PDF Download
3711708.3723452.pdf
01 February 2026
Total Citations: 0
Total Downloads: 183

Latest updates: https://dl.acm.org/doi/10.1145/3711708.3723452

Published: 01 March 2025

Citation in BibTeX format

FCPC '25: 1st FastCode Programming Challenge
March 1 - 5, 2025
NV, Las Vegas, USA

Conference Sponsors:
SIGHPC
SIGPLAN

SHORT-PAPER

Cache-optimized BFS on multi-core CPUs

SALVATORE DOMENICO ANDALORO, University of Trento, Trento, TN, Italy

THOMAS PASQUALI, University of Trento, Trento, TN, Italy

FLAVIO VELLA, University of Trento, Trento, TN, Italy

Open Access Support provided by:

University of Trento

FCPC '25: Proceedings of the 1st FastCode Programming Challenge (March 2025)
https://doi.org/10.1145/3711708.3723452
ISBN: 9798400714467

Cache-optimized BFS on multi-core CPUs

Salvatore D. Andaloro
University of Trento
Trento, Italy
salvatore.andaloro@studenti.unitn.it

Thomas Pasquali*
University of Trento
Trento, Italy
thomas.pasquali@unitn.it

Flavio Vella
University of Trento
Trento, Italy
flavio.vella@unitn.it

Abstract

Breadth-First Search (BFS) performance on shared-memory systems is often limited by irregular memory access and cache inefficiencies. This work presents two optimizations for BFS graph traversal: a bitmap-based algorithm designed for small-diameter graphs and MergedCSR, a graph storage format that improves cache locality for large-scale graphs. Experimental results on real-world datasets show an average 1.3× speedup over a state-of-the-art implementation, with MergedCSR reducing RAM accesses by approximately 15%.

Keywords: graph, algorithm, breadth-first search, parallel computing, multi-core CPU

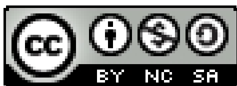
1 Introduction

Graph analysis has become fundamental across various fields such as social network analysis [11], bioinformatics [15, 17], logistics [5, 16], machine learning [10] and others [4]. Breadth-First Search (BFS) is a fundamental graph traversal algorithm used in many other graph algorithms [9, 12].

Despite its simplicity, an efficient implementation of a parallel Breadth-First Search presents several challenges. Achieving effective load balancing across multiple processing units, particularly in multi-core CPU environments, is a non-trivial task due to the irregular and unpredictable structure of graphs [7, 8]. This irregularity can lead to unbalanced workload distribution. Numerous optimization strategies have been proposed in the literature to address these challenges [1, 2, 6]. These can broadly be classified as follows: techniques that involve reordering vertices in memory to improve locality, methods that alter the sequence in which vertices are accessed and strategies that involve changing the data structures used during the BFS.

This work presents two BFS optimizations: the first utilizes bitmaps to leverage available parallelism, while the second is based on a graph storage format specifically designed for large-scale graphs, called MergedCSR. MergedCSR aims to improve the spatial locality of graph data and to reduce the amount of cache misses. Our implementations are executed on multi-core CPUs using OpenMP and are evaluated on real-world undirected and unweighted graphs drawn from different fields, providing a comprehensive assessment of their performance across a variety of practical scenarios. The code is open-source and available at https://github.com/HicrestLaboratory/MergedCSR.

*Corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.
FCPC '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright is held by the owner/author(s).
ACM ISBN 979-8-4007-1446-7/2025/03.
https://doi.org/10.1145/3711708.3723452

2 Background

The BFS algorithm begins at a *source* vertex and includes all its adjacent nodes in the frontier. Subsequently, it explores the neighbors of the nodes in the frontier and stores, for example, their distance from the *source* vertex, continuing this process incrementally until every node has been visited.

In general, BFS performance is heavily influenced by the structural properties of the graph, such as diameter, average degree and sparsity. Low diameter graphs often exhibit power-law degree distributions and small-world phenomena [14]. Beamer et al.[2] proposed two strategies to populate frontiers efficiently: Top-Down and Bottom-Up. Top-Down iterates over the vertices in the current frontier, examining their neighbors to populate the next frontier. This approach is generally effective when the size of the frontier is small, as there are few edges to explore. Conversely, Bottom-Up iterates over unvisited vertices, identifying those that have neighbors in the current frontier. This strategy is advantageous when the frontier is large, as it reduces unnecessary edge examinations and leverages available parallelism more effectively. The transition between the two approaches is based on the number of edges to check from the frontier, the number of edges to check from unexplored vertices and the number of vertices in the frontier. The threshold is tunable using two parameters, α and β. We refer to this adaptive algorithm as Hybrid-BFS. Building on this foundation, Arai et al. [1] proposed a dynamic strategy to compute optimal values for α and β based on graph characteristics and a technique called forest pruning. The goal of the latter is to reduce the number of vertices to be explored by pruning the vertices for which parents can be determined before a *source* vertex is given. This strategy can be applied only on tree-structured subgraphs.

Various storage formats have been developed to represent graphs. Among these, the compressed sparse row (CSR) format is widely adopted for its simplicity and compactness. This format stores the graph using two arrays: one containing the IDs of the neighbors of all nodes, named `col_idx`, and one for pointers that indicate where the neighbors of each node start in `col_idx`, named `row_ptr`. Torok [13] proposed to merge the two CSR arrays into a single structure. In Section 3.2, we extend this format idea to enhance BFS performance for large-diameter graphs.

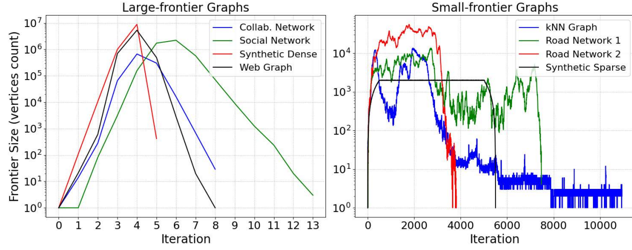


Figure 1. Frontier sizes. These plots show the number of vertices in each frontier for increasing distance from the *source* (y-axis is in log scale). On the left we have small-diameter graphs, on the right, large-diameter graphs.

An important factor to consider is the number of vertices in the frontier during each iteration of the BFS algorithm. As shown in Fig. 1, there is a clear relationship between the graph's diameter and the size of the frontier. Specifically, graphs with smaller diameters tend to generate larger frontiers, while graphs with larger diameters usually generate smaller frontiers. The typical sizes of the frontier heavily influence the memory usage and the amount of cache misses. Consequently, we will categorize datasets into small-frontier and large-frontier. Since all our graphs with limited diameters have considerable frontier sizes, we will interchange these terms to denote graphs types (respectively for large diameters and small frontiers).

3 Methodology

In our work, we mainly focus on the design of efficient data structures for BFS graph traversal. Given that the characteristics of graphs heavily influence the choice of data structures, we classify graphs into one of two categories: (i) graphs that generate large frontiers during the BFS traversal, or (ii) graphs that generate small frontiers. During a BFS on large-frontier graphs, we extensively employ the bitmap data structure and apply the Hybrid-BFS algorithm [2]. In contrast, during the BFS of small-frontier graphs, we store the graph in the MergedCSR format and perform only Top-Down steps. Both implementations are bulk-synchronous, meaning that all vertices at a given distance from the *source* are visited before exploring vertices at the next level.

3.1 Bitmap-based method for Large-Frontier Graphs

Exploring a small-diameter graph requires only a few frontier expansions, since the distance between any two vertices is by definition small. Typically, the first and last iterations have small frontiers, while the mid iterations have large frontiers (see the left plot in Fig. 1). To optimize for the dense mid steps, we use bitmaps to store visited vertices and frontiers. This allows efficient node-parallelism for both Top-Down and Bottom-Up steps.

3.2 MergedCSR method for Small-Frontier Graphs

Graphs with small frontiers present distinct challenges. First, they require many more iterations and they typically have a large number of vertices. Consequently, the memory footprint of the `row_ptr` and `col_idx` array often exceeds the cache capacity of commercial-grade processors. Moreover, for small-frontier graphs, the Top-Down approach is generally preferred over Bottom-Up, since the Bottom-Up approach is used only when the frontier is large. In the traditional CSR format, a Top-Down step requires accessing three data structures: (i) `row_ptr`, to retrieve the start of a vertex's adjacency list, (ii) `col_idx`, to traverse neighbors, (iii) `distances`, to keep track of visited vertices and their distance from the *source*. Each time a new vertex is explored during a Top-Down step, all three data structures need to be accessed. These memory accesses often result in cache misses, significantly impacting the algorithm's performance. This highlights the need for a more memory-efficient data structure. To address these challenges and reduce cache misses during a BFS traversal, we propose the MergedCSR format, a cache-efficient graph representation.

3.3 The MergedCSR Format

Inspired by Torok [13], MergedCSR optimizes the standard CSR format by merging the vertex and neighbor arrays into a single data structure. Fig. 2 gives an example of this format. The MergedCSR array is a modified version of the `col_idx` array, in which each vertex ID is replaced with a pointer to the start of the corresponding vertex data within the MergedCSR array. Additionally, two entries are added for each vertex: one to store the vertex degree and the other to store the distance from the *source*. The size occupied by the MergedCSR array is therefore $2|V| + |E|$. At the end of the BFS traversal, distances are extracted from the MergedCSR array and stored in the `distances` array using a modified version of `row_ptr`, which accounts for the additional information inserted in the MergedCSR array. Section 4.2 underlines the ability of MergedCSR to reduce cache misses. Our adaptation of the MergedCSR format is tailored for a parallel shared-memory BFS implementation. Unlike Torok's approach [13], which focuses primarily on improving spatial locality, we integrate algorithm-specific data (e.g. distance, parent) directly into the merged array. These modifications make MergedCSR adaptable to different algorithm requirements while preserving its benefits.

3.4 Implementation Details

This section reviews the key implementation choices made to optimize the runtime of BFS traversal.

Method Selection: The method selection is guided by the following empirical heuristic: if $\frac{|E|}{|V|} < 10$, we apply the method for small-frontier graphs (Section 3.2); otherwise, we apply the method for large-frontier graphs (Section 3.1).

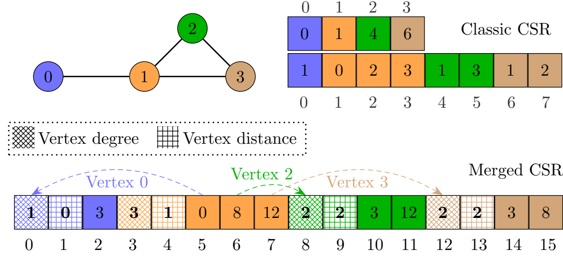


Figure 2. Example of the MergedCSR format. Dashed arrows show the semantics of vertex 1 adjacency list. We assume that a BFS has already been completed considering *source*=0.

Vectorization: SIMD instructions were applied wherever possible. Vectorization usually applies to simple loops, such as those iterating over a bitmap and the process of collecting distances from the MergedCSR format.

Leaf pruning: Vertices with a single neighbor are not added to the next frontier, as their sole neighbor would have already been visited. However, this optimization proved inefficient for graphs with large frontiers. The cost of checking vertex degrees outweighed the benefit, as the conditional statement interfered with the compiler's ability to vectorize loops.

Workload balancing: Multiple OpenMP scheduling strategies were explored: (i) *static* pre-allocates iterations to threads; (ii) *dynamic* lets threads request chunks as they are free; (iii) *guided* assigns progressively smaller chunks; (iv) *auto* lets the compiler decide. What we observed by changing strategy is that, despite the unbalanced nature of graph traversal, static scheduling performed similarly to dynamic and guided. This is likely due to improved cache locality within cores (when using a *static* scheduling) combined with the overhead associated to more complex scheduling methods. We also tested splitting high-degree vertices (hubs) into smaller ones to improve balance, but this had no measurable impact on runtime, likely for the same reasons and the added overhead of handling the newly introduced vertices.

Bitmap inefficiencies: During the initial and final BFS iterations, the bitmap representation is inefficient because the frontiers contain a small number of vertices, therefore the bitmaps are sparsely populated. However, we found the overhead of predicting and switching formats dynamically negated the advantages for most datasets.

Hybrid-BFS Parameters: The α and β parameters were selected as 4 and 24 (resp.) after performing a sensitive analysis over all datasets. Theoretically, these parameters can be selected dynamically, however, this added complexity is not compensated by the performance gains.

Bitmap Prefetching: To optimize memory access, bitmaps for the current frontier, next frontier, and visited vertices are initialized in a parallel block with static scheduling. This ensures that chunks are preloaded into caches close to the cores that will access them during the BFS traversal.

Graph Name	$\|V\|$	$\|E\|$	Diameter
Social Network	4.8M	84M	Small
Web Graph	6.6M	294.3M	Small
Collaboration Network	1.1M	110.4M	Small
Synthetic Dense	9.9M	980.1M	Small
Road Network 1	21.9M	58.2M	Large
Road Network 2	87.0M	112.9M	Large
kNN Graph	24.6M	154.3M	Large
Synthetic Sparse	9.9M	39.2M	Large

Table 1. Datasets Characteristics.

4 Evaluation

This section evaluates the performance (runtime), cache utilization (cache misses), and scalability (speedup with increasing thread count) of our implementations.

To the best of our knowledge, no widely recognized state-of-the-art implementation exists for distance-based BFS on shared memory systems. Consequently, we utilized the parent-based BFS implementation from the GAP Benchmark Suite [2, 3][1] as baseline. This implementation returns a list of each vertex's parent, rather than its distance from the *source*. To ensure a fair comparison, we extended our implementations to parent-based BFS and evaluated both versions.

For cache utilization, we created a third implementation as a baseline, which employs the classical CSR format, stores frontiers in lists, and tracks visited vertices using a bitmap. This comparison highlights the efficiency gains introduced by the use of MergedCSR format.

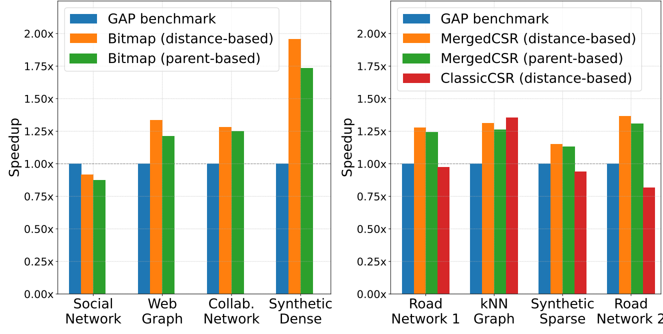
Our evaluation is based on the datasets provided by the PPoPP'25 FastCode Challenge[2], summarized in Table 1. The algorithms are implemented in C++, compiled with g++ 12.2.0, and use OpenMP 4.5 for parallel execution. Cache utilization was profiled using perf v4.18. Experiments were conducted on a dedicated node of the Leonardo Supercomputer[3], featuring an Intel Xeon Platinum 8358 CPU @ 2.6 GHz (32 cores), with 48 KiB L1d cache, 32 KiB L1i cache, 1.28 MiB L2 cache per core, 48 MiB shared L3 cache, and 100 GB DDR4 RAM @ 3.2 GHz.

We executed each algorithm 10 times, selecting a random *source* vertex for each run[4]. The reported value is the arithmetic mean of the results.

4.1 Performances

Fig. 3 compares the performance of our implementations for distance-based and parent-based BFS. For large-frontier graphs, our approach consistently uses bitmaps, unlike GAP, which switches between bitmaps and lists. This is beneficial as only a few Top-Down iterations occur in our datasets, though social network graphs, requiring more Top-Down

[1] Available at https://github.com/sbeamer/gapbs
[2] https://fastcode.org/events/fastcode-challenge/spe4ic/#dataset-diversity
[3] https://leonardo-supercomputer.cineca.eu/it/home-it/
[4] The random selection is deterministic, ensuring all programs use the same set of *source* vertices.



(a) Large-frontier graphs, using 32 threads.

(b) Small-frontier graphs, using 24 threads.

Figure 3. Speedup relative to GAP Benchmark. Bar labels contain the traversed edges per second. "ClassicCSR" refers to the third implementation described in Section 4.

iterations (see Fig. 1), show some performance decrease. Overall, our implementation achieves a 1.32× speedup[5] on large-frontier graphs. For small-frontier graphs, MergedCSR achieves an average 1.27× speedup for distance-based BFS, with the greatest benefits in high-vertex-count graphs. In the Synthetic Sparse dataset, the vertex data fits in L3 cache, limiting the impact of our cache-optimized approach. Fig. 3b highlights the benefits of the MergedCSR format compared to classical CSR. For parent-based BFS, the MergedCSR format was modified to store for each vertex the original vertex ID, the parent ID (instead of the distance), and the vertex degree, increasing spatial complexity from $2|V| + |E|$ to $3|V| + |E|$. The results of parent-based implementation indicate minimal differences compared to distance-based one, achieving an overall 1.23× speedup.

4.2 Cache Utilization

Frequent cache misses introduce significant memory access latency, negatively impacting execution speed. To profile cache miss percentages, we used the *perf* tool[6]. We recall that L1 and L2 cache are private per core, whereas L3 (LLC) is shared among all cores. Table 2 shows the reduction in cache misses achieved with MergedCSR compared to a baseline that uses the classic CSR format. The most significant gains occur in the L1 miss ratio, where improved locality reduces higher-level memory accesses. This is further confirmed by the 15% reduction in L3 miss rate, aligning with the performance gains in Section 4.1.

4.3 Scalability Analysis

For graphs with large frontiers (Fig. 4), speedups closely follow the ideal curve, indicating the effectiveness of parallelism. In contrast, graphs with small frontiers (Fig. 5) present a greater challenge, as the improvements from additional threads become less pronounced. This is likely due to the relatively small frontier sizes which limit the effectiveness of parallelism.

	ClassicCSR	MergedCSR	Improvement
L1	44.9% (46.6%)	38.6% (38.3%)	-14.1% (-18.0%)
L2	19.2% (62.7%)	14.5% (66.4%)	-24.2% (5.9%)
L3	14.3% (64.9%)	12.2% (65.2%)	**-14.7%** (0.4%)

Table 2. Cache miss Rate (Ratio) using 24 threads. Values represent the geometric mean across the small-frontier graphs. The relative improvement is computed as the ratio of MergedCSR results to those of the ClassicCSR (lower is better).

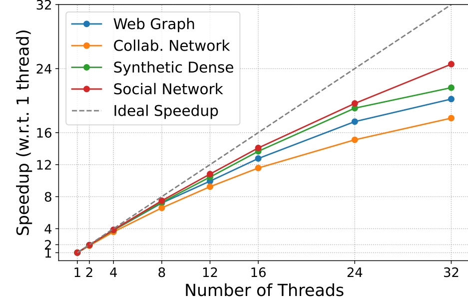


Figure 4. Bitmap-based method (Section 3.1) Scalability.

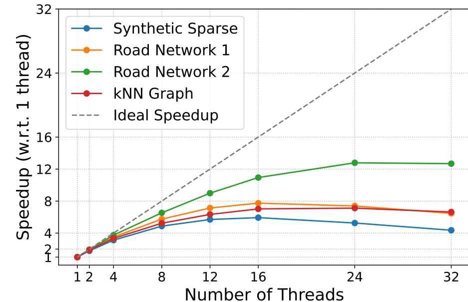


Figure 5. MergedCSR method (Section 3.2) Scalability.

5 Conclusion

We presented two optimized BFS implementations designed for multi-core CPUs, focusing on efficient parallelism and memory management. The bitmap-based approach achieves high performance for large-frontier graphs, while MergedCSR reduces cache misses, improving performance for small-frontier graphs. Experimental results demonstrate consistent speedups over baseline implementations. The MergedCSR format has proven to be effective for storing algorithm-specific data, making it a promising choice for broader applications. As future work, we plan to extend MergedCSR to other graph algorithms, including triangle counting, and biconnected components, as well as evaluate its suitability for distributed memory systems.

Acknowledgments

This project is partially supported by the EuroHPC JU project within Net4Exa project under grant agreement No 101175702.

[5]This and following speedups are computed using the geometric mean.
[6]Tracking events: *L1-dcache-load-misses, L1-dcache-loads, l2_rqsts.all_demand_data_rd, l2_rqsts.demand_data_rd_miss, LLC-load-misses, LLC-loads, mem_inst_retired.all_loads.*

References

[1] Junya Arai, Masahiro Nakao, Yuto Inoue, Kanto Teranishi, Koji Ueno, Keiichiro Yamamura, Mitsuhisa Sato, and Katsuki Fujisawa. 2024. Doubling Graph Traversal Efficiency to 198 TeraTEPS on the Supercomputer Fugaku. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[2] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. (2012).

[3] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC] https://arxiv.org/abs/1508.03619

[4] J. Dörpinghaus, T. Hübenthal, and D. Stepanov. 2024. A Novel DFS/BFS Approach Towards Link Prediction. *arXiv preprint* (2024). https://arxiv.org/abs/2409.11687

[5] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory.. In *SODA*, Vol. 5. 156–165.

[6] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C. Hart. 2012. Edge v. Node Parallelism for Graph Centrality Metrics. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 15–28. https://doi.org/10.1016/B978-0-12-385963-1.00002-2

[7] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel graph analytics. *Commun. ACM* 59, 5 (April 2016), 78–87. https://doi.org/10.1145/2901919

[8] ANDREW LUMSDAINE, DOUGLAS GREGOR, BRUCE HENDRICKSON, and JONATHAN BERRY. 2007. CHALLENGES IN PARALLEL GRAPH PROCESSING. *Parallel Processing Letters* 17, 01 (2007), 5–20. https://doi.org/10.1142/S0129626407002843 arXiv:https://doi.org/10.1142/S0129626407002843

[9] M. E. J. Newman. 2010. *Networks: an introduction.* Oxford University Press, Oxford; New York. http://www.amazon.com/Networks-An-Introduction-Mark-Newman/dp/0199206651/ref=sr_1_5?ie=UTF8&qid=1352896678&sr=8-5&keywords=complex+networks

[10] T. I. Papon, T. Chen, and S. Zhang. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proceedings of the ACM on Management of Data* (2024). https://dl.acm.org/doi/abs/10.1145/3654928

[11] A. E. Pratiwi and S. Kundu. 2024. Estimating Diffusion Degree on Graph Stream Generated from Social and Web Networks. In *International Conference on Web Intelligence and Data Engineering*. Springer. https://link.springer.com/chapter/10.1007/978-3-031-62362-2_23

[12] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2013. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Phoenix, AZ, USA, 2014-05). IEEE, 550–559. https://doi.org/10.1109/IPDPS.2014.64

[13] Ryan Torok. 2020. Improving Graph Workload Performance by Rearranging the CSR Memory Layout. (2020).

[14] Duncan J. Watts and Steven H. Strogatz. [n. d.]. Collective dynamics of 'small-world' networks. 393, 6684 ([n. d.]), 440–442. https://doi.org/10.1038/30918

[15] Haiyuan Yu and Mark Gerstein. 2006. Genomic analysis of the hierarchical structure of regulatory networks. *Proceedings of the National Academy of Sciences* 103, 40 (2006), 14724–14731. https://doi.org/10.1073/pnas.0508637103 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.0508637103

[16] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: a high-performance graph DSL. 2 (2018), 1–30. Issue OOPSLA. https://doi.org/10.1145/3276491

[17] F. Ziche, N. Bombieri, and R. Giugno. 2024. GPU-Accelerated BFS for Dynamic Networks. In *European Conference on Parallel Processing*. Springer. https://link.springer.com/chapter/10.1007/978-3-031-69583-4_6