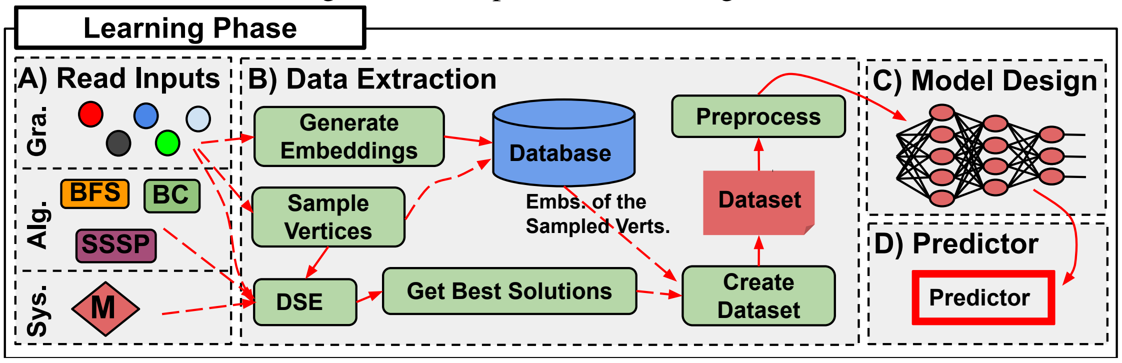
6.2 GraphNroll

Similar to *PredG*, as described in chapter 5, *GraphNroll* is a two-phase framework. The *Learning Phase* of *GraphNroll* is employed by the system administrator on the available NUMA machine. It learns the best TM+PM solution for each configuration of algorithm, input graph, and sampled source vertices (explained later) to create a *Predictor* (Fig. 6.5). In the *Execution Phase*, the end-user or data analyst uses the *Predictor* to give the best TM+PM solution to execute a given single-source algorithm over an input graph starting from a new source vertex (Fig. 6.6).

6.2.1 Learning Phase

To build the *Predictor*, *GraphNroll* learns the best TM+PM solutions based on the behavior of the given algorithms and NUMA system by processing different input graphs from different source vertices. For that, it applies the following steps, as can be seen in Fig. 6.5:

Figure 6.5: GraphNroll's Learning Phase.



Source: The author.

A) Read Inputs. In this step, the system administrator is responsible for feeding *GraphNroll* with the following inputs: *(Gra.)* the input graphs in edgelist files representation (e.g., *orkut.el* and *texas.el*); *(Alg.)* the binary of the graph algorithms compiled in the target machine; and *(Sys.)* the description of the target NUMA machine. With these inputs, *GraphNroll* sets up the input graphs and algorithms and identifies the machine configuration (e.g., number of cores, number of NUMA nodes, and memories' size) before moving to the next step.

B) Data Extraction. In this step, *GraphNroll* creates a *Dataset* comprised of (*i*) the embeddings of a subset of sampled source vertices of each given input graph and (*ii*) the ideal combination of TM+PM for each algorithm running each input graph starting from the sampled source vertices.

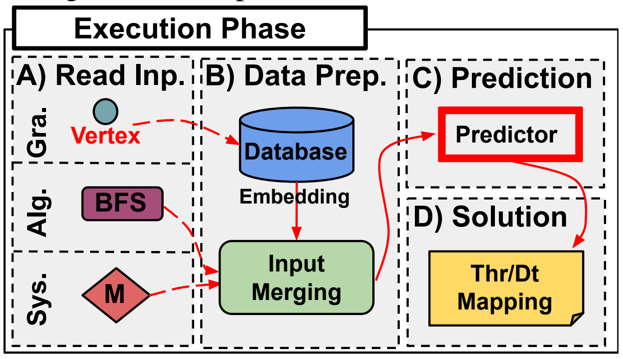
For *(i)*, *GraphNroll* considers all the given input graphs to generate the embeddings of all of their vertices (*Generate Embeddings*). Each embedding is a 1-dimensional vector representing the sub-graph structure where the vertex belongs (i.e., it encodes the vertex-to-vertex relationship in a vector representation). Hence, although different vertices will have different embeddings, those that belong to sub-graphs with similar structures will tend to be more similar to each other than those that belong to sub-graphs with different structures. To generate the embeddings, *GraphNroll* uses the PyTorch BigGraph, a scalable tool developed by Facebook AI Research, to create and handle large graph embeddings for ML (LERER et al., 2019). For each trained input graph, *GraphNroll* uses PyTorch BigGraph with different models to generate 20-dimension embeddings by following the methodology of Lerer et al. (LERER et al., 2019): *GraphNroll* adjusts the learning rates from 0.001-0.1, margins from 0.05-0.2, and negative batch sizes of 100-500, and each configuration is trained for 10 epochs. In the end, *GraphNroll* returns the highest accurate model, which is stored in a *Database* to be used in the *Execution Phase*.

For *(ii)*, a *DSE* is performed to find the best TM+PM solution for each algorithm, input graph, and sampled vertex. Because graph data may contain billions of vertices and edges – the largest publicly-available real-world graph, the HyperlinkWeb, comprises over 3.5 billion vertices and 128 billion edges (DHULIPALA; BLELLOCH; SHUN, 2021) – performing a *DSE* that covers the whole graph vertices for all graphs is impractical. Hence, *GraphNroll* samples representative source vertices (*Sample Vertices*) based on the work of Beamer et al. (BEAMER; ASANOVIĆ; PATTERSON, 2015), where vertices with a non-zero degree are randomly selected from each input graph. Then, *GraphNroll* performs the *DSE* by executing the given algorithms with the input graphs starting from each sampled source vertex on the target NUMA system, considering the following TM and PM policies combination available on the Linux OS (detailed in chapter 2): *OS's Default* (**De**), *Scatter* (**Sc**), *Contiguous* (**Co**), and *Close* (**Cl**) for TM; and *First-Touch* (**De**), *Interleave* (**In**), and *NUMA Balancing* (**NU**) for PM.

With the *DSE*, *GraphNroll* selects the TM+PM solutions that result in the best performance for each sampled vertex of each input graph (*Get Best Solutions*). Both outputs of steps *(i)* and *(ii)* are merged (*Create Dataset*) to compose the rows of the *Dataset*, e.g., embeddings + classes. Finally, this *Dataset* is preprocessed (*Preprocess*) by discretization, normalization, and data augmentation procedures to avoid overfitting or underfitting issues that can bias the ML model. For discretization, *GraphNroll* applies the One-Hot Encoding strategy to encode each categorical feature to numbers. For normalization, it fits the *Dataset* to the range of $[0, 1]$ using the *Min-Max* strategy. Lastly, for data augmentation, it performs a random strategy that randomly selects, modifies, and inserts examples in the minority target classes, resulting in a balanced *Dataset*. The resulting *Dataset* is then moved to the next step.

C) Model Design. With the created *Dataset*, *GraphNroll* performs the analytic hyperparameter optimization to build the best ANN models by automatically changing the ANN model's parameters to find the one that delivers the best accuracy. The values evaluated for each parameter are the number of hidden layers, varying in the range of [1, 5]; the number of neurons in each hidden layer (8, 16, 32, 64, and 128); activation function (Sigmoid, ReLU, LeakyReLU, Softmax, and Softplus); learning rate (0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, 0.009); and momentum (0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09). For each combination of these parameters, *GraphNroll* trains throughout several epochs until it achieves a threshold of 0.1 for the mean square error (value analytically found by exhaustive evaluations, which precludes the model of

Figure 6.6: GraphNroll's Execution Phase.



Source: The author.

overfitting). The final product of the *Learning Phase* is the resulting model, referred to as *Predictor* (see illustration D in Fig.6.5).

6.2.2 Execution Phase

This phase employs the *Predictor* generated in the previous phase to find the best TM+PM solutions without any further application execution. For that, *GraphNroll* applies the following steps (see Fig. 6.6):

A) Read Inputs. The data analyst is responsible for feeding *GraphNroll* with the following inputs: (*Gra.*) the input graph required to be processed in edgelist files representation (e.g., *orkut.el* and *google.el*); (*Alg.*) the binary of the algorithm that was considered during the *Learning Phase*; (*Vertex*) the ID of the new source vertex to start the execution; and (*Sys.*) indicating the current NUMA machine in which *GraphNroll* must have been trained in the *Learning Phase*.

B) Data Preparation. *GraphNroll* obtains the previously generated embedding of the source vertex in the *Database* (see *Data Extraction* section) and merges it with the descriptions of the target algorithm and system to feed the *Predictor*.

C) Prediction. With the merged data obtained in the previous step, *GraphNroll* applies the *Predictor* to find the ideal TM+PM solutions for the graph, starting with the specified vertex, algorithm, and system. Finally, *GraphNroll* sets the predicted thread and data mapping configuration on the NUMA system and starts the application execution (see illustration D in Fig.6.6).

6.2.3 Implementation Details

We implemented *GraphNroll* using the same methodology as that of *PredG*, as presented in chapter 5, employing the same Python version and settings for TM and PM.

6.3 Methodology

Graph Algorithms. We evaluate *GraphNroll*'s ability to optimize the following three single-source algorithms available on the GAP Benchmark Suite (GAPBS) repository (BEAMER; ASANOVIĆ; PATTERSON, 2015): BC, BFS, and SSSP (the same used in chapter 5). We compiled each application with GNU *g*++ 10.1.0 and OpenMP 4.5, with the *-O3* optimization flag.

Graph Data Input. We consider 11 representative real-world input graphs: road, amazon, berkley, california, cit-patents, google, orkut, pennsylvania, texas, wikitalk, and youtube (LESKOVEC; KREVL, 2014).

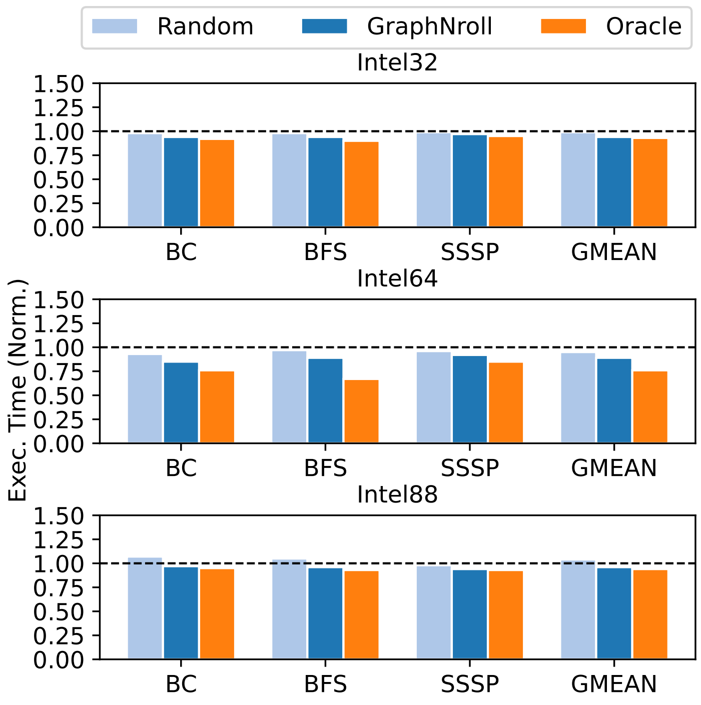
Execution Environment. We performed the experiments on the same three NUMA machines as described in chapter 5, all running Linux kernel version 4.19: *Intel32*, *Intel64*, and *Intel88*.

Evaluated Configurations. We compared *GraphNroll* with three different strategies:

- ***Default***, the *regular execution*, which uses the *De-De* thread and data mapping configuration;
- ***Random***, a random TM+PM solution generated for each algorithm, input graph, and initial vertex executed on each machine;
- ***Oracle***, the best TM+PM for each input graph and algorithm executed on a given source vertex found through an exhaustive search evaluating all 12 possible combinations of mapping policies.

Training and Validation Vertices Sets. For each evaluated input graph, we sampled 50 different source vertices and split them into an 80%-20% proportion as training and validation sets (i.e., 40 and 10 vertices used only for *GraphNroll*'s *Learning* and *Execution* phases, respectively).

Figure 6.7: Execution times of BC, BFS, and SSSP on the evaluated machines (*Intel32*, *Intel64*, and *Intel88*) when using the different compared strategies. Results are normalized to the *Default*, so the lower, the better.



Source: The author.

Table 6.2: *GraphNroll*'s predictions for 3 different vertices of the road (roa), cit-patents (cit), and amazon (ama): TM-PM (x.xx) indicates the predicted thread and data mapping policies combination followed by the percentage difference from the *Oracle* solution.

	Intel32			**Intel64**			**Intel88**		
	V1	**V2**	**V3**	**V1**	**V2**	**V3**	**V1**	**V2**	**V3**
bc-roa	Co-De (0.00)	Cl-De (0.24)	Co-De (0.85)	Cl-De (0.00)	Cl-De (0.17)	Sc-De (0.00)	Co-De (0.00)	Cl-De (0.00)	Sc-De (0.00)
bc-cit	Cl-De (0.00)	De-NU (0.00)	Cl-De (0.00)	Sc-NU (3.05)	Cl-De (0.00)	Sc-NU (3.05)	Cl-NU (0.00)	De-NU (1.44)	Cl-NU (0.00)
bc-ama	Co-In (2.83)	Cl-NU (0.00)	Co-De (2.24)	De-De (0.00)	De-In (9.27)	De-De (0.00)	De-NU (0.00)	De-De (5.29)	De-De (0.00)
bfs-roa	De-NU (0.00)	De-NU (16.03)	Co-De (0.00)	Cl-De (0.00)	De-De (0.00)	Co-De (0.00)	Sc-NU (0.00)	Co-De (9.20)	De-NU (1.79)
bfs-cit	Cl-De (0.00)	Cl-De (7.01)	Cl-De (0.00)	Sc-In (0.00)	De-In (0.00)	Sc-In (0.00)	Cl-NU (0.00)	Cl-NU (10.92)	Cl-NU (0.00)
bfs-ama	De-NU (0.00)	Sc-De (1.78)	Cl-De (2.65)	Cl-In (28.19)	Co-De (0.00)	Sc-In (0.00)	Cl-In (0.00)	De-De (0.00)	Cl-In (0.00)
sssp-roa	Cl-In (1.92)	Cl-De (0.90)	Sc-In (1.86)	Sc-In (0.36)	Sc-In (1.34)	Sc-In (0.00)	Sc-De (0.00)	Cl-NU (0.00)	Sc-De (0.00)
sssp-cit	Co-NU (0.00)	Sc-De (1.24)	Co-NU (0.00)	Co-NU (25.12)	Cl-NU (34.10)	Co-NU (25.12)	Co-NU (3.57)	Cl-NU (0.27)	Co-NU (3.57)
sssp-ama	Sc-NU (0.00)	Co-NU (0.00)	Cl-In (0.89)	Co-NU (0.00)	Sc-De (11.37)	Co-NU (0.00)	Cl-NU (0.00)	Cl-NU (0.00)	Sc-De (0.00)

6.4 Results

6.4.1 *GraphNroll*'s Performance and Solutions

We start by comparing the performance achieved by the *GraphNroll*'s solutions with the evaluated strategies (*Default*, *Random*, and *Oracle* – see section 6.3). For that, we show in Fig. 6.7 the execution time (y-axis) for each strategy (bars with different colors) grouped by the evaluated algorithms along with the geometric mean (GMEAN) – *x*-axis – for each machine (each different bar chart). Each bar shows the average of all evaluated input graphs and 10 source vertices used in the *GraphNroll*'s *Execution Phase* (see section 6.3) with mean variation coefficients for all machines of 0.16, 0.22, 0.11 for *Default*, *Random*, and *GraphNroll*, respectively. The variation coefficient indicates the standard deviation relative to the mean of each strategy considering all input graphs and source vertices. Results are normalized to the *Default*, hence, the lower, the better.

When considering the geometric mean, *GraphNroll* outperforms the *Default* by 6%, 14%, and 4%, and the *Random* by 4%, 8%, and 8%, on *Intel32*, *Intel64*, and *Intel88*, respectively. In the most significant scenario, it achieves up to 18% improvement over the *Default* when executing BFS on *Intel64*.

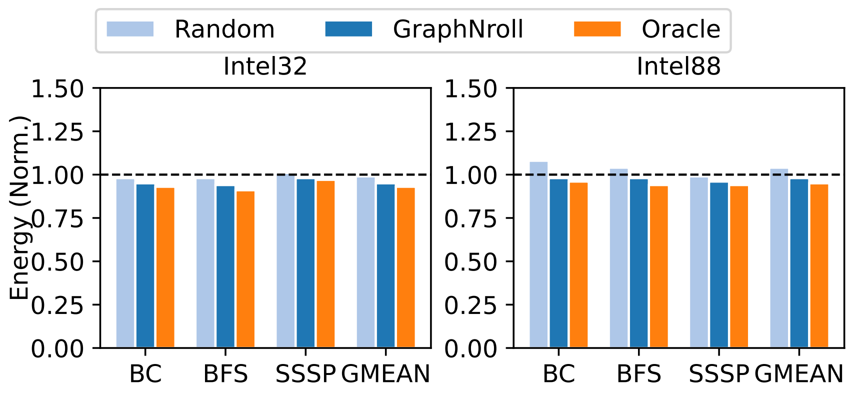
Table 7.1 depicts the solutions predicted by *GraphNroll* compared to the *Oracle* ones on each target architecture. *TM-PM* indicates the predicted combination of thread and data mapping policies, while the number between parenthesis (x.xx) represents how distant the performance is from the *Oracle*, given in percentage of the execution time. We highlight the results for the 3 most representative graphs evaluated in this chapter: *road* represents a mesh, *cit* is a citation network, and *ama*, a social network graph. When considering the cases in which *GraphNroll* achieved the same results as the *Oracle* (i.e., percentage distance is 0.00), one can highlight the *no one-fits-all behavior* in the single-source algorithm execution, as we also presented in the DSE of this chapter (see 6.1). Therefore, we can see that the ideal TM+PM solution changes if the source vertex changes: the single-source algorithm will process different amounts of data with distinct structures. This behavior is evident in the case of *bc-cit* when starting from *V1* and *V2* on the *Intel32* machine. Even if we keep the same source vertex, variations still arise due to differences in the input graph, algorithm, and NUMA machine, as discussed in section 6.1.

6.4.2 Impact of *GraphNroll* on the Energy and Energy-Delay Product

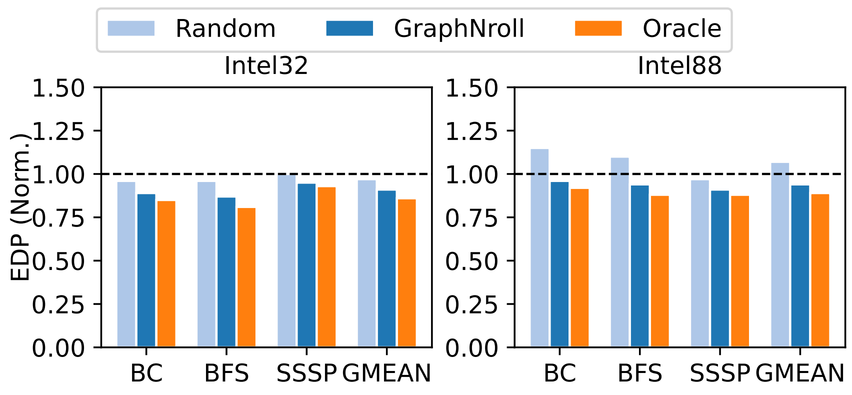
In Fig.6.8, we compare the energy and energy-delay product (EDP) results of *GraphNroll* with the evaluated configurations on the *Intel32* and *Intel88* (*Intel64* machine does not support hardware energy counters). We collected the energy spent by the DRAM and core domains, i.e., CPU and cache memories, through the Intel Running Average Power Limit (RAPL) (HÄHNEL et al., 2012). Similar to the performance results, each

Figure 6.8: Energy and EDP of BC, BFS, and SSSP algorithms on *Intel32* and *Intel88* when using the different compared strategies. Results are normalized to the *Default*, so the lower, the better.

(a) Energy



(b) EDP



Source: The author.

bar shows the average of all evaluated input graphs execution from 10 source vertices and normalized to the *Default*, hence, the lower, the better. Such results showed mean variation coefficients, for all machines, of 0.24, 0.08, and 0.07 on energy and 0.39, 0.09, and 0.09 on EDP for *Default*, *Random*, and *GraphNroll*, respectively.

As one can observe when considering the overall geometric mean, *GraphNroll* reduces energy consumption and EDP compared to all other approaches. It reduces the energy consumption in 4% and 3%, and the EDP in 11% and 7% over the *Default* on the *Intel32* and *Intel88*, respectively. Compared to the *Random* solution, *GraphNroll* is better by 4% and 6% in energy and by 8% and 15% in EDP on *Intel32* and *Intel88*. *GraphNroll* delivers results comparable to the *Oracle* since it is only, on average, 2% and 5% far from the *Oracle* in energy and EDP, respectively.

6.4.3 Costs

While executing single-source algorithms using the *Default* or *Random* TM+PM configurations is straightforward with no training overheads, such strategies lack solution quality. As we show in Fig. 6.7, *Default* and *Random* are up to 49% and 45% (BFS on *Intel64*) far from the *Oracle*, respectively. On the other hand, one can try to find the *Oracle* solution to extract the full performance potential from the single-source algorithms at hand. However, to find the *Oracle* solution, one must execute an exhaustive search to evaluate all possible TM+PM combinations for new source vertices, which takes a considerable amount of time. For example, it takes 3.23 hours to find the best solutions for the SSSP processing the *texas* input graph starting from 50 different source vertices and running on *Intel64*.

GraphNroll provides the benefit of quickly delivering a solution, as *Default* and *Random*) strategies, but with solutions very close to the *Oracle* (on average 7%). It also achieves the top-3 best solutions in 62.51% of the times, while the *Default* and *Random* reach the top-3 in only 20.56% and 24.54% of the times. The *GraphNroll*'s presents an average inference time of 0.0665 seconds (with a standard deviation of 0.0161) to predict the best solution for all machines, algorithms, input graphs, and source vertices.

It is important to mention that all the above results consider the availability of vertex embeddings. Otherwise, one must train the embeddings using PyTorch Big Graph, which can be time-consuming. Table 6.3 presents the execution time required to train embeddings for each evaluated graph. Due to time-sharing limitations on the clusters where we conducted our experiments and some scalability issues of PyTorch Big Graph, we were unable to obtain the embeddings of larger graphs, like twitter, kron, urand, and web evaluated in chapters 4 and 5. To maintain consistency with the input graphs evaluated in previous work, we assessed only the 11 graphs shown in Table 6.3.

Table 6.3: Time consumed to train the embeddings.

Graphs	**Time**
orkut	01:39:29
road	01:13:54
cit-patents	00:15:09
berkley	00:08:21
roadNet-CA	00:06:33
wikitalk	00:06:23
google	00:06:03
roadNet-TX	00:04:50
roadNet-PA	00:03:51
youtube	00:03:47
amazon	00:01:17

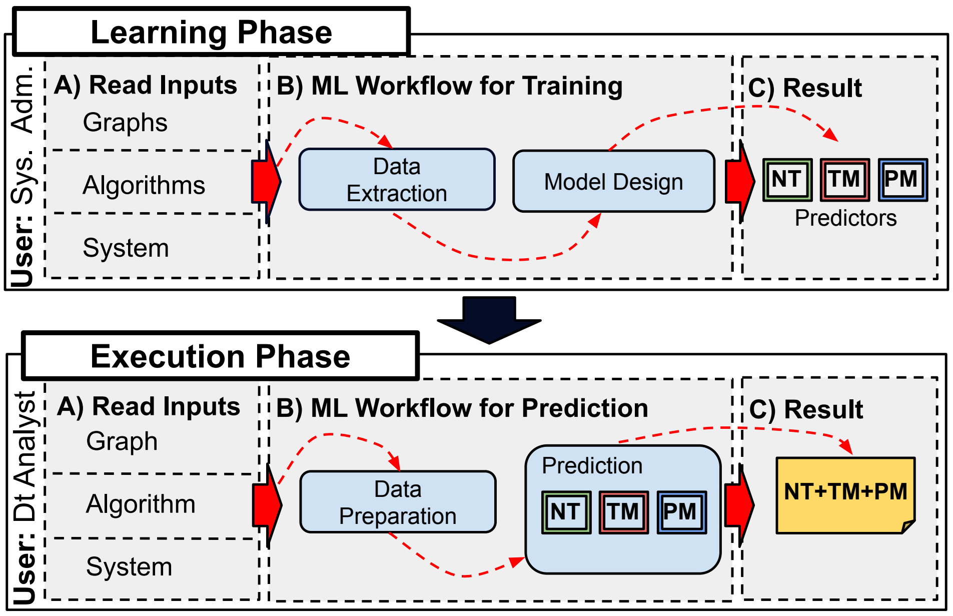
Source: The author.

7 POTIGRAPH: ADJUSTING NUMBER OF THREADS AND THREAD/DATA MAPPING

As we have shown in the previous chapters, a smart **Thread Mapping (TM)** and Data/**Page mapping (PM)** across the available resources are mandatory to improve the performance of graph applications on NUMA machines. However, while TM and PM may mitigate some of the intrinsic NUMA penalties, e.g., costly remote memory accesses (ZHANG; CHEN; CHEN, 2015), another performance limitation remains w.r.t. the parallel applications' scalability, where the increase in the amount of resources (e.g., cores) will not proportionally deliver the same performance improvement levels. The reasons are well-known in literature (SCHWARZROCK et al., 2020; LUAN et al., 2022; MALAVE; SHINDE, 2022), and so to achieve the best performance, in many times applying thread throttling, which artificially decreases the **Number of Threads (NT)**, may also be as important (SCHWARZROCK et al., 2020).

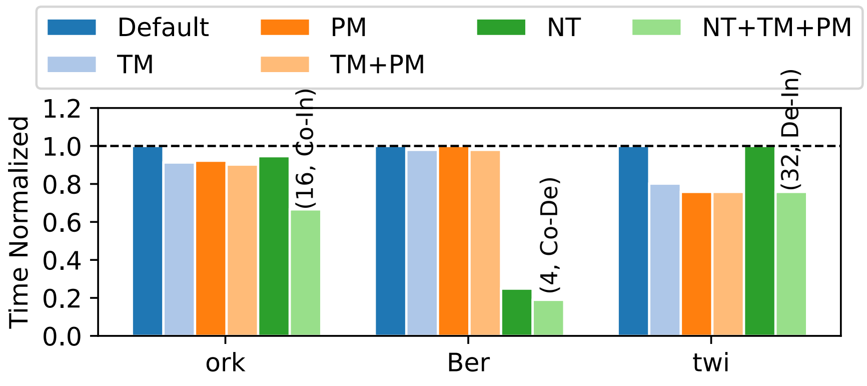
Motivated by this, in this chapter, we extend *PredG* (as discussed in Chapter 5) to optimize **NT** alongside **TM+PM** (referred throughout this work as **NT+TM+PM**). This extension results in a new framework named *PotiGraph*. An overview of *PotiGraph* is depicted in Figure 7.1. It also comprises two phases, with a major difference from *PredG* being the creation of three distinct predictions that simultaneously predict each of the respective variables (**NT**, **TM**, and **PM**). These predictors are represented by squares

Figure 7.1: *PotiGraph* framework overview.



Source: The author.

Figure 7.2: Comparison of different solutions: Default, the regular execution; TM, only TM; PM, only PM; NT, only number of threads; TM+PM, TM and PM together; and NT+TM+PM, NT, TM and PM together.



Source: The author.

with different colors in C of the *Learning Phase* and in B of the *Execution Phase* depicted in Fig. 7.1. Due to the limitations explained at the end of the previous chapter, *PotiGraph* does not consider single-source graph algorithms specifically.

To demonstrate the *PotiGraph*'s potential of optimizing **NT** alongside **TM** and **PM**, we present Figure 7.2. The figure showcases the execution time comparison of the SSSP algorithm on three input graphs: ork, Ber, and twi. The evaluated solutions include: the Linux's *Default Configuration* (DIENER et al., 2014; GUREYA et al., 2020b; MALAVE; SHINDE, 2022); the optimization of a single parameter (NT, TM, or PM) while keeping the others at their default settings; the optimization of TM+PM with NT at max; and the simultaneous optimization of NT, TM, and PM (more details are given in section 7.2). Results are normalized against the *Default* (the lower, the better). Observations from the results include:

- When the application scales to the maximum thread count (twi), optimizing TM+PM (as we have done so far) results in noteworthy performance enhancements. For instance, in the twi graph, this optimization yields a 24% improvement over the *Default* configuration;
- When the application exhibits limited scalability, we can enhance its performance by adjusting NT+TM+PM. For instance, in scenarios where SSSP does not scale with increasing NT (ork and Ber), optimizing TM+PM results in performance improvements of up to 10%. Nevertheless, optimizing all parameters (NT+TM+PM) leads to significantly higher improvements, reaching up to 80% (Ber).

Hence, it is evident that optimizing NT, TM, and PM together is crucial for exploiting their full potential. *PotiGraph* adopts the same *PredG* approach, leveraging high-level features from input graphs to train three separate Artificial Neural Network (ANN) models. These models predict the best NT+TM+PM solutions (when deployed on the target NUMA machines) without requiring further application execution, utilizing only these high-level features. Consequently, *PotiGraph* strikes a balance between the benefits of offline and online methods, mitigating their respective drawbacks.

7.1 PotiGraph

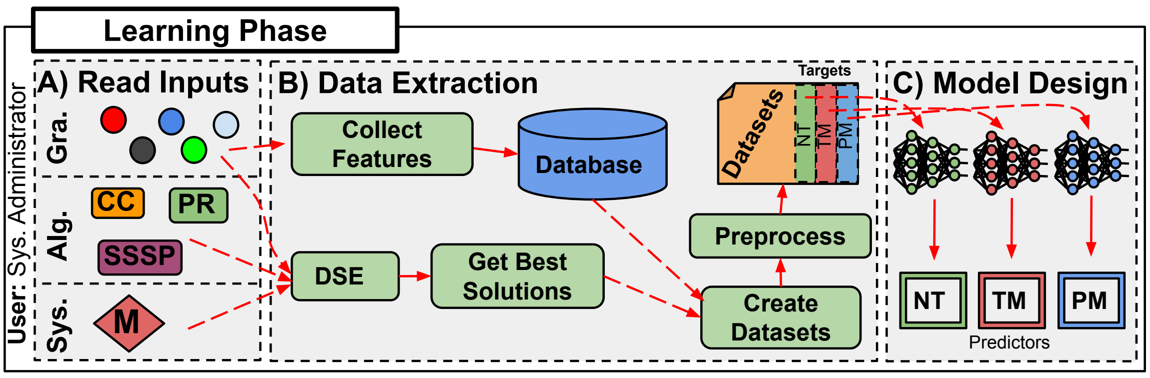
Similar to *PredG* (see chapter 5), *PotiGraph* consists of two phases, as illustrated in Fig. 7.3 and Fig. 7.4: *Learning Phase*, when it learns the best NT+TM+PM combination for each graph algorithm and input data to create three different *Predictors* (one for each variable to be optimized – NT, TM, and PM) (Fig. 7.3); and *Execution Phase*, which will be triggered when a new incoming graph arrives for execution, and is used to deliver the best NT+TM+PM combination by using the predictors (Fig. 7.4).

7.1.1 Learning Phase

The *Learning Phase* is composed of the following steps, as can be seen in Fig. 7.3:

A) Read Inputs. *PotiGraph* is fed with the following inputs (the same as *PredG* in chapter 5): *(Gra.)* the input graphs in edgelist files representation (e.g., *google.el* and *texas.el*); *(Alg.)* the binary of the graph algorithms compiled in the target machine; and *(Sys.)* the description of the target NUMA machine.

Figure 7.3: *PotiGraph*'s Learning Phase.

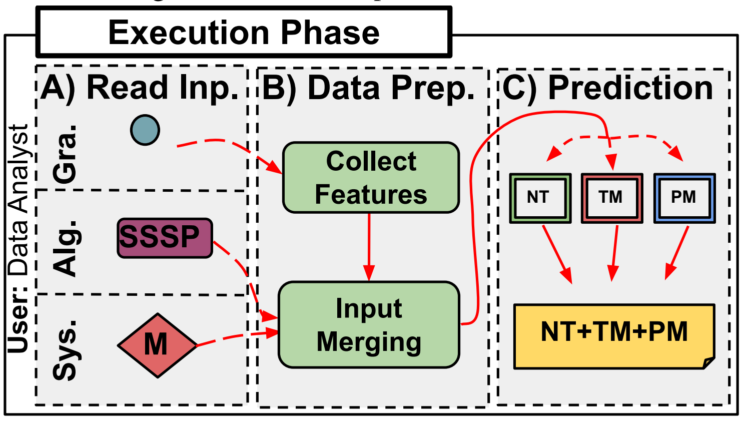


Source: The author.

B) Data Extraction. *PotiGraph* creates three *Datasets*, one for each optimization target (NT, TM, and PM). Each *Dataset* comprises (*i*) the high-level features of the input graphs and (*ii*) the ideal solution of the respective target for each graph algorithm running a given input graph. The high-level features (*i*) of the graph data are also given as input if they are present in the graph source (which is the case for most of them (ROSSI; AHMED, 2015; LESKOVEC; KREVL, 2014)), otherwise, they are extracted using the NetworKit tool (STAUDT; SAZONOVS; MEYERHENKE, 2016) (*Collect Features*), a tool for large-scale network analysis. These features are then stored in a *Database* for subsequent use in the *Execution Phase*. Then, *PotiGraph* performs a design space exploration (DSE) to find optimal NT+TM+PM combinations for each graph and algorithm. The DSE is done by applying a binary search for the NT, and, for each probe of NT, it tests all possibilities for TM and PM (outlined in chapter 2). Then, data from both (*i*) and (*ii*) are integrated (*Create Datasets*) and undergo preprocessing (*Preprocess*), including One-Hot Encoding for categoricals, *Min-Max* normalization to a $[0, 1]$ range, and Random data augmentation for balanced class representation. This results in the refined *Datasets*, which will be used during the step below.

C) Model Design. *PotiGraph* generates **three different models** using the previously created datasets: *(i)* a regression model to NT and *(ii)* classification models to TM and PM. For that, it performs analytic hyperparameter optimizations, changing the model's parameters to find the one that delivers the best accuracy. In case *(i) PotiGraph* considers only one neuron with a Linear activation function in the output layer. For the remaining layers, it tunes the number of hidden layers considering the range [1, 5]; the number of neurons in each hidden layer (8, 16, 32, 64, and 128); the activation function (Sigmoid, ReLU, LeakyReLU, Softmax, and Softplus); the learning rate (0.001 to 0.009, interval of 0.001); and momentum (0.01 to 0.09, interval of 0.01). For case (*ii*), *PotiGraph* does the same as above, but considering 4 and 3 neurons with the Softmax activation function in the last layer for the TM and PM model, respectively. For each combination of these parameters, *PotiGraph* trains throughout several epochs until it achieves a threshold of 0.1 for the mean square error (value analytically found by exhaustive evaluations to avoid overfitting). These resulting models are referred to as the NT, TM, and PM predictors, which work individually to predict the best number of threads, thread mapping, and data mapping, respectively.

Figure 7.4: *PotiGraph*'s Execution Phase.



Source: The author.

7.1.2 Execution Phase

This phase is similar to *PredG* (see chapter 5), but applying the three different predictors for predicting NT, TM, and PM simultaneously (NT+TM+PM). The steps performed by *PotiGraph* are (see Fig. 7.4):

A) Read Inputs. *PotiGraph* receives the following inputs: (*Gra.*) the *new input graph* in edgelist files representation; (*Alg.*) the binary of the graph algorithm (that should have been considered during the *Learning Phase*); and (*Sys.*) the current NUMA machine.

B) Data Preparation. *PotiGraph* collects the same high-level graphs' features from the input graph, which is available in the graph data or will be collected by using the NetworKit tool (*Collect Features*). Then, the features are merged with the descriptions of the target algorithm and system to feed the NT, TM, and PM predictors.

C) Prediction. *PotiGraph* applies the predictors to find the best NT+TM+PM combination for the *incoming input graph*. Then, it fires the application execution with the predicted solution set in the target system.

7.1.3 Implementation Details

We implemented *PotiGraph* using the same methodology as that of *PredG*, as presented in chapter 5, employing the same Python version and settings for TM and PM. To adjust the number of threads, *PotiGraph* uses *OMP_NUM_THREADS* environmental variable.

7.2 Methodology

Graph Algorithms. We evaluated the graph processing algorithms available in the GAPBS repository (BEAMER; ASANOVIĆ; PATTERSON, 2015) also evaluated by *PredG*: BC, BFS, CC, PR, and SSSP. They were parallelized and compiled in the same way.

Graph Data Input. We considered the same 15 input graphs evaluated by *PredG*. From these graphs, we considered 5 for validation (used only in the *Execution Phase*): road, cit-patents, amazon, california, and berkley, leaving the remaining 10 graphs for training.

Considered High-Level Features. We considered the same features as in *PredG* (also shown in Table 2.1 of chapter 2).

Execution Environment. We ran our experiments on *Intel32* and *Intel88*, each configured with Linux kernel v. 4.19 (see sections 4.1 and 5.2 for details).

Compared Solutions. We compared *PotiGraph* with the following:

- ***Default***, the regular execution, with the maximum NT, TM as defined by the Linux's scheduler, and *First-Touch* as PM (DIENER et al., 2014; GUREYA et al., 2020b; MALAVE; SHINDE, 2022);
- ***Random***, a random combination of NT+TM+PM;
- ***Oracle** (100% accuracy)*, when the predictors determine the optimal NT+TM+PM configuration. It is important to note that TM and PM only consider the traditional policies explained in chapter 2;
- ***BstN***, the best solution for NT but with defaults TM and PM;
- ***BstTP***, the best solution for TM+PM but with default NT;
- ***PredG***, our proposal to optimize TM+PM, as shown in Chapter 5;
- ***NTP***, the optimization of NT, TM, and PM, in this specific order (which is the best order for most generic applications (SCHWARZROCK et al., 2020));
- ***TNP***, the optimization of TM, NT, and PM, in this specific order;
- ***TPN***, the optimization of TM, PM, and NT, in this specific order;

Note that the *Oracle* consistently provides the optimal NT+TM+PM combination, resulting in superior performance compared to other strategies. We considered the same four TM and three PM policies as described in chapter 2 for all the compared strategies. Results have a standard deviation between 0.0014 and 0.0033 (15 executions each).

Table 7.1: Solutions predicted by *PotiGraph*.

	Intel32			**Intel88**		
	NT	**TM-PM**	**Diff to Oracle**	**NT**	**TM-PM**	**Diff to Oracle**
BC-roa	**16**	**Cl-De**	(1.00)	**40**	**De-De**	(1.00)
BC-cit	**16**	**Sc-NU**	(1.00)	**20**	**Sc-NU**	(1.00)
BC-ama	**8**	Sc-**De**	(1.14)	**12**	Co-**NU**	(1.01)
BC-CA	**16**	De-**De**	(1.38)	**40**	**De-NU**	(1.00)
BC-Ber	13	**Co-NU**	(1.02)	**14**	**Co-De**	(1.00)
BFS-roa	18	**Cl-NU**	(1.30)	21	**Co-De**	(1.00)
BFS-cit	**16**	**Sc-De**	(1.00)	25	**Sc-De**	(1.00)
BFS-ama	**16**	**Sc-NU**	(1.00)	**22**	**Co-In**	(1.00)
BFS-CA	**16**	**Co-NU**	(1.00)	29	**Cl-NU**	(1.00)
BFS-Ber	**8**	**Co-NU**	(1.00)	**18**	**Co-De**	(1.00)
CC-roa	**32**	**De-In**	(1.00)	**88**	**De-In**	(1.00)
CC-cit	**32**	**Cl-In**	(1.00)	**88**	**Co**-De	(1.16)
CC-ama	**32**	**Sc-NU**	(1.00)	36	De-**NU**	(1.06)
CC-CA	32	De-**NU**	(1.03)	**88**	**Cl-De**	(1.00)
CC-Ber	**16**	**Sc-NU**	(1.00)	**42**	**Sc**-De	(1.10)
PR-roa	**32**	**Cl-In**	(1.00)	45	**De-In**	(1.00)
PR-cit	**32**	**Sc-NU**	(1.00)	**88**	**Sc-NU**	(1.00)
PR-ama	**16**	**Sc-NU**	(1.00)	**34**	**Cl-NU**	(1.00)
PR-CA	**32**	**De-NU**	(1.00)	**88**	**Co-In**	(1.00)
PR-Ber	**32**	Sc-NU	(1.30)	35	**Sc-In**	(1.00)
SSSP-roa	15	**Sc-In**	(1.20)	**44**	Co-**NU**	(1.02)
SSSP-cit	**4**	**Co-NU**	(1.00)	**2**	**Co-NU**	(1.00)
SSSP-ama	**2**	**Co-De**	(1.00)	**2**	**Co**-NU	(1.03)
SSSP-CA	**1**	**Co-De**	(1.00)	**1**	**Co-NU**	(1.00)
SSSP-Ber	**4**	**Co-De**	(1.00)	**4**	**Co-De**	(1.00)

Source: The author.

7.3 Results

7.3.1 Accuracy of *PotiGraph*

We start by discussing the NT+TM+PM configurations predicted by *PotiGraph*. For that, Table 7.1 depicts the configuration (represented by the columns **NT**, **TM-PM**) and the execution time difference to the *Oracle*, for each graph application and input, on both architectures. Hence, a value *1.00* means that the *Execution Phase* of *PotiGraph* successfully predicted either the same configuration as the *Oracle* (i.e., perfect predictors) or a similar configuration that delivers the same performance. To facilitate the visualization, we highlight in **bold** the individual solutions of *NT*, *TM*, and *PM* that were correctly predicted by *PotiGraph*. *PotiGraph* accurately predicts the optimal configuration in 74% of cases (37 out of 50 executions). When evaluating the accuracy of each predictor individually, we find that the NT has an accuracy of 80%, the TM has 86%, and the PM has 94%.

When the predictors and *Oracle* do not match, in most cases, the predictors deliver configurations that are close to the *Oracle* performance-wise: in the geometric mean across all graphs, input sets, and NUMA architectures, the performance of the executions with configurations given by the predictors are only 3% slower than executing with the configurations delivered by the *Oracle*. Only in specific scenarios, such as with the BC-CA on the *Intel32*, when *PotiGraph* incorrectly predicts the TM variable, the execution is significantly slower (38%).

Another important point to highlight is that similar to the optimization of only TM+PM, there is also the no *one-fits-all* behavior for NT+TM+PM as the input graph, algorithm, and machine changes. Representative examples include: BC on *Intel88* yielding different solutions with changes in the input graph; BFS and PR processing the *CA* input graph on *Intel32* achieving their best performance with different configurations; and the BC-road solution on *Intel32* differing from the solution on *Intel88*.

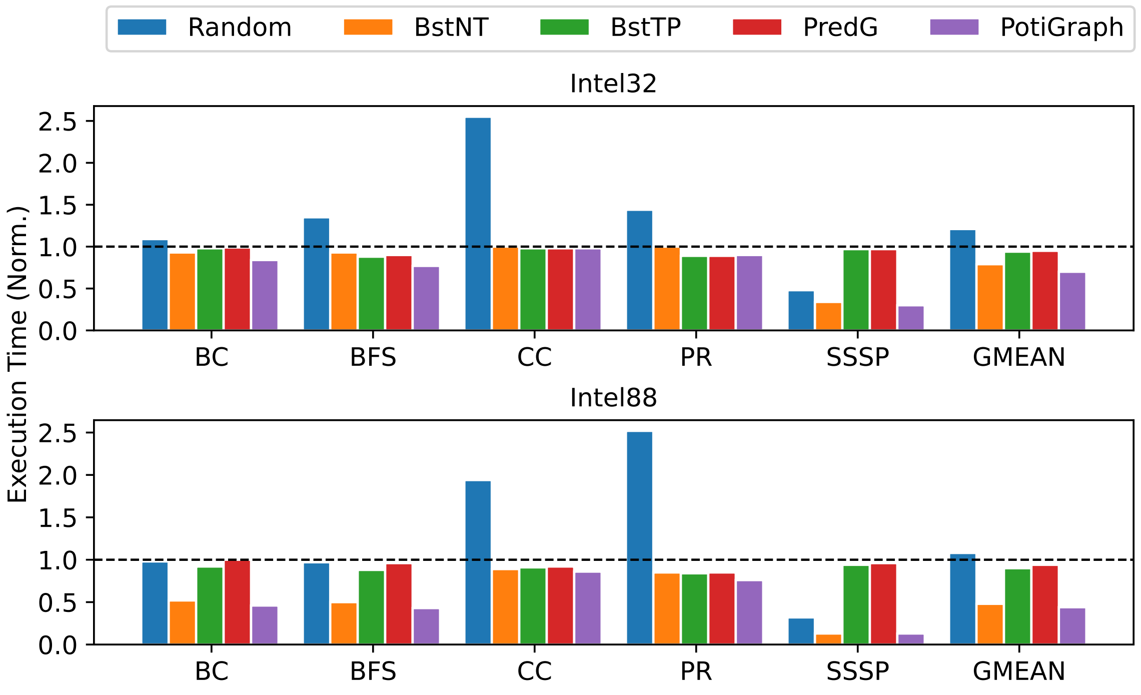
7.3.2 Performance Evaluation

Figures 7.5 and 7.6 demonstrate the execution time for each graph algorithm (*x-axis*) considering the average of all input graphs evaluated along with the overall geometric mean on each NUMA architecture (*GMEAN*). In each subfigure (a and b), results are normalized to the *Default* strategy (the lower, the better).

In Figure 7.5, we compare the *PotiGraph* results with *Default*, *Random*, *BstN*, *BstTP*, and *PredG*. When compared to the *Default* strategy, *PotiGraph* reduces the execution time by up to 87% (*SSSP* on the *Intel88*) and 44% on the overall geometric mean of algorithms, inputs, and machines. These significant improvements are result of a better use of the hardware resources: by employing ideal thread mapping, the *PotiGraph* solutions mitigate cache pollution as each thread can effectively use caches without frequent cache evictions caused by other threads (it reduces by 29% and 38% the misses in L2 and L3 caches over the *Default*[1]); Also, by ideally selecting the page mapping policy,

[1]We obtained hardware counter values from *Intel32* using Intel Performance Counter Monitor (PCM)

Figure 7.5: *PotiGraph* vs. *Default*, *Random*, *BstN*, *BstTP*, and *PredG*. Execution time normalized to *Default*, represented by the black line: ↓ values = better execution time of each strategy.

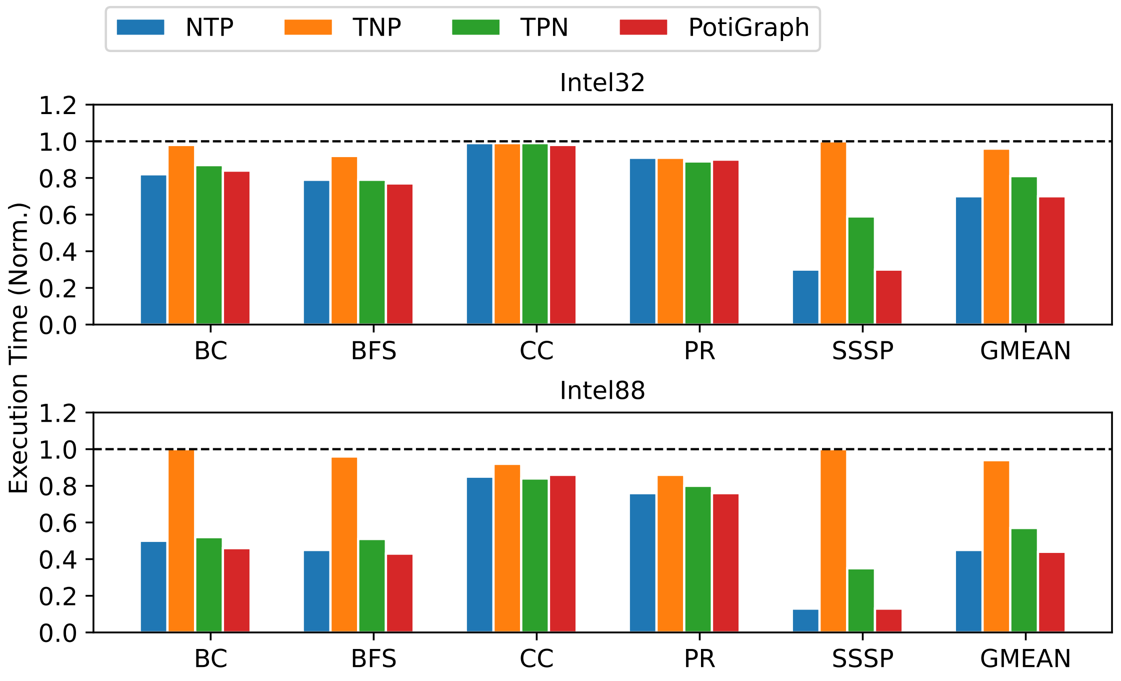


the system can ensure that graph data is stored closer to the processing threads, optimizing memory access patterns, reducing memory latency, and enhancing memory bandwidth utilization (*PotiGraph* reduces by 62% and 21.5% the number of remote memory accesses and accesses to the most congested memory controller over the *Default*). Moreover, when compared to the *Random* strategy, the execution time is reduced by up to 70% for the execution of the *PR* algorithm on the *Intel88*.

Now, let us compare the performance improvements of *PotiGraph* over two offline strategies, *BstN* and *BstTP*, and the hybrid (*PredG*) – also in Figure 7.5 –, *PotiGraph* achieves better performance results across all algorithms, inputs, and architectures. On the overall geometric mean, *PotiGraph* delivers 10%, 39%, and 41% better performance than *BstN*, *BstTP*, and PredG, respectively. Since *BstN* operates with the optimal thread count, its reliance on default thread and page mapping precludes it from archiving the same levels of reduction in remote memory accesses as when optimizing NT+TM+PM (its reduction in remote accesses is 32% lower than the reductions of *PotiGraph*'s solutions). In contrast, although *BstTP* employs the ideal mapping, its falls to select the optimal thread count significantly reduces performance. Notice that the performance improvements of *PotiGraph* are even more significant when compared to *PredG*. Since *PredG* optimizes

(WILLHALM; DEMENTIEV; FAY, 2012).

Figure 7.6: *PotiGraph* vs. *NTP*, *TNP*, and *TPN*. Execution time normalized to *Default*, represented by the black line: ↓ values = better execution time for each strategy.

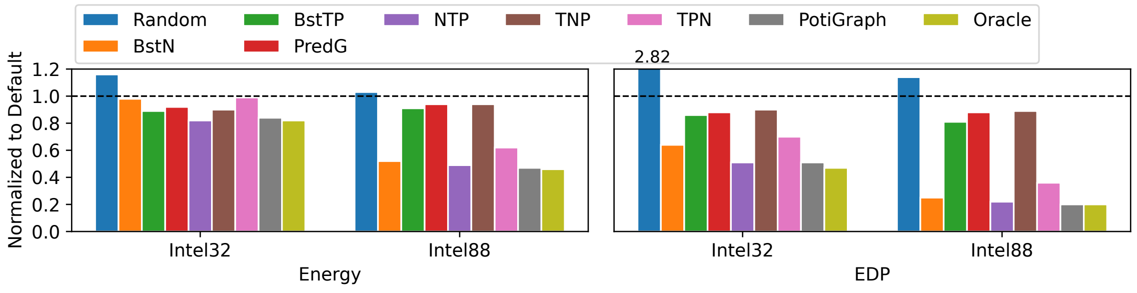


only TM+PM, its best case is limited to *BstTP*. Therefore, these results reinforce the need for a strategy that optimizes the three variables (thread count and thread/page mapping) at the same time.

Another point to consider is the (offline) time taken for *BstN* and *BstTP* to find the best configuration: for instance, they take 1060.15s and 546.30s, respectively, to find an optimal configuration for a single graph application (BC algorithm processing the *road* input on *Intel32*), and must be re-executed for every new input graph. On the other hand, the *Execution Phase* of *PotiGraph* takes only 0.1234 seconds, on average, with a standard deviation of 0.0504, to predict the best NT+TM+PM for all combinations of algorithms, input graphs, and machines. While *PredG* exhibits similar optimization costs to *PotiGraph* (only 0.0492 seconds, on average, with a standard deviation of 0.0102), our results show that it performs poorly compared to *PotiGraph*. It is worth noting that, since *PredG* optimizes only TM+PM, its best achievable solutions yield performance at most equal to that of *BstTP*.

Similarly, *PotiGraph* outperforms the optimization paths proposed by Schwarzrock et al. (SCHWARZROCK et al., 2020) for generic applications in most scenarios, as shown in Figure 7.6. On average of all experiments, *PotiGraph* reduces the execution time by 2%, 42%, and 19% compared to the optimization sequences NTP, TNP, and TPN, respectively. However, it is worth mentioning that, in the same way as *BstN* and

Figure 7.7: Energy and EDP normalized to the *Default*, represented by the black line: ↓ values = better energy and EDP.



BstTP, all these combinations need an exhaustive search for each parameter and must re-execute whenever the algorithm, input graph, or machine changes. For instance, for the BC algorithm processing the *road* input on *Intel32*, *NTP* takes 1374.15s.

7.3.3 The Impact of *PotiGraph* on the Energy and EDP

When proposing techniques to optimize the performance of graph applications running on HPC environments, a key challenge is not significantly increasing energy consumption. Hence, to show that *PotiGraph* can also bring energy savings while optimizing the performance, we compare the energy consumption[2] and the energy-delay product (EDP) of *PotiGraph* with all the previously discussed strategies in Figure 7.7. The results of each strategy consider the geometric mean of all graph applications and input graphs on each multicore architecture. Note that now we use the default execution as the baseline. Because *PotiGraph* can better use the hardware resources by tuning the number of threads, the thread, and page mapping to the input graph, it provides lower energy consumption and better EDP than the other strategies even targeting performance only. When considering the overall geometric mean, it is 42% and 68% better than *Default* and only 2% and 5% distant from the *Oracle* on the two respective metrics.

[2]We collected energy consumed by the DRAM and core domains through the Intel Running Average Power Limit (RAPL) (HÄHNEL et al., 2012).

8 FINAL CONSIDERATIONS AND CONCLUSION

In this thesis, we proposed to optimize the execution of parallel graph applications on NUMA machines. For that, we explored the adjustment of thread mapping (TM) and data mapping (PM) as well as the number of threads (NT) through different Machine Learning (ML) strategies. Our main proposals leveraged the available input graphs' high-level features to implement ML approaches that can learn from different graphs and predict the best solution of TM+PM or NT+TM+PM as a new graph arrives for processing without any further application execution. The effectiveness of our proposals was experimentally proven by optimizing different algorithms, input graphs, and NUMA machines and comparing them with other strategies. On top of that, we also showed that no single solution of TM+PM or NT+TM+PM delivers the best performance for all machines, algorithms, input graphs, or the source vertices (for the algorithms that require a source vertex to start execution). Therefore, improving the graph application's performance over the above variation is a challenge this thesis faces.

8.1 Limitations

Although our proposals have improved the performance and energy savings of the parallel graph applications, they have some limitations, which we describe below.

- ***Learning Overhead.*** Even though our *PredG*, *GraphNroll*, and *PotiGraph* frameworks overcome the limitation of *Graphith* so that they do not require to be re-executed every time a new graph arrives for execution, they yet present an expensive learning overhead. Notice that the *Learning Phase* of such approaches (executed only once before the frameworks are deployed on the target system) perform a Design Space Exploration (DSE) that considers the execution of different algorithms, input graphs, and NUMA machines, which takes considerable time. Based on our experiments, *PredG*, *GraphNroll*, and *PotiGraph* took 66.05, 850.45, and 5059.60 minutes to be deployed of the target machines.
- ***Optimizing Only for New Input Graphs.*** Since *PredG*, *GraphNroll*, and *PotiGraph* learn and decide the best solution based on the input graph's high-level features, they are adaptive only for the cases where a new input graph arrives for execution. Thus, the frameworks must be re-executed if one wants to insert new algorithms or

Table 8.1: Execution time (in seconds) to collect the high-level features of the entire graph.

Graph	**Seconds**
kron	-
road	19.80
twitter	1033.05
urand	893.51
web	323.16
cit-patents	15.31
amazon	3.22
orkut	75.60
youtube	4.40
roadNet-CA	4.25
roadNet-PA	2.75
roadNet-TX	2.58
berkley	3.03
boogle	3.53
wikitalk	4.90

Source: The author.

machines.

- ***Overhead of Collecting the High-Level Features.*** Our proposals assume that the input graph's high-level features are already available in the data source. However, when unavailable, the user must collect the features using a graph analytic tool, such as the NetworKit (STAUDT; SAZONOVS; MEYERHENKE, 2016) used by our frameworks (PyTorch BigGraph in the case of *GraphNroll* (LERER et al., 2019)). Depending on the input graph, it can be expensive. In Fig. 8.1, we summarize the costs of collecting features for all input graphs evaluated in this thesis.
- ***Limited Available NUMA Machines.*** Our experiments are limited to the NUMA machines available on the *Parque Computacional de Alto Desempenho* (PCAD) <http://gppd-hpc.inf.ufrgs.br> from UFRGS.

8.2 Future Works

Graph processing is a comprehensive research topic that different research groups worldwide have studied (BATARFI et al., 2015; YAN et al., 2017; HEIDARI et al., 2018).

Therefore, there is significant scope for research in graph optimization. Next, we describe some perspectives on future research topics that can be derived from this thesis.

- ***Heterogeneous Multi-core.*** Since energy efficiency is one of the most critical constraints of processor design, new multiprocessors with an asymmetric microarchitecture have become available on the market, such as the Apple M1 SoC and the Intel Alder Lake processor (BILBAO; SAEZ; PRIETO-MATIAS, 2023). They are inspired by ARM's big.LITTLE processors but more suitable for high-performance workloads, proving effective for applications like graphics-intensive games and scientific simulation software (SAEZ; PRIETO-MATIAS, 2022). Optimizing the graph processing on such machines may be a challenge since, beyond the irregularity of the graph processing computation, there will be the irregularity of the heterogeneous processors.
- ***NUMA Machine with Hybrid Memories.*** We have observed a rise in works that propose optimizing graph processing on NUMA machines comprised of RAM and NVM memories (DUAN et al., 2019; LIU et al., 2021). However, no work proposes an approach to predict the best solution using only the high-level features of the input graphs.
- ***Dynamic Voltage and Frequency Scaling (DVFS).*** Another room for optimization is to adjust the CPU's frequency levels for graph applications' execution through the DVFS technique. Such a technique leverages time intervals of process inactivity, commonly caused by I/O operations or memory requests. Since the input voltage has a quadratic influence on dynamic power, a system supporting the DVFS technique can take advantage of the CPU idleness (e.g., data-synchronization or communication among threads) to achieve cubic power reduction (OLIVEIRA; LORENZON; BECK, 2018). Therefore, adjusting DVFS for graph execution may yield significant energy-saving benefits.

8.3 List of Publications

Publications related to this proposal:

1. (***Outstanding Paper Award***) Schwarzrock, J., **Rocha, H. M. G. A.**, Lorenzon, A. F., and Beck, A. C. S. (2020, December). Effective Exploration of Thread Throttling and Thread/Page Mapping on NUMA Systems. In 2020 IEEE 22nd International

Conference on High Performance Computing and Communications (HPCC) (pp. 239-246). IEEE. (SCHWARZROCK et al., 2020)

2. **Rocha, H. M. G. A.**, Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2021, March). Boosting Graph Analytics by Tuning Threads and Data Affinity on NUMA Systems. In 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (pp. 161-168). IEEE. (ROCHA et al., 2021)
3. Moori, M. K., **Rocha, H. M. G. D. A.**, Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2021, October). Aumentando a Eficiência na Execução de Algoritmos de Grafos em HPC. In Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho (pp. 132-143). SBC. (MOORI et al., 2021)
4. **Rocha, H. M. G. A.**, Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2022, July). Using Machine Learning to Optimize Graph Execution on NUMA Machines. In 2022 59th ACM/IEEE Design Automation Conference (DAC) (pp. 1-6). IEEE. (ROCHA et al., 2022)
5. Moori, M. K., **Rocha, H. M. G. D. A.**, Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2022) Improving the Efficiency of Graph Algorithm Executions on HPC. Concurrency and Computation: Practice and Experience, e6600. (MOORI et al., 2023b)
6. Moori, M. K., **Rocha, H. M. G. D. A.**, Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2022). Decidindo entre GPU e CPU para Processar Grafos com Base em Métricas de Alto Nível. In Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho. SBC. (MOORI et al., 2022)
7. (***Best Student Paper Award***) Moori, M. K., **Rocha, H. M. G. D. A.**, Matheus A. Silva, Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2023, March). Automatic CPU-GPU Allocation for Graph Execution. In 2023 31th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE. (MOORI et al., 2023a)
8. **Rocha, H. M. G. D. A.**, Querol, V. B., Lorenzon, A. F., and Beck, A. C. S. (2023, November). Optimizing Single-Source Graph Execution on NUMA Machines. In 2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE. (ROCHA et al., 2023)
9. Moori, M. K., **Rocha, H. M. G. D. A.**, Lorenzon, A. F., and Beck, A. C. S. (2024). Allok: A Machine Learning Approach for Efficient Graph Execution on CPU-GPU

Clusters. In The Journal of Supercomputing (SUPE). Springer.

Other publications:

1. Tonetto, R. B., **Hiago, M. D. A.**, Nazar, G. L., and Beck, A. C. S. (2020, July). A machine learning approach for reliability-aware application mapping for heterogeneous multicores. In 2020 57th ACM/IEEE Design Automation Conference (DAC) (pp. 1-6). IEEE. (TONETTO et al., 2020a)
2. **Rocha, H.**, Korol, G., Jordan, M., Krause, A., Silveira, R., Vieira, C., ... and Beck, A. C. S. (2020, August). Firefly: An Open-source Rocket-based Intermittent Framework. In 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI) (pp. 1-6). IEEE. (ROCHA et al., 2020a)
3. Tonetto, R. B., **Hiago, M. D. A.**, Zatt, B., Beck, A. C. S., and Nazar, G. L. (2020, October). A Reliability-Oriented Machine Learning Strategy for Heterogeneous Multicore Application Mapping. In 2020 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). IEEE. (TONETTO et al., 2020b)
4. da Silva, V., Medeiros, T., **Rocha, H.**, Luizelli, M., Rossi, F., Beck, A. C., and Lorenzon, A. (2020, October). Análise da execuçao concorrente de aplicaçoes paralelas em arquiteturas multicore. In Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho (pp. 61-72). SBC. (SILVA et al., 2020)
5. **Rocha, H.**, Schwarzrock, J., Pereira, M., Schnorr, L., Navaux, P., Lorenzon, A., and Beck Filho, A. C. S. (2020, November). AtTune: A Heuristic based Framework for Parallel Applications Autotuning. In Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais (pp. 151-156). SBC. (ROCHA et al., 2020b)
6. (***3° Best Paper Award***) **Rocha, H. M. G. D. A.**, Beck, A. C. S., Maia, S. M., Kreutz, M. E., and Pereira, M. M. (2020, November). A Routing based Genetic Algorithm for Task Mapping on MPSoC. In 2020 X Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE. (ROCHA et al., 2020c)
7. da Silva, V. S., Nogueira, A. G., de Lima, E. C., **de A. Rocha, H. M.**, Serpa, M. S., Luizelli, M. C., ... and Francisco Lorenzon, A. (2021). Smart resource allocation of concurrent execution of parallel applications. Concurrency and Computation: Practice and Experience, e6600. (SILVA et al., 2023)
8. Schwarzrock, J., **Rocha, H. M. G. A.**, Lorenzon, A. F., and Beck, A. C. S. (2022, June). Smoothing on Dynamic Concurrency Throttling. In 2022 IEEE Interna-

tional Parallel and Distributed Processing Symposium Workshop (pp. 1-8). IEEE. (SCHWARZROCK et al., 2022)

9. **Rocha, H. M. G. D. A.**, Beck, A. C. S., Maia, S. M., Kreutz, M. E., and Pereira, M. M. (2023). Using Evolutionary Metaheuristics to Solve the Mapping and Routing Problem in Networks on Chip. Design Automation for Embedded Systems (p. 1-33). Springer. (ROCHA et al., 2023)
10. Moori, M. K., **Rocha, H. M. G. D. A.**, Lorenzon, A. F., and Beck, A. C. S. (2023, November). Searching for the Ideal Number of Threads on Asymmetric Multiprocessors. In 2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE. (MOORI et al., 2023)

REFERENCES

AASAWAT, T. et al. Hygn: Hybrid graph engine for numa. In: IEEE. **2020 IEEE International Conference on Big Data (Big Data)**. [S.l.], 2020. p. 383–390.

AASAWAT, T. K.; REZA, T.; RIPEANU, M. Scale-free graph processing on a numa machine. In: IEEE. **2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)**. [S.l.], 2018. p. 28–36.

AGARWAL, V. et al. Scalable graph exploration on multicore processors. In: IEEE. **SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2010. p. 1–11.

ALESSI, F. et al. Application-level energy awareness for openmp. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2015. p. 219–232.

AMARAL, L. A. N. et al. Classes of small-world networks. **Proceedings of the national academy of sciences**, National Acad Sciences, v. 97, n. 21, p. 11149–11152, 2000.

BARI, M. A. S. et al. Arcs: Adaptive runtime configuration selection for power-constrained openmp applications. In: IEEE. **2016 IEEE International Conference on Cluster Computing**. [S.l.], 2016. p. 461–470.

BATARFI, O. et al. Large scale graph processing systems: survey and an experimental evaluation. **Cluster Computing**, Springer, v. 18, n. 3, p. 1189–1213, 2015.

BEAMER, S.; ASANOVIC, K.; PATTERSON, D. Direction-optimizing breadth-first search. In: IEEE. **SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2012. p. 1–10.

BEAMER, S.; ASANOVIĆ, K.; PATTERSON, D. The gap benchmark suite. **arXiv preprint arXiv:1508.03619**, 2015.

BEAMER, S.; ASANOVIC, K.; PATTERSON, D. Locality exists in graph processing: Workload characterization on an ivy bridge server. In: IEEE. **2015 IEEE International Symposium on Workload Characterization**. [S.l.], 2015. p. 56–65.

BEAMER, S.; ASANOVIĆ, K.; PATTERSON, D. Reducing pagerank communication via propagation blocking. In: IEEE. **2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.], 2017. p. 820–831.

BENIAMINE, D. et al. Tabarnac: visualizing and resolving memory access issues on numa architectures. In: ACM. **Proceedings of the 2nd Workshop on Visual Performance Analysis**. [S.l.], 2015. p. 1.

BILBAO, C.; SAEZ, J. C.; PRIETO-MATIAS, M. Flexible system software scheduling for asymmetric multicore systems with pmcsched: A case for intel alder lake. **CCPE**, Wiley Online Library, p. e7814, 2023.

BONDY, J. A.; MURTY, U. S. R. et al. **Graph theory with applications**. [S.l.]: Macmillan London, 1976.

BRANDES, U.; PICH, C. Centrality estimation in large networks. **International Journal of Bifurcation and Chaos**, World Scientific, v. 17, n. 07, p. 2303–2318, 2007.

BROQUEDIS, F. et al. Structuring the execution of openmp applications for multicore architectures. In: IEEE. **2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)**. [S.l.], 2010. p. 1–10.

BROQUEDIS, F. et al. Forestgomp: an efficient openmp environment for numa architectures. **International Journal of Parallel Programming**, Springer, v. 38, n. 5-6, p. 418–439, 2010.

BULUÇ, A.; GILBERT, J. R. The combinatorial blas: Design, implementation, and applications. **The International Journal of High Performance Computing Applications**, Sage Publications Sage UK: London, England, v. 25, n. 4, p. 496–509, 2011.

CHADHA, G.; MAHLKE, S.; NARAYANASAMY, S. When less is more (limo): controlled parallelism forimproved efficiency. In: **Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems**. [S.l.: s.n.], 2012. p. 141–150.

CHEN, D. Z. Developing algorithms and software for geometric path planning problems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 4es, p. 18–es, 1996.

CORBET, J. **Toward better NUMA scheduling**. 2012. Available from Internet: <https://lwn.net/Articles/486858/>.

CORMEN, T. H. et al. **Introduction to algorithms**. [S.l.]: MIT press, 2022.

CRUZ, E. H.; DIENER, M.; NAVAUX, P. O. Using the translation lookaside buffer to map threads in parallel applications based on shared memory. In: IEEE. **2012 IEEE 26th International Parallel and Distributed Processing Symposium**. [S.l.], 2012. p. 532–543.

CRUZ, E. H.; DIENER, M.; NAVAUX, P. O. Communication-aware thread mapping using the translation lookaside buffer. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 27, n. 17, p. 4970–4992, 2015.

CRUZ, E. H. et al. An efficient algorithm for communication-based task mapping. In: IEEE. **2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.], 2015. p. 207–214.

CURTIS-MAURY, M. et al. Prediction-based power-performance adaptation of multithreaded scientific codes. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 19, n. 10, p. 1396–1410, 2008.

CURTIS-MAURY, M. et al. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: **Proceedings of the 20th annual international conference on Supercomputing**. [S.l.: s.n.], 2006. p. 157–166.

DASHTI, M. et al. Traffic management: a holistic approach to memory placement on numa systems. **ACM SIGARCH Computer Architecture News**, ACM, v. 41, n. 1, p. 381–394, 2013.

DHULIPALA, L.; BLELLOCH, G. E.; SHUN, J. Theoretically efficient parallel graph algorithms can be fast and scalable. **ACM Transactions on Parallel Computing (TOPC)**, ACM New York, NY, USA, v. 8, n. 1, p. 1–70, 2021.

DIENER, M. et al. Affinity-based thread and data mapping in shared memory systems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 49, n. 4, p. 1–38, 2016.

DIENER, M.; CRUZ, E. H.; NAVAUX, P. O. Communication-based mapping using shared pages. In: IEEE. **2013 IEEE 27th International Symposium on Parallel and Distributed Processing**. [S.l.], 2013. p. 700–711.

DIENER, M.; CRUZ, E. H.; NAVAUX, P. O. Locality vs. balance: Exploring data mapping policies on numa systems. In: IEEE. **2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.], 2015. p. 9–16.

DIENER, M. et al. kmaf: Automatic kernel-level management of thread and data affinity. In: **Proceedings of the 23rd international conference on Parallel architectures and compilation**. [S.l.: s.n.], 2014. p. 277–288.

DIENER, M. et al. Characterizing communication and page usage of parallel applications for thread and data mapping. **Performance Evaluation**, Elsevier, v. 88, p. 18–36, 2015.

DOEKEMEIJER, N.; VARBANESCU, A. L. A survey of parallel graph processing frameworks. **Delft University of Technology**, v. 21, p. 5, 2014.

DUAN, Z. et al. Hinuma: Numa-aware data placement and migration in hybrid memory systems. In: IEEE. **2019 IEEE 37th International Conference on Computer Design (ICCD)**. [S.l.], 2019. p. 367–375.

FRASCA, M.; MADDURI, K.; RAGHAVAN, P. Numa-aware graph mining techniques for performance and energy efficiency. In: IEEE. **SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2012. p. 1–11.

FU, H.-H.; LIN, D. K.; TSAI, H.-T. Damping factor in google page ranking. **Applied Stochastic Models in Business and Industry**, Wiley Online Library, v. 22, n. 5-6, p. 431–444, 2006.

GAREY, M. R.; JOHNSON, D. S. **Computers and intractability**. [S.l.]: wh freeman New York, 2002.

GOLDBERG, A. V.; HARRELSON, C. Computing the shortest path: A search meets graph theory. In: CITESEER. **SODA**. [S.l.], 2005. v. 5, p. 156–165.

GOLDBERG, D. E. **Genetic Algorithms in Search Optimization & Machine learning, Second Reprint**. [S.l.]: Pearson Education Asia pte. Ltd, 2000.

GREGOR, D.; LUMSDAINE, A. The parallel bgl: A generic library for distributed graph computations. **Parallel object-oriented scientific computing (POOSC)**, Glasgow, UK, v. 2, n. 1, 2005.

GUI, C.-Y. et al. A survey on graph processing accelerators: Challenges and opportunities. **Journal of Computer Science and Technology**, Springer, v. 34, n. 2, p. 339–371, 2019.

GUREYA, D. et al. Bandwidth-aware page placement in numa. In: IEEE. **2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.], 2020. p. 546–556.

GUREYA, D. et al. Bandwidth-aware page placement in numa. In: IEEE. **IPDPS**. [S.l.], 2020. p. 546–556.

HÄHNEL, M. et al. Measuring energy consumption for short code paths using rapl. **ACM SIGMETRICS Performance Evaluation Review**, ACM New York, NY, USA, v. 40, n. 3, p. 13–17, 2012.

HARISH, P.; NARAYANAN, P. J. Accelerating large graph algorithms on the gpu using cuda. In: SPRINGER. **International conference on high-performance computing**. [S.l.], 2007. p. 197–208.

HEIDARI, S. et al. Scalable graph processing frameworks: A taxonomy and open challenges. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 3, p. 1–53, 2018.

HOLZSCHUHER, F.; PEINL, R. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In: **Proceedings of the Joint EDBT/ICDT 2013 Workshops**. [S.l.: s.n.], 2013. p. 195–204.

HONG, S. et al. Green-marl: a dsl for easy and efficient graph analysis. In: **Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2012. p. 349–362.

HONG, S. et al. Accelerating cuda graph algorithms at maximum warp. **Acm Sigplan Notices**, ACM New York, NY, USA, v. 46, n. 8, p. 267–276, 2011.

HUBERMAN, B. A. **The laws of the Web: Patterns in the ecology of information**. [S.l.]: mit Press, 2003.

IVÁN, G.; GROLMUSZ, V. When the web meets the cell: using personalized pagerank for analyzing protein interaction networks. **Bioinformatics**, Oxford University Press, v. 27, n. 3, p. 405–407, 2011.

JEANNOT, E.; MERCIER, G.; TESSIER, F. Process placement in multicore clusters: Algorithmic issues and practical techniques. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 25, n. 4, p. 993–1002, 2013.

JEONG, H. et al. Lethality and centrality in protein networks. **Nature**, Nature Publishing Group, v. 411, n. 6833, p. 41–42, 2001.

JUNG, C. et al. Adaptive execution techniques for smt multiprocessor architectures. In: **Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming**. [S.l.: s.n.], 2005. p. 236–246.

KALAVRI, V.; VLASSOV, V.; HARIDI, S. High-level programming abstractions for distributed graph processing. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 30, n. 2, p. 305–324, 2017.

KANG, U.; TSOURAKAKIS, C. E.; FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In: IEEE. **2009 Ninth IEEE international conference on data mining**. [S.l.], 2009. p. 229–238.

KLEEN, A. **An NUMA API for Linux**. 2004. Available from Internet: <http://andikleen.de/numaapi3.pdf>.

KLEEN, A. A numa api for linux. **Novel Inc**, 2005.

KLEINBERG, J.; TARDOS, E. **Algorithm design**. [S.l.]: Pearson Education India, 2006.

KNOWLES, J.; CORNE, D. Instance generators and test suites for the multiobjective quadratic assignment problem. In: SPRINGER. **Evolutionary Multi-Criterion Optimization: Second International Conference, EMO 2003, Faro, Portugal, April 8–11, 2003. Proceedings 2**. [S.l.], 2003. p. 295–310.

KRAUSE, A. et al. Nemesys-a showcase of data oriented near memory graph processing. In: **Proceedings of the 2019 International Conference on Management of Data**. [S.l.: s.n.], 2019. p. 1945–1948.

KYROLA, A.; BLELLOCH, G.; GUESTRIN, C. {GraphChi}:{Large-Scale} graph computation on just a {PC}. In: **10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)**. [S.l.: s.n.], 2012. p. 31–46.

LAKHOTIA, K. et al. Gpop: A scalable cache-and memory-efficient framework for graph processing over parts. **ACM Transactions on Parallel Computing (TOPC)**, ACM New York, NY, USA, v. 7, n. 1, p. 1–24, 2020.

LEE, J. et al. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: **Proceedings of the 37th annual international symposium on Computer architecture**. [S.l.: s.n.], 2010. p. 270–279.

LEPERS, B.; QUÉMA, V.; FEDOROVA, A. Thread and memory placement on numa systems: Asymmetry matters. In: **2015 USENIX Annual Technical Conference**. [S.l.: s.n.], 2015. p. 277–289.

LERER, A. et al. Pytorch-biggraph: A large scale graph embedding system. **Proceedings of Machine Learning and Systems**, v. 1, p. 120–131, 2019.

LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford Large Network Dataset Collection**. 2014. <http://snap.stanford.edu/data>.

LI, D. et al. Hybrid mpi/openmp power-aware computing. In: IEEE. **2010 IEEE International Symposium on Parallel & Distributed Processing**. [S.l.], 2010. p. 1–12.

LI, J.; MARTINEZ, J. F. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: IEEE. **The Twelfth International Symposium on High-Performance Computer Architecture, 2006.** [S.l.], 2006. p. 77–87.

LI, R. et al. Precise segmentation of densely interweaving neuron clusters using g-cut. **Nature communications**, Nature Publishing Group, v. 10, n. 1, p. 1–12, 2019.

LINUX. **The Linux Kernel documentation: Linux Scheduler**. 2021. Available from Internet: <https://www.kernel.org/doc/html/latest/scheduler/index.html>.

LIU, W. et al. Hngraph: Parallel graph processing in hybrid memory based numa systems. In: IEEE. **2021 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.], 2021. p. 388–397.

LORENZON, A. F.; FILHO, A. C. S. B. **Parallel Computing Hits the Power Wall: Principles, Challenges, and a Survey of Solutions**. [S.l.]: Springer Nature, 2019.

LORENZON, A. F. et al. Aurora: Seamless optimization of openmp applications. **TPDS**, IEEE, v. 30, n. 5, p. 1007–1021, 2018.

LORENZON, A. F.; SOUZA, J. D.; BECK, A. C. S. Laant: A library to automatically optimize edp for openmp applications. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition, 2017**. [S.l.], 2017. p. 1229–1232.

LOW, Y. et al. Graphlab: A new framework for parallel machine learning. **arXiv preprint arXiv:1408.2041**, 2014.

LU, S. et al. Cache-efficient fork-processing patterns on large graphs. In: **Proceedings of the 2021 International Conference on Management of Data**. [S.l.: s.n.], 2021. p. 1208–1221.

LUAN, G. et al. Online thread auto-tuning for performance improvement and resource saving. **IEEE TPDS**, IEEE, v. 33, n. 12, p. 3746–3759, 2022.

LUCE, R. D.; PERRY, A. D. A method of matrix analysis of group structure. **Psychometrika**, Springer, v. 14, n. 2, p. 95–116, 1949.

MALAVE, S.; SHINDE, S. Reinforced manta ray foraging optimiser for determining the optimal number of threads in multithreaded applications. **IJISAE**, v. 10, n. 3s, p. 17–26, 2022.

MALEWICZ, G. et al. Pregel: a system for large-scale graph processing. In: **Proceedings of the 2010 ACM SIGMOD International Conference on Management of data**. [S.l.: s.n.], 2010. p. 135–146.

MARATHE, A. et al. A run-time system for power-constrained hpc applications. In: SPRINGER. **International conference on high performance computing**. [S.l.], 2015. p. 394–408.

MARATHE, J.; THAKKAR, V.; MUELLER, F. Feedback-directed page placement for ccnuma via hardware-generated memory traces. **Journal of Parallel and Distributed Computing**, Elsevier, v. 70, n. 12, p. 1204–1219, 2010.

MARRETT, K. et al. Recut: a concurrent framework for sparse reconstruction of neuronal morphology. **bioRxiv**, Cold Spring Harbor Laboratory, 2021.

MCCOLL, R. C. et al. A performance evaluation of open source graph databases. In: **Proceedings of the first workshop on Parallel programming for analytics applications**. [S.l.: s.n.], 2014. p. 11–18.

MCCUNE, R. R.; WENINGER, T.; MADEY, G. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 48, n. 2, p. 1–39, 2015.

MITCHELL, J. C. Social networks. **Annual review of anthropology**, JSTOR, v. 3, p. 279–299, 1974.

MOORI, M. K. et al. Searching for the ideal number of threads on asymmetric multiprocessors. In: IEEE. **2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2023. p. 1–8.

MOORI, M. K. et al. Aumentando a eficiência na execuçao de algoritmos de grafos em hpc. In: SBC. **Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2021. p. 132–143.

MOORI, M. K. et al. Decidindo entre gpu e cpu para processar grafos com base em metricas de alto nível. In: SBC. **Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2022. p. 288–299.

MOORI, M. K. et al. Automatic cpu-gpu allocation for graph execution. In: IEEE. **2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.], 2023. p. 27–34.

MOORI, M. K. et al. Improving the efficiency of graph algorithm executions on high-performance computing. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 35, n. 18, p. e7419, 2023.

NGUYEN, D.; LENHARTH, A.; PINGALI, K. A lightweight infrastructure for graph analytics. In: **Proceedings of the twenty-fourth ACM symposium on operating systems principles**. [S.l.: s.n.], 2013. p. 456–471.

OLIVEIRA, C. C. d. **Odin: online, non-intrusive and self-tuning DCT and DVFS to optimize openMP applications**. Dissertation (Master) — Federal University of Rio Grande do Sul, Brazil, Porto Alegre, 2019. 92 p. Masters thesis (M.Sc. degree in computer science) - Institute of Informatics, Federal University of Rio Grande do Sul, Brazil, Porto Alegre, 2019.

OLIVEIRA, C. C. D.; LORENZON, A. F.; BECK, A. C. S. Automatic tuning tlp and dvfs for edp with a non-intrusive genetic algorithm framework. In: IEEE. **2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2018. p. 146–153.

O'MALLEY, T. et al. **Keras Tuner**. 2019. <https://github.com/keras-team/keras-tuner>.

PAPADIMITRIOU, G.; CHATZIDIMITRIOU, A.; GIZOPOULOS, D. Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus. In: IEEE. **2019 IEEE international symposium on high performance computer architecture (HPCA)**. [S.l.], 2019. p. 133–146.

PETERSON, L. L.; DAVIE, B. S. **Computer networks: a systems approach**. [S.l.]: Elsevier, 2007.

POPOV, M.; JIMBOREAN, A.; BLACK-SCHAFFER, D. Efficient thread/page/-parallelism autotuning for numa systems. In: **ACM ICS**. [S.l.: s.n.], 2019. p. 342–353.

PORTERFIELD, A. K. et al. Power measurement and concurrency throttling for energy reduction in openmp programs. In: IEEE. **2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum**. [S.l.], 2013. p. 884–891.

PUSUKURI, K. K.; GUPTA, R.; BHUYAN, L. N. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In: IEEE. **2011 IEEE International Symposium on Workload Characterization**. [S.l.], 2011. p. 116–125.

RAMKUMAR, A. et al. Iterated fast local search algorithm for solving quadratic assignment problems. **Robotics and Computer-Integrated Manufacturing**, Elsevier, v. 24, n. 3, p. 392–401, 2008.

RAPOPORT, A.; HORVATH, W. J. A study of a large sociogram. **Behavioral science**, Wiley Online Library, v. 6, n. 4, p. 279–291, 1961.

ROCHA, H. et al. Firefly: An open-source rocket-based intermittent framework. In: IEEE. **2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2020. p. 1–6.

ROCHA, H. et al. Attune: A heuristic based framework for parallel applications autotuning. In: SBC. **Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. [S.l.], 2020. p. 151–156.

ROCHA, H. M. G. d. A. et al. A routing based genetic algorithm for task mapping on mpsoc. In: IEEE. **2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2020. p. 1–8.

ROCHA, H. M. G. d. A. et al. Optimizing single-source graph execution on numa machines. In: IEEE. **2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2023. p. 1–8.

ROCHA, H. M. G. d. A. et al. Boosting graph analytics by tuning threads and data affinity on numa systems. In: IEEE. **2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.], 2021. p. 161–168.

ROCHA, H. M. G. de A. et al. Using machine learning to optimize graph execution on numa machines. In: **Proceedings of the 59th ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 2022. p. 1027–1032.

ROCHA, H. M. Gomes de A. et al. Using evolutionary metaheuristics to solve the mapping and routing problem in networks on chip. **Design Automation for Embedded Systems**, Springer, p. 1–33, 2023.

ROGERS, I. The google pagerank algorithm and how it works. 2002.

ROSSI, R. A.; AHMED, N. K. The network data repository with interactive graph analytics and visualization. In: **AAAI**. [s.n.], 2015. Available from Internet: <https://networkrepository.com>.

ROSSI, R. A.; AHMED, N. K. An interactive data repository with visual analytics. **ACM SIGKDD Explorations Newsletter**, ACM New York, NY, USA, v. 17, n. 2, p. 37–41, 2016.

ROY, A.; MIHAILOVIC, I.; ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In: **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**. [S.l.: s.n.], 2013. p. 472–488.

SAEZ, J. C.; PRIETO-MATIAS, M. Evaluation of the intel thread director technology on an alder lake processor. In: **ACM SIGOPS APSYS**. [S.l.: s.n.], 2022. p. 61–67.

SAHU, S. et al. The ubiquity of large graphs and surprising challenges of graph processing. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 11, n. 4, p. 420–431, 2017.

SAHU, S. et al. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. **The VLDB Journal**, Springer, v. 29, n. 2, p. 595–618, 2020.

SARKER, S. et al. Critical nodes in river networks. **Scientific Reports**, Springer, v. 9, n. 1, p. 1–11, 2019.

SCHWARZROCK, J. et al. Potential gains in edp by dynamically adapting the number of threads for openmp applications in embedded systems. In: IEEE. **2017 VII Brazilian Symposium on Computing Systems Engineering**. [S.l.], 2017. p. 79–85.

SCHWARZROCK, J. et al. Effective exploration of thread throttling and thread/page mapping on numa systems. In: IEEE. **2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)**. [S.l.], 2020. p. 239–246.

SCHWARZROCK, J. et al. Smoothing on dynamic concurrency throttling. In: IEEE. **2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.], 2022. p. 962–971.

SENANAYAKE, U.; PIRAVEENAN, M.; ZOMAYA, A. The pagerank-index: Going beyond citation counts in quantifying scientific impact of researchers. **PloS one**, Public Library of Science San Francisco, CA USA, v. 10, n. 8, p. e0134794, 2015.

SENSI, D. D. Predicting performance and power consumption of parallel applications. In: IEEE. **2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.], 2016. p. 200–207.

SENSI, D. D.; TORQUATI, M.; DANELUTTO, M. A reconfiguration algorithm for power-aware parallel applications. **ACM Transactions on Architecture and Code Optimization**, ACM New York, NY, USA, v. 13, n. 4, p. 1–25, 2016.

SHAFIK, R. A. et al. Adaptive energy minimization of openmp parallel applications on many-core systems. In: **Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures**. [S.l.: s.n.], 2015. p. 19–24.

SHAO, B.; LI, Y. Parallel processing of graphs. **Graph Data Management: Fundamental Issues and Recent Developments**, Springer, p. 143–162, 2018.

SHUN, J.; BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In: **Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming**. [S.l.: s.n.], 2013. p. 135–146.

SILVA, V. da et al. Análise da execuçao concorrente de aplicaçoes paralelas em arquiteturas multicore. In: SBC. **Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2020. p. 61–72.

SILVA, V. S. da et al. Smart resource allocation of concurrent execution of parallel applications. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 35, n. 17, p. e6600, 2023.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Adaptive, efficient, parallel execution of parallel programs. In: **Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2014. p. 169–180.

STAUDT, C. L.; SAZONOVS, A.; MEYERHENKE, H. Networkit: A tool suite for large-scale complex network analysis. **Network Science**, Cambridge University Press, v. 4, n. 4, p. 508–530, 2016.

STOLZ, S.; SCHLERETH, C. Predicting tie strength with ego network structures. **Journal of Interactive Marketing**, SAGE Publications Sage CA: Los Angeles, CA, v. 54, n. 1, p. 40–52, 2021.

STUTZ, P.; BERNSTEIN, A.; COHEN, W. Signal/collect: graph algorithms for the (semantic) web. In: SPRINGER. **International Semantic Web Conference**. [S.l.], 2010. p. 764–780.

SUJEETH, A. et al. Optiml: an implicitly parallel domain-specific language for machine learning. In: **Proceedings of the 28th International Conference on Machine Learning (ICML-11)**. [S.l.: s.n.], 2011. p. 609–616.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 43, n. 3, p. 277–286, 2008.

SUN, J.; VANDIERENDONCK, H.; NIKOLOPOULOS, D. S. Graphgrind: Addressing load imbalance of graph partitioning. In: **Proceedings of the International Conference on Supercomputing**. [S.l.: s.n.], 2017. p. 1–10.

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In: ACM. **ACM SIGOPS Operating Systems Review**. [S.l.], 2007. v. 41, n. 3, p. 47–58.

TONETTO, R. B. et al. A machine learning approach for reliability-aware application mapping for heterogeneous multicores. In: IEEE. **2020 57th ACM/IEEE Design Automation Conference (DAC)**. [S.l.], 2020. p. 1–6.

TONETTO, R. B. et al. A reliability-oriented machine learning strategy for heterogeneous multicore application mapping. In: IEEE. **2020 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.], 2020. p. 1–5.

TRAHAY, F. et al. Numamma: Numa memory analyzer. In: ACM. **Proceedings of the 47th International Conference on Parallel Processing**. [S.l.], 2018. p. 19.

WANG, W.; DAVIDSON, J. W.; SOFFA, M. L. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In: IEEE. **2016 IEEE International Symposium on High Performance Computer Architecture**. [S.l.], 2016. p. 419–431.

WATTS, D. J.; STROGATZ, S. H. Collective dynamics of ‘small-world’networks. **nature**, Nature Publishing Group, v. 393, n. 6684, p. 440–442, 1998.

WHITE, J. G. et al. The structure of the nervous system of the nematode caenorhabditis elegans: the mind of a worm. **Phil. Trans. R. Soc. Lond**, v. 314, n. 1, p. 340, 1986.

WILLHALM, T.; DEMENTIEV, R.; FAY, P. Intel performance counter monitor-a better way to measure cpu utilization. **Dosegljivo: https://software. intel. com/en-us/articles/intelperformance-countermonitor-a-better-way-to-measure-cpu-utilization.[Dostopano: September 2014]**, 2012.

WILLHALM, T.; DEMENTIEV, R.; FAY, P. **Intel performance counter monitor**. 2016.

YADAV, S.; SHUKLA, S. Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. In: IEEE. **2016 IEEE 6th International conference on advanced computing (IACC)**. [S.l.], 2016. p. 78–83.

YAN, D. et al. Big graph analytics platforms. **Foundations and Trends® in Databases**, Now Publishers, Inc., v. 7, n. 1-2, p. 1–195, 2017.

ZHANG, K.; CHEN, R.; CHEN, H. Numa-aware graph-structured analytics. In: **Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming**. [S.l.: s.n.], 2015. p. 183–193.

ZHANG, Y. et al. Graphit: A high-performance graph dsl. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 2, n. OOPSLA, p. 1–30, 2018.

ZHU, X. et al. Gemini: A {Computation-Centric} distributed graph processing system. In: **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)**. [S.l.: s.n.], 2016. p. 301–316.