This is the accepted manuscript, the final version can be found at: https://ieeexplore.ieee.org/document/10324068
© Copyright 2024 IEEE

2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC) | 979-8-3503-9410-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBESC60926.2023.10324068

# Optimizing Single-Source Graph Execution on NUMA Machines

Hiago Mayk G. de A. Rocha[1], Vicenç Beltran Querol[2], Arthur F. Lorenzon[1], Antonio Carlos S. Beck[1]
[1]Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[2] Barcelona Supercomputing Center – Barcelona, Spain
Email: hmgarocha@inf.ufrgs.br, vbeltran@bsc.es, {aflorezon, caco}@inf.ufrgs.br

***Abstract*—Graphs are data structures capable of representing problems from different domains, such as logistics and social networks. However, these massive graphs stored in high-performance computing (HPC) servers start processing from distinct source vertices (i.e., single-source: such as a different user or message in a social network). Therefore, the amount of vertices and the structure of the sub-graphs to be processed will also change depending on the source, highly influencing the graph algorithm behavior and performance. In this paper, we propose *GraphNroll*, a framework to extract the full potential of these single-source graph algorithms by exploiting the fact that the majority of the HPC servers are on Non-Uniform Memory Access (NUMA) architectures and, therefore, highly dependent of thread and data mapping. *GraphNroll* adjusts the thread and data mapping on NUMA machines by using only the embeddings of the source vertices (i.e., vector representations that encode the graph topology) to find the best configuration in terms of thread/data mapping. For that, *GraphNroll* learns over such embeddings to build a machine learning model capable of predicting the mapping solutions for a new algorithm execution without any profiling. Our results show that *GraphNroll* is, on average, 8% (up to 18%) faster than the Linux OS *Default* and *Random* solutions on three NUMA machines while reducing energy consumption.**

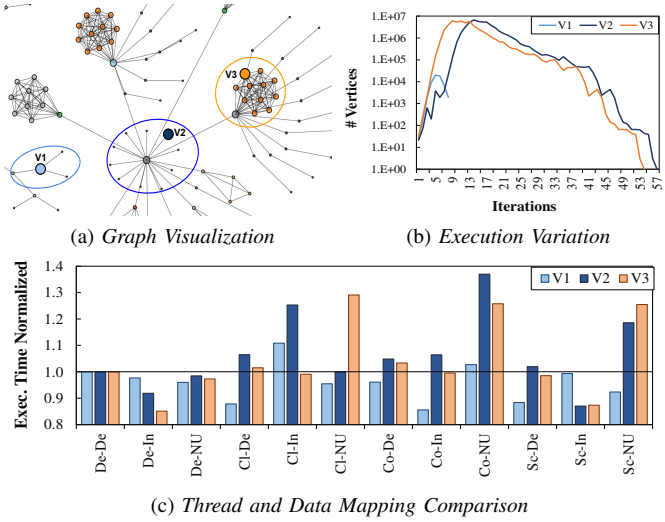
***Index Terms*—Single-Source Algorithms, Graph's High-Level Features, Thread and Data Mapping, NUMA Machines**

## I. Introduction

Graphs are powerful data structures able to model different kinds of problems, such as natural language, transport network, and social network [1]–[3]. The exponential increase in massive graphs has been pushing forward the development of efficient parallel methods, usually executed on modern Non-Uniform Memory Access (NUMA) machines on high-performance computing (HPC) servers. NUMA machines present a complex memory hierarchy where the memory access latency varies according to which nodes' memories are located. Therefore, a high number of remote memory accesses may cause congestion in the memory controllers/interconnect links, significantly decreasing performance [4]–[6]. Since graph applications are communication-bound with irregular communication patterns, their execution on shared memory systems incurs high irregular data dependency and poor data locality, putting extra pressure on such NUMA machines. In this scenario, the most effective way to increase performance is by improving the thread and data placement across the available resources (e.g., cores, nodes, and memories) [5], [7], [8].

However, the few existing NUMA-aware graph processing frameworks [4], [9]–[12] are oblivious to two important aspects: *a)* as real-world graphs present diverse topological structures, e.g., meshes and social networks, optimizing for a specific graph algorithm running over a given input will not necessarily be effective if the input graph changes [2], [6]; and *b)* there is also a variation when executing single-source graph algorithms [13] (i.e., the ones that require a source vertex to start execution). Significant examples of single-source graph algorithms are breadth-first search (BFS) and the single-source shortest path (SSSP), widely applied for telecom network routing, neural image reconstruction, and social network analysis [1]. Therefore, every time a new execution of such algorithms is fired (so the source vertex changes), different parts of the graph with distinct structures and amounts of vertices/edges are processed.

To illustrate the discussion above, we show in Fig. 1 the BFS executing the *web* input graph starting from three different source vertices (*V1*, *V2*, and *V3*). While Fig. 1.a depicts the sub-graphs structures to which each considered vertex belongs, Fig. 1.b shows a distinct number of vertices being processed throughout the iterations starting the BFS execution in these source vertices. For example, (*i*) only a small fraction of the




(a) *Graph Visualization* (b) *Execution Variation*

(c) *Thread and Data Mapping Comparison*

Fig. 1. 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 threads and data mapping combinations. **Legend:** the <thread>/<data> refers to the combination of the thread and data mapping policies.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on July 15,2024 at 15:52:11 UTC from IEEE Xplore. Restrictions apply.

graph is processed when started from *V1*, and (*ii*) although *V1* and *V2* process a significant number of vertices, they vary in the number of vertices and the sub-graphs structures traversed throughout the iterations. With that, the threads process different graph data, changing how they communicate and access the memory regions. The main outcome of these variations is that the best combination of thread and data mapping to execute the algorithm will change according to the source vertex, as illustrated in Fig. 1.c. In this figure, we show the BFS's execution time when processing the *web* input graph from the three considered source vertices (*V1*, *V2*, and *V3* – bars with different colors) over 12 different combinations of thread and data mapping policies (each group of bars) normalized to the Linux' default solution (De-De) – these combinations will be further discussed in Section III. 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.

Therefore, it is evident that some sort of adaptability is necessary so the optimization becomes resilient across different inputs and distinct source vertices that belong to the same input. The problem of adaptability is that most of the time, it comes with the cost of online processing (i.e., overhead from the process of learning at runtime) since it is necessary to collect information and converge to a solution as the application executes. Offline approaches, on the other hand, have zero overhead at runtime (since they are based on optimizing before deployment). However, as just discussed, changes in the source vertex will directly influence the graph algorithm processing, precluding the use of pure offline approaches.

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.

Based on that, we propose *GraphNroll*. It is a Machine Learning (ML) framework for enhancing the single-source graph algorithms' execution time by predicting ideal thread and data mapping policies configuration as the source vertices change for any input graph, algorithm, and NUMA machine. *GraphNroll* comprises two phases: *Learning* and *Execution*. In the *Learning Phase*, GraphNroll builds and trains an ANN model based on the embeddings of representative source vertices and the execution of several algorithms on different NUMA systems. Then, in the *Execution Phase*, before starting the algorithm execution from a given source vertex, *GraphNroll* predicts the most suitable thread and data mapping policies by only using the trained ANNs with the embedding of the given source vertex.

By evaluating three widely used single-source algorithms processing eleven real-world graphs on three different NUMA machines, *GraphNroll* is on average 8% (up to 18%) and 7% (up to 13%) faster than the Linux OS *Default* and *Random* solutions. *GraphNroll* also provides lower energy and energy-delay product (EDP) results, being on average 3% and 7% better than *Default* and 6% and 15% than *Random* on such respective metrics.

## II. Related Work

Although several frameworks have been proposed in the graph processing field [4], [9]–[12], [14]–[16], only a few works optimize the execution of graph applications for NUMA systems, however, they are *(i)* not suitable for different graph algorithms as they use the NUMA topology information to optimize specific graph computation [9], [12], [16]; *(ii)* time-consuming as they require reordering or partitioning the graph's vertices before [4], [10], [11] or during the application execution [9] (which slowdowns execution) for each different graph being processed; or *(iii)* rely on the user to re-write the application code [4], [11]. On top of that, none of them analyses the variation incurred in the execution of single-source algorithms on NUMA machines.

Regarding the performance optimization of generic parallel applications by tuning thread and data mapping on NUMA machines, several works have already proposed offline [5], [7], [17]–[21] or online [22], [23] strategies. However, such strategies are not suitable for single-source graph algorithms since: (*i*) the offline ones require new profiling if the input graph, or even the source vertex, changes, and (*ii*) the online approaches will always come with the intrinsic learning overhead (as explained in the previous section).

***Our Contributions.*** To the best of our knowledge, we are the first to explore the impact of thread and data mapping when executing single-source graph algorithms from different source vertices on NUMA machines. We go even further by proposing *GraphNroll*, a framework capable of finding the best thread and data mapping policies combination as the source vertex changes without any additional application execution after being trained. *GraphNroll* is orthogonal to the other graph processing frameworks, being able to improve even more their performance by setting the ideal mappings policies.

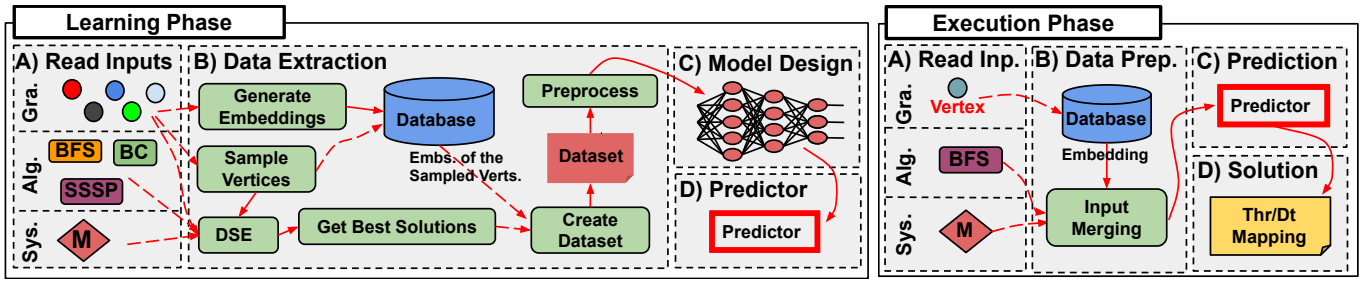
## III. GraphNroll Framework

*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 combination of thread and data mapping for each configuration of algorithm, input graph, and sampled source vertices (explained later) to create a *Predictor* (Fig. 2.a). In the *Execution Phase*, the end-user or data analyst uses the *Predictor* to give the best configuration of thread/data mapping to execute a given single-source algorithm over an input graph starting from a new source vertex (Fig. 2.b).

### A. Learning Phase

To build the *Predictor*, *GraphNroll* learns the best thread and data mapping solutions based on the behavior of the given algorithms and NUMA system by processing different input

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on July 15,2024 at 15:52:11 UTC from IEEE Xplore. Restrictions apply.




(a) (b)

Fig. 2. *GraphNroll* framework overview.

graphs from different source vertices. For that, it applies the following steps, as can be seen in Fig. 2.a:

***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 thread and data mapping for each algorithm running each input graph starting from the sampled source vertex.

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 in a vector representation the vertex-to-vertex relationship). 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 [**?**]. 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. [**?**]: *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 thread and data mapping configurations 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 [1] – 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. [13], 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 thread and data mapping policies combination available on the Linux OS:

- *Thread Mapping. OS's Default* (**De**), which is defined by the Linux's scheduler; *Scatter* (**Sc**), placing threads evenly across the nodes in a round-robin way; *Contiguous* (**Co**), which continuously places threads considering first the physical processing units (PU) and later the logical PUs; and *Close* (**Cl**), which places neighboring threads on the physical and logical PUs of the same core. All policies consider the threads' ID to drive their placements;
- *Data Mapping. First-Touch* (**De**), the Linux's default police, in which the data is placed in the node where the running thread that causes the first-page fault is; *Interleave* (**In**), putting each consecutive data into a different node; *NUMA Balancing* (**NU**), starting by the *De* police, but migrating data to the node where the thread accessing that data is running whenever a page fault rises.

With the *DSE*, *GraphNroll* selects the thread and data mapping 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,

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on July 15,2024 at 15:52:11 UTC from IEEE Xplore. Restrictions apply.

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 overfitting). The final product of the *Learning Phase* is the resulting model, referred to as *Predictor* (see illustration D in Fig.2.a).

### B. Execution Phase

This phase employs the *Predictor* generated in the previous phase to find the best thread and data mapping policies without any further application execution. For that, *GraphNroll* applies the following steps (see Fig. 2.b):

***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 combination of thread and data mapping policies 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.2.b).

### C. Implementation Details

We implemented *GraphNroll* in Python version 3.8.10. Currently, it works for OpenMP applications, so it uses the *GOMP_CPU_AFFINITY* OpenMP environmental to set each specific thread mapping policy (see section II). However, applications implemented with other shared memory libraries (e.g., PThreads) can also be optimized with some extension in the *GraphNroll*'s source code to set thread mapping accordingly. For the data mapping, *GraphNroll* sets the Linux's file $/proc/sys/kernel/numa_balancing$ as 0 for using only First-Touch or 1 for using First-Touch with NUMA Balancing, migrating data as needed. To set the Interleave policy, it uses the *numactl* tool. To get the execution time of each application execution, *GraphNroll* uses the *time.time()* function available on the Python3's **time** module.

## IV. Methodology

***Graphs Algorithms.*** We evaluate *GraphNrolll*'s ability to optimize the following three single-source algorithms available on the GAP Benchmark Suite (GAPBS) repository [13]: ***BFS***, a traversal algorithm that visits all graph's vertices depth by depth; the ***Betweenness Centrality (BC)***, which computes the betweenness centrality score for the graph's vertices, where the BC score of a vertex $i$ is a fraction of the shortest paths between all vertices that pass through $i$; and the ***Single-Source Shortest Path (SSSP)***, an algorithm that computes the shortest paths to all vertices reachable from a source vertex. 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, which cover two comprehensive classes of topologies (i.e., meshes and social networks) and different dimensions (varying from ∼5.49 thousand to ∼23.94 million vertices): amazon, berkley, california, cit-patents, google, orkut, pennsylvania, texas, wikitalk, and youtube [24].

***Execution Environment.*** We performed the experiments in three NUMA machines (using the Linux kernel v. 4.19): ***Intel32***: 2x 8-core Intel Xeon E5-2640 v2 (Ivy Bridge) @2.0GHz, 2-way SMT (2 nodes / 16 cores / 32 threads); ***Intel64***: 4x 8-core Intel Xeon X7550 (Nehalem) @2.0 GHz, 2-way SMT (4 nodes / 32 cores / 64 threads); ***Intel88***: 2x 22-core Intel Xeon E5-2699 v4 (Broadwell) @2.2 GHz, 2-way SMT (2 nodes / 44 cores / 88 threads).

***Evaluated Configurations.*** We compared *GraphNroll* with three different strategies: *(i)* ***Default***, the *regular execution*, which uses the *De-De* thread and data mapping configuration; *(ii)* ***Random***, a random solution generated for each algorithm, input graph, and initial vertex executed on each machine; *(iii)* ***Oracle***, the best thread and data mapping combination 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).

## V. Experimental Results

### A. 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 IV). For that, we show in Fig. 3 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 IV) 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 *Oracle*, 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*. Moreover, *GraphNroll* is only 7% distant from the *Oracle* on the average of all the evaluated machines.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on July 15,2024 at 15:52:11 UTC from IEEE Xplore. Restrictions apply.

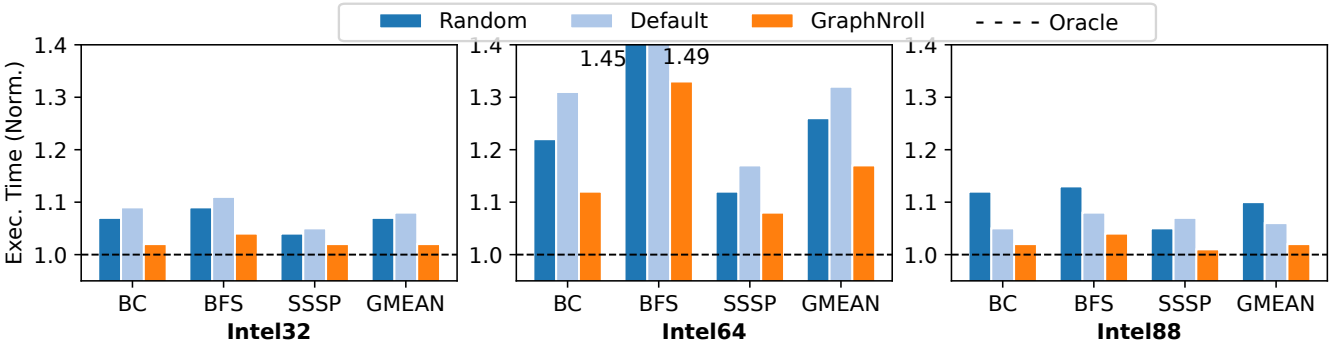



Fig. 3. Execution times of BC, BFS, and SSSP on the evaluated machines (*Intel32*, *Intel64*, and *Intel88*) when using the different compared strategies. Results are normalized to the *Oracle*, so the lower, the better.

TABLE I

*GraphNroll*'S PREDICTIONS FOR 3 DIFFERENT VERTICES OF THE ROAD (ROA), CIT-PATENTS (CIT), AND AMAZON (AMA): THR-DT (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) |

Table I depicts the solutions predicted by *GraphNroll* compared to the *Oracle* ones on each target architecture. *Thr-Dt* 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 work: *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. Therefore, we can see that the ideal thread and data mapping combination changes, and they do so for these main reasons:

- *Source-Vertices*. If the vertex changes, the single-source algorithm will process different amounts of data with distinct structures. This scenario can be observed for the *bc-cit* starting from *V1* and *V2* on the *Intel32* machine;
- *Input Graphs*. If the input graph changes, the algorithm may process a graph with a completely different structure. A representative example is the *bfs* processing the *road* and *cit* on *Intel32* (both from *V1*), being a mesh and a citation graph, respectively;
- *Graph Algorithms*. Even if the input graph and source vertex remain the same, each algorithm traverses the input graphs differently. This scenario can be observed for the *bc* and *bfs* processing the *road* graph from *V3* on *Intel64*;
- *NUMA Machines*. Each NUMA machine presents a distinct memory hierarchy organization, so the performance of the same algorithm will very likely change on different NUMA machines. For example, when executing *bfs* algorithm over the *cit* from *V1* on *Intel32* and on *Intel64* give different mapping solutions.

### *B. Costs of the Compared Solutions*

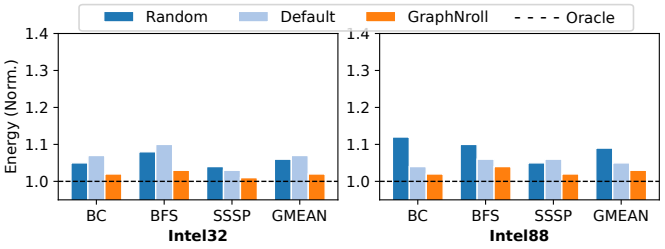
While executing single-source algorithms using the *Default* or *Random* thread and data mapping is straightforward with no training overheads, such strategies lack solution quality. As we show in Fig. 3, *Default* and *Random* are up to 49% and 45% (BFS on *Intel64*) far from the *Oracle*, respectively. 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 thread and data mapping 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 average inference time of 30 predictions for all machines, algorithms, input graphs, and source vertices is 0.0665 seconds, with a standard deviation of 0.0161.

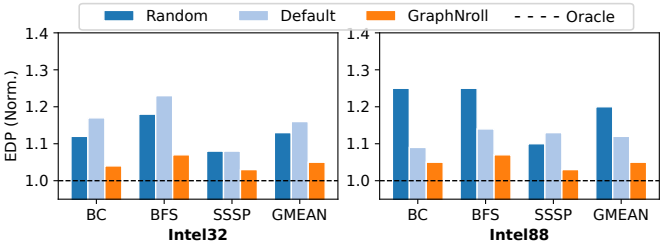
### *C. Impact of GraphNroll on the Energy and Energy-Delay Product*

In Fig.4, 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 hw 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) [25],

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on July 15,2024 at 15:52:11 UTC from IEEE Xplore. Restrictions apply.




(a) Energy




(b) EDP

Fig. 4. Energy and EDP of BC, BFS, and SSSP algorithms on *Intel32* and *Intel88* when using the different compared strategies. Results are normalized to the *Oracle*, so the lower, the better.

[26]. Similar to the performance results, each bar shows the average of all evaluated input graphs and 10 source vertices and are normalized to the *Oracle*, 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.

## VI. Conclusion

This paper proposed *GraphNroll*, a graph processing framework developed to extract the full optimization potential of single-source graph algorithms. *GraphNroll* uses the embeddings of the input graphs to feed an ML capable of predicting the ideal thread and data mapping as the single-source algorithms are executed from new source vertices without any additional application execution. Our results showed that *GraphNroll* overcomes the Linux *Default* and *Random* solutions by an average of 8% (up to 18%), while yet presenting an energy consumption of only 2%, on average, distant from the *Oracle*. As future work, we intend to analyze *GraphNroll*'s behavior on NUMA systems with more than 4 nodes.

## Acknowledgment

This study was financed in part by the CAPES - Finance Code 001, FAPESP grant #2021/06825-8, FAPERGS, and CNPq. Some experiments used the PCAD infrastructure, http://gppd-hpc.inf.ufrgs.br, at INF/UFRGS.

## References

[1] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM TOPC*, vol. 8, no. 1, pp. 1–70, 2021.
[2] M. K. Moori, H. M. G. d. A. Rocha, M. A. Silva, J. Schwarzrock, A. F. Lorenzon, and A. C. S. Beck, "Automatic cpu-gpu allocation for graph execution," in *PDP*. IEEE, 2023, pp. 27–34.
[3] M. K. Moori, H. M. G. de A. Rocha, J. Schwarzrock, A. F. Lorenzon, and A. C. S. Beck, "Improving the efficiency of graph algorithm executions on high-performance computing," *CCPE*, vol. 35, no. 18, p. e7419, 2023.
[4] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *ACM SIGPLAN PPoPP*, 2015, pp. 183–193.
[5] J. Schwarzrock, H. M. G. d. A. Rocha, A. C. S. Beck, and A. F. Lorenzon, "Effective exploration of thread throttling and thread/page mapping on numa systems," in *HPCC/SmartCity/DSS*. IEEE, 2020, pp. 239–246.
[6] H. M. G. d. A. Rocha, J. Schwarzrock, A. F. Lorenzon, and A. C. S. Beck, "Boosting graph analytics by tuning threads and data affinity on numa systems," in *PDP*. IEEE, 2021, pp. 161–168.
[7] M. Diener, E. H. Cruz, and P. O. Navaux, "Communication-based mapping using shared pages," in *IPDPS*. IEEE, 2013, pp. 700–711.
[8] H. M. G. de A. Rocha, J. Schwarzrock, A. F. Lorenzon, and A. C. S. Beck, "Using machine learning to optimize graph execution on numa machines," in *59th ACM/IEEE DAC*, 2022, pp. 1027–1032.
[9] M. Frasca, K. Madduri, and P. Raghavan, "Numa-aware graph mining techniques for performance and energy efficiency," in *SC*. IEEE, 2012, pp. 1–11.
[10] T. Aasawat, T. Reza, K. Yoshizoe, and M. Ripeanu, "Hygn: Hybrid graph engine for numa," in *IEEE BigData*, 2020, pp. 383–390.
[11] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *ICS*. ACM, 2017.
[12] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *SC*. IEEE, 2010, pp. 1–11.
[13] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[14] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN PPoPP*, 2013, pp. 135–146.
[15] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *SOSP*. ACM, 2013, pp. 472–488.
[16] Y. Yasui, K. Fujisawa, E. L. Goh, J. Baron, A. Sugiura, and T. Uchiyama, "Numa-aware scalable graph traversal on sgi uv systems," in *Workshop on High Performance Graph Processing*. ACM, 2016, pp. 19–26.
[17] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 47–58.
[18] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of openmp applications for multicore architectures," in *IPDPS*. IEEE, 2010, pp. 1–10.
[19] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß, "kmaf: automatic kernel-level management of thread and data affinity," in *ACM PACT*, 2014, pp. 277–288.
[20] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88, pp. 18–36, 2015.
[21] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in numa architectures: A statistical learning approach," in *ICPP*, 2019, pp. 1–10.
[22] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus," in *HPCA*. IEEE, 2019, pp. 133–146.
[23] I. Sánchez Barrera, D. Black-Schaffer, M. Casas, M. Moretó, A. Stupnikova, and M. Popov, "Modeling and optimizing numa effects and prefetching with machine learning," in *ICS*. ACM, 2020, pp. 1–13.
[24] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[25] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.
[26] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "On the influence of static power consumption in multicore embedded systems," in *ISCAS*. IEEE, 2015, pp. 1374–1377.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on July 15,2024 at 15:52:11 UTC from IEEE Xplore. Restrictions apply.