UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer, Communication and Electronic Engineering

Final Dissertation

Optimizing Breadth-First Search on Modern Energy-Efficient Multicore CPUs

Supervisor
Flavio Vella
Co-Supervisor
Thomas Pasquali

Student
Salvatore D. Andaloro

Academic year 2024/2025

Acknowledgments

I would like to thank my co-supervisor and friend Thomas, for his invaluable help in supporting the development of this project and for providing the original motivation for this thesis. My sincere thanks to Prof. Vella for his insightful advice and guidance, and to all the other university professors and staff who have contributed to my education.

To my wonderful family, I offer my heartfelt appreciation for their unwavering support, for providing me with a great education and immense warmth, and to all my friends for their help, counsel, and the joyful times we have shared and will share.

Ringraziamenti

Vorrei ringraziare il mio correlatore e amico Thomas, per il suo prezioso aiuto nel seguire lo sviluppo di questo progetto e per avermi dato lo spunto iniziale di questa tesi. Desidero inoltre ringraziare il Prof. Vella per i suoi consigli e la sua guida, e tutti gli altri professori e il personale universitario che hanno contribuito alla mia formazione.

Desidero ringraziare di cuore la mia splendida famiglia, per l'affetto e la fiducia con cui mi ha sempre sostenuto e per avermi sempre garantito un'istruzione di qualità. Inoltre, vorrei ringraziare i miei amici per il loro sostegno, i loro consigli e i momenti felici che abbiamo condiviso e che condivideremo.

Contents

Abstract

Breadth-First Search (BFS) is a fundamental algorithm for graph analysis, but its performance on modern multicore CPUs is generally limited by irregular memory access. The non-contiguous memory accesses inherent in traversing large-scale graphs lead to poor cache utilization and high-latency memory stalls, creating a significant performance bottleneck for parallel implementations. This thesis investigates and implements two distinct optimization strategies for parallel BFS, targeting modern energy-efficient multicore architectures and focusing on large-diameter graphs, such as road networks.

The first contribution is a cache-optimized implementation in C++ with OpenMP, which proposes a novel MergedCSR data structure. By co-locating vertex metadata with its adjacency list, this format enhances spatial locality to reduce cache misses. The second contribution is an explicitly parallelized implementation in C with pthreads, which provides fine-grained control over the execution model. This version employs a persistent thread pool, a chunk-based frontier with dynamic work-stealing for load balancing, and a lightweight, custom barrier for scalable synchronization.

Both implementations were evaluated against the GAP Benchmark Suite on a diverse set of graphs across x86, RISC-V, and ARM platforms. The results demonstrate that the MergedCSR data structure significantly improves memory performance, enabling a geomean speedup of up to 1.5x over the baseline on large-diameter graphs. Furthermore, the explicit pthreads implementation exhibits superior scalability due to its custom synchronization and outperforms the GAP benchmark with a geomean of 2.28x on road networks and 1.87x on random geometric graphs. This work concludes that achieving optimal performance for memory-bound graph algorithms requires a holistic approach: combining cache-aware data structure design with a fine-grained parallel execution model with low-overhead synchronization.

1 Introduction

Breadth-First Search (BFS) is a fundamental graph traversal algorithm. Its implementations are used in many other algorithms, such as Dijkstra's Algorithm [15], the Ford-Fulkerson Method for Maximum Flow [18] and Prim's Algorithm for Minimum Spanning Tree [38]. Due to its popularity, it was also chosen as the kernel of various benchmarks and competitions; for example, it is the primary kernel of the Graph500 competition, which is a rating of supercomputer systems [32], and it is part of the GAP benchmark suite [6], a standard graph processing benchmark suite.

The BFS exploration algorithm takes as input an undirected graph and a *source* node, exploring all of its neighbors before visiting the nodes at the next depth level. The time complexity of BFS is well-known and is typically expressed as $\mathcal{O}(|V| + |E|)$, where $|V|$ represents the number of vertices in the graph and $|E|$ represents the number of edges in the graph (this notation is used consistently throughout the thesis). This is because, in the worst-case scenario, BFS will visit every vertex and traverse every edge in the graph once [11]. A naive implementation of the Breadth-First Search algorithm using a queue data structure is shown in Algorithm 1.

The principles of graph traversal, exemplified by algorithms like BFS, are central to the field of graph analysis, which has become an essential tool across numerous domains. In social network analysis, for instance, graph algorithms are crucial for understanding community structures [33], identifying influential individuals [28], and tracking the spread of information [46] or misinformation [12]. Bioin-

Algorithm 1: Breadth-First Search (BFS)

Input: A graph $G = (V, E)$, a source vertex $s \in V$
Output: All vertices reachable from s have their distance and parent vertex annotated

```
/* Initialize all vertices */
foreach v ∈ V do
    v.visited ← false;
    v.distance ← ∞;
    v.parent ← NIL;
/* Initialize the source vertex */
s.visited ← true;
s.distance ← 0;
/* Create a queue for BFS */
Let Q be a new queue;
enqueue(Q, s);
/* Repeat until the frontier is empty */
while Q is not empty do
    u ← dequeue(Q);
    /* Remove the vertex from the queue, and check each of its neighbors */
    foreach v in G.Adj[u] do
        /* If the neighbor has not been visited already, mark it as visited, set
           the parent and the distance, and add it to the queue */
        if v.visited is false then
            v.visited ← true;
            v.distance ← u.distance + 1;
            v.parent ← u;
            enqueue(Q, v);
```

formatics also heavily depends on graph analysis to model complex biological systems [48]. In logistics and transportation, graphs are used to model road networks [41], flight paths [35], and supply chains [45]. Furthermore, the rise of machine learning has opened up new frontiers for graph analysis [9]. Graph-based machine learning models are now used for a variety of tasks, including recommendation systems that suggest products or connections [20], fraud detection by identifying unusual patterns in transaction networks [37], and in the field of natural language processing to understand the relationships between words and concepts [47]. For graph analysis to be truly effective across diverse fields, it must operate on extremely large datasets comprising millions or even billions of nodes and edges. For example, the road network graph of Europe, a common graph used in benchmarks, contains 51 million vertices and 108 million edges[1]. This graph alone occupies nearly one gigabyte in its uncompressed form. Such data volumes make processing computationally and memory intensive. To efficiently handle these massive datasets, it is crucial to employ modern computational approaches like parallel computing.

Parallel computing is a computation method where multiple processors or computers work together to solve a common problem. The main goal of parallel computing is to increase performance by breaking down large tasks into smaller subtasks that can be executed simultaneously. A prominent approach in parallel computing is the use of multicore processors. A multicore processor is a single chip that contains two or more independent processing units, called cores. Each core can read and execute program instructions independently, allowing for the parallel processing of multiple tasks.

Two primary methods exist for programming on multicore processors: manual parallelization and framework-based parallelization. Manual parallelization requires the programmer to manage threads directly using low-level libraries, such as pthreads. This includes the explicit creation of threads and the usage of synchronization primitives to ensure correct concurrent execution. Alternatively, a parallelization framework such as OpenMP [13] allows the programmer to insert directives into the code. These directives identify sections that can be executed in parallel, offloading the tasks of thread creation, management, and synchronization to the compiler. This model greatly simplifies the parallelization of certain structures, such as for loops, but the introduced level of abstraction can add some performance overhead when an algorithm's structure does not align well with the framework's parallelization model. In such cases, an explicitly-parallelized implementation may be necessary to achieve superior performance. The parallelization of the Breadth-First Search algorithm, detailed in Section 3.2, serves as a case study for this limitation.

Another important aspect to consider when optimizing a program is the memory access pattern. Modern computer architectures feature a hierarchical memory model designed to bridge the speed gap between the fast processor registers and the slower main memory (RAM). This system mitigates the latency difference between processor registers and RAM by using multiple levels of caches, such as L1, L2, and L3. Caches are smaller and faster than RAM and are physically closer to the processor cores. The effectiveness of caching relies on the principles of temporal and spatial locality. Temporal locality is the observation that a program is likely to access a specific data location again shortly after its initial access. Caches exploit this by storing recently used data, so subsequent requests are served from the faster cache rather than the slower main memory. Spatial locality is the principle that a program is likely to access memory locations near a recently accessed address. To leverage this, data is fetched from RAM into the cache in contiguous blocks called cache lines, making subsequent accesses to adjacent data faster.

Breadth-First Search processes each vertex a single time, which unfortunately nullifies performance benefits from temporal locality as data is generally not reused often. Significant performance improvements may be achieved by exploiting spatial locality instead. One way of accomplishing this is by structuring the program to load a vertex and its neighbors into memory simultaneously. A data structure that exploits this approach to improve cache utilization is presented in Section 3.1.

In Chapter 4, a comprehensive analysis is conducted to evaluate the proposed implementations on a diverse range of datasets and hardware. This includes an x86 AMD server-grade processor, the emerging open-source RISC-V architecture, and the specialized ARM-based NVIDIA Grace Hopper superchip.

[1] `https://sparse.tamu.edu/DIMACS10/europe_osm`

The RISC-V architecture represents a significant shift in processor design. Originating from research at the University of California, Berkeley, RISC-V is an open-source instruction set architecture (ISA) [4]. Unlike proprietary ISAs from companies like Intel and AMD, RISC-V is freely available for anyone to use, modify, and build upon without paying licensing fees. This open nature has made it a compelling platform for both academic research and commercial development [31]. Given its unique characteristics and growing importance, it therefore provides a valuable point of comparison for software optimization.

The NVIDIA Grace Hopper superchip is specifically designed for high-performance computing (HPC) and artificial intelligence (AI). This platform integrates a high-core-count, ARM-based Grace CPU with a powerful Hopper GPU on a single module [16]. Given its unique architectural approach and its deployment in leading supercomputing systems, the Grace Hopper platform offers an interesting point of comparison for evaluating the performance of memory-bound algorithms on next-generation, ARM-based HPC hardware.

2 Background

Despite the algorithmic simplicity of BFS, achieving high performance execution is a significant challenge on modern multi-core architectures. The primary performance bottleneck is not the algorithm's asymptotical computational complexity, which is linear in the number of vertices and edges, but rather its memory access pattern. The irregular and unpredictable structure of most real-world graphs results in non-contiguous memory accesses, which leads to poor cache utilization and high-latency stalls as the processor waits for data to be fetched from main memory [27, 29]. Consequently, state-of-the-art research has largely focused on two areas: refining the traversal strategy itself [3, 5, 23] and redesigning the underlying graph data structures to be more amenable to the memory hierarchies of modern hardware [43, 22].

2.1 Direction-Optimizing BFS

A pivotal advancement in traversal strategy was the introduction of direction-optimizing or Hybrid BFS, proposed by Beamer et al. in 2012 [5]. In a Breadth-First Search, the set of vertices at the current depth of the search is known as the frontier. In this strategy, each frontier expansion is approached either using the Top-Down approach or the Bottom-Up approach. The Top-Down approach, which was used in the other BFS algorithms prior to this work, involves each vertex in the current frontier exploring its neighbors to identify unvisited vertices, which then form the frontier for the next level. The Bottom-Up approach instead inverts the search. It iterates through all vertices and, for those that have not been visited, it checks if they have a parent in the current frontier. A visual representation of both strategies is shown in Fig. 2.1.

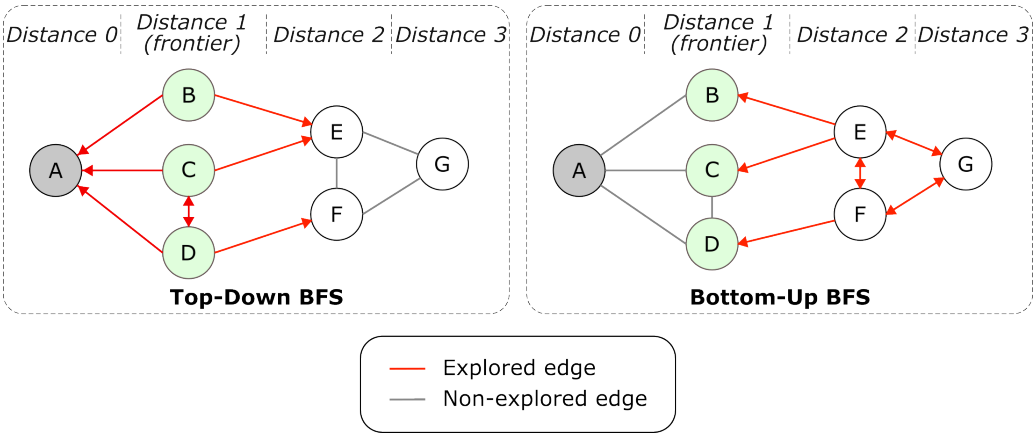


Figure 2.1: Comparison of Top-Down and Bottom-Up Breadth-First Search strategies, starting from source vertex A, for discovering vertices at distance 2. The frontier is the set of vertices at distance 1 $\{B, C, D\}$. The Top-Down approach inspects the neighbors of each vertex in the frontier. This process requires 8 edge traversals to find the next level. However, this strategy performs some superfluous work: it revisits the already visited vertex A, it redundantly traverses the edge C-E even though E was already discovered through B-E, and it unnecessarily traverses edge C-D and D-C. In contrast, the Bottom-Up approach inspects all unexplored vertices $\{E, F, G\}$, and checks whether any of their neighbors belong to the current frontier $\{B, C, D\}$. This method requires 9 edge traversals, since edges between unexplored vertices (such as E-F, E-G, and F-G) are traversed in both directions.

2.2 Graph diameter and frontier expansions

The effectiveness of the Top-Down and Bottom-Up strategies is intrinsically linked to a graph's structural properties, most notably its diameter. The diameter dictates the shape and size of the BFS frontier at each level of the traversal, which is the primary factor in determining the most efficient approach [5, 2, 3]. This relationship gives rise to two broad classes of graphs: small-diameter and large-diameter.

Small-Diameter Graphs, such as the social networks shown in the plot, are characterized by the "small-world" phenomenon [1]. The frontier size exhibits a distinct pattern: it quickly grows to encompass a significant fraction of the total vertices, and then rapidly collapses. For these graphs, a hybrid traversal is essential. The Top-Down strategy is efficient for the initial and final levels, but during the intermediate phase where the frontier is massive, the Bottom-Up strategy is superior, because it is more efficient to have the small set of unvisited nodes "find" the enormous frontier than the other way around.

Large-Diameter Graphs, such as road networks, Finite Element Models, and Random Geometric Graphs, lack vertices with high degree that create short paths. Consequently, the BFS frontier progresses in a slow, wave-like manner, never "exploding" in size. As shown in the figure, the frontier size for the road networks, FEM and RGG graphs remains relatively small throughout the entire, much longer traversal (the y-axis is logarithmic). For these graphs, the Top-Down approach is almost always the more efficient strategy, as the number of outgoing edges from the frontier is never large enough to justify the high overhead of a Bottom-Up search, which would require iterating over the vast set of unvisited vertices at each step.

This dichotomy in frontier behavior is the core motivation for the direction-optimizing algorithm. The heuristics used to switch between the two modes try to dynamically classify the state of the traversal at each level, applying the Bottom-Up optimization only when the frontier grows large enough to resemble the intermediate state of a small-diameter graph traversal.

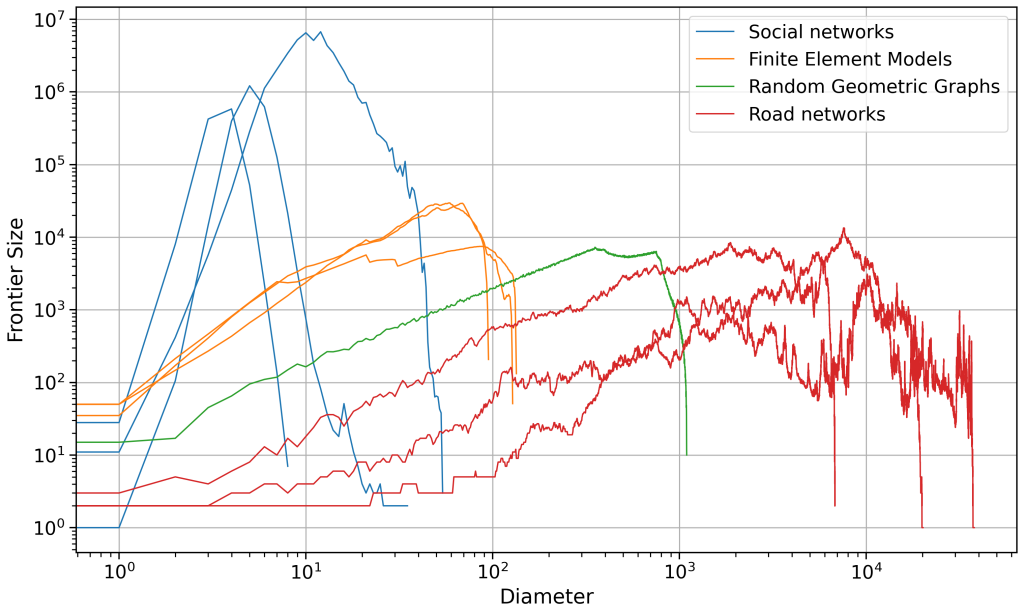


Figure 2.2: The evolution of BFS frontier size for different graphs, colored by their graph class. The plot illustrate the frontier size at each step. Social Networks exhibit an explosive growth and subsequent collapse of the frontier, peaking at over a million vertices in fewer than 100 steps. Road Networks show a very long traversal where the frontier size remains relatively small and stable. Random Geometric Graphs display a steady expansion, confirming its large-diameter nature where the frontier never becomes excessively large. Finite Element Models exhibit intermediate behaviour, they show a more pronounced frontier growth than the road networks, representing a case where a hybrid approach might occasionally activate a Bottom-Up step.

2.3 The Compressed Sparse Row (CSR) format

The performance of the Top-Down and Bottom-Up traversal strategies is also dependent on the underlying data structure used to represent the graph. A widely adopted and memory-efficient format for representing sparse graphs is the Compressed Sparse Row (CSR) format. The graph's structure is encoded using two primary arrays, called `col_idx` and `row_ptr`. The `col_idx` array contains the concatenated adjacency lists of all vertices. The `row_ptr` array contains the offset of each adjacency list, where the entry at index i points to the start of the i-th vertex's adjacency list within the `col_idx` array. The neighbors of a vertex i are therefore located in the segment of the adjacency array delimited by the offsets at index i and $i+1$. An example of the CSR format is shown in the top-right of Fig. 2.3. The space complexity of CSR is $\mathcal{O}(|V| + |E|)$, where $|V|$ indicates the number of vertices and $|E|$ indicates the number of edges.

Despite the algorithmic efficiency gained by this hybrid approach, the standard CSR data structure can lead to significant cache inefficiencies. During a traversal, processing a single vertex may require scattered memory accesses to separate arrays: the `row_ptr` for the CSR pointers, the `col_idx` for the neighbor lists, and algorithm-specific data like parent IDs or distances [43, 2]. To enhance spatial locality, the MergedCSR format was proposed by Torok [43]. This format redesigns the memory layout by merging a vertex's adjacency list into a single, contiguous data structure. By co-locating all data required to process a vertex, this layout minimizes cache misses and reduces pressure on the memory subsystem. An example of the Merged CSR layout is shown in Fig. 2.3. The space complexity of MergedCSR is equal to the standard CSR, i.e. $\mathcal{O}(V + E)$. An improvement of the original Merged CSR format is discussed in Section 3.1.

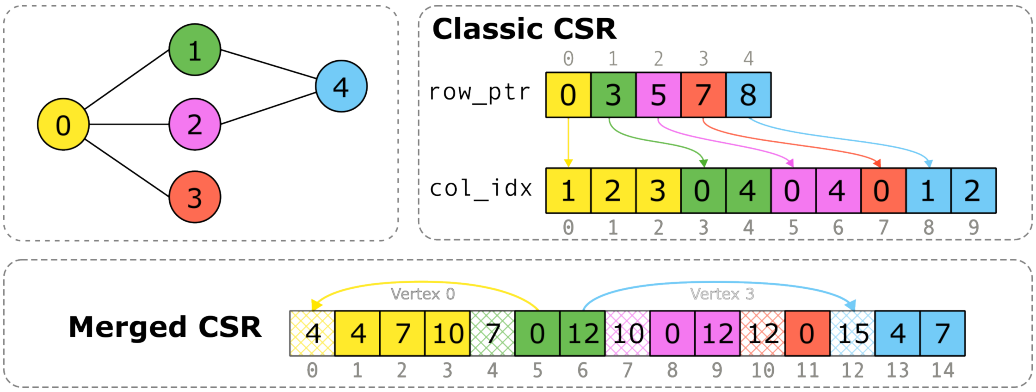


Figure 2.3: An example of the standard CSR format (top) and a cache-optimized Merged CSR layout (bottom) for the depicted graph. In the Merged CSR format, the hatched cells contain the index of the next vertex's adjacency list. Other cells contain the index of the neighboring vertex inside the Merged CSR. Arrows show the semantics of vertex 1 adjacency list, which has as neighbors vertex 0 (at location 0 in teh MergedCSR array) and vertex 4 (at location 12 in the MergedCSR array).

While the CSR and MergedCSR formats provide fast retrieval of a vertex's neighbors, which is ideal for the Top-Down step, the Bottom-Up step requires also another lookup: a rapid method to check if a vertex is an element of the frontier. This check is performed by inspecting the distances array. To facilitate this lookup, some implementations employ a bitmap to store the current frontier to improve cache utilization [5, 2, 3, 34]. This is because a bitmap occupies much less space than the distances array: each cell in the distance array requires 32 bits compared to only one bit in the bitmap, increasing the chances of a cache hit.

2.4 Parallelization Strategies

To leverage the computational power of multicore processors, the workload of exploring each level of the graph must be distributed across multiple processing cores for concurrent execution. Due to the different structure of the Top-Down and Bottom-Up steps, the two exploration strategies are parallelized differently.

For the Top-Down step, parallelization is achieved by partitioning the current frontier among the

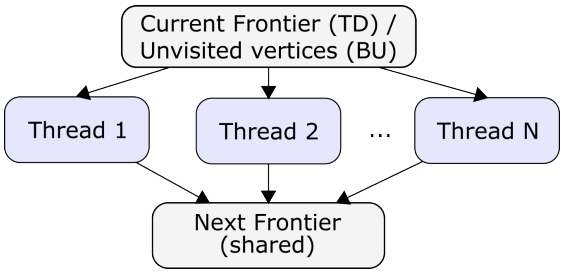
available threads. Each thread is assigned a subset of frontier vertices and is responsible for exploring the adjacency lists of its assigned vertices. The primary challenge in this phase is managing concurrent writes to the next frontier.

In the Bottom-Up step, the source of parallelism is the complete set of vertices, which is partitioned among the threads. Each thread iterates through its the vertices in the assigned subset and checks if they are still unvisited by performing a lookup in the bitmap. If so, it checks if any of them is part of the current frontier by performing a lookup in another bitmap.

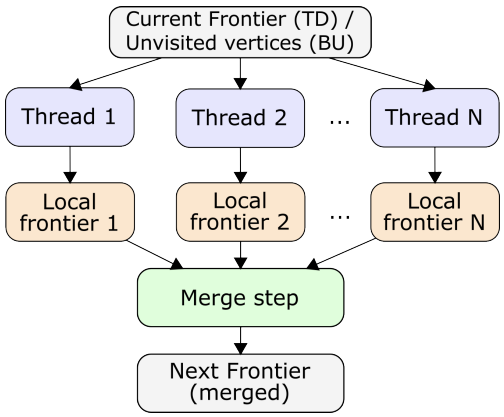
In both implementations, when multiple threads discover the same unvisited vertex simultaneously, a race condition occurs. To prevent a vertex from being added multiple times, one solution is to use atomic compare-and-swap operations to ensure only one thread successfully marks the vertex and adds it to the frontier [5]. This approach is shown in Fig. 2.4a. Alternatively, Leiserson and Schardl observed that adding a vertex multiple times to the frontier is a benign race condition that does not affect correctness, but only performance [26]. Their implementation therefore forgoes synchronization, accepting minor redundant work in exchange for lower overhead.

The presence of an inherently sequential data structure such as the next frontier has led to the development of alternative strategies that trade off synchronization overhead with other costs. One common solution to eliminate write contention is having each thread populate its own thread-local frontier queue [5, 26]. This avoids the need for atomic operations during the parallel exploration phase but introduces a subsequent merge step at the end of each level, where all thread-local queues must be consolidated into a single frontier for the next iteration. The overhead of this merge phase scales with the number of threads and the number of vertices in the frontier. This approach is shown in Fig. 2.4b.

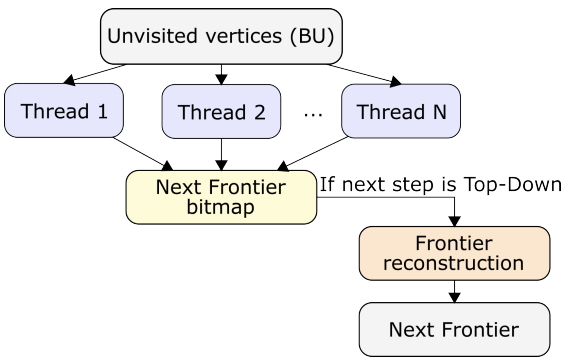
A further optimization for the Bottom-Up steps avoids creating an explicit frontier queue by having threads update only the next frontier's bitmap [2]. The drawback of this 'bitmap-only' approach emerges when the algorithm switches from the Top-Down to the Bottom-Up approach or vice versa. This is because the Top-Down step requires an explicit list of frontier vertices to iterate over, necessitating a full scan of the vertex set to reconstruct the frontier from the bitmap. This operation has a cost proportional to the total number of vertices. This approach is shown in Fig. 2.4c.



(a) Shared Frontier with Atomic Operations.



(b) Thread-Local Frontiers with a Final Merge Step.



(c) Bitmap-based Approach for Bottom-Up Steps.

Figure 2.4: Proposed approaches for handling parallelization.

2.5 Performance Optimizations for Parallel BFS

This section presents an overview of selected techniques that have been proposed for optimizing the parallel BFS algorithm.

Implementations targeting NUMA systems often employ optimizations which can be relevant also for multicore CPUs. In a NUMA system, each processor (or socket) possesses its own local memory, which offers significantly lower access latency and higher bandwidth compared to accessing memory attached to other processors via high-speed interconnects. Although the algorithms presented in Chapter 3 focus on multicore CPUs, previous research has shown that also multicore programs benefit from NUMA-like optimizations [7]. For example, the chiplet architecture of modern AMD EPYC processors and the segmented L3 cache of Intel Cascade Lake results in non-uniform L3 access latency, which varies based on the physical proximity of the accessing core to the cache slice containing the data [44].

The Polymer system [49], which was one of the first NUMA-aware graph processing systems, co-locates vertices and connected edges within the same NUMA node as much as possible and keeps only a lightweight copy of other vertices' data, such as degree and the start index of neighboring edges. Moreover, they use a Sense-Reversal Centralized Barrier [30], which uses atomic fetch-and-add instructions to reduce contention on the software barrier between each BFS frontier expansion. As for the frontier, they use a variation of thread-local frontiers with a final merge step (as shown in Fig. 2.4b).

The optimized BFS implementation by Tithi et al. [42], instead focuses on avoiding locks and atomic instructions in shared-memory parallel BFS through optimistic parallelization. This method allows potentially conflicting operations to run in parallel, knowing that conflicts will be rare and manageable without compromising correctness. They implemented the BFS algorithm using both centralized job queues and distributed randomized work-stealing, demonstrating that lock-free versions generally outperform their lock-based counterparts. The authors also discussed strategies for optimizing their algorithms for NUMA machines, such as co-locating threads with their assigned queues on the same socket or prioritizing same-socket work-stealing targets.

Booth and Lane [14], introduced iCh, a loop scheduling method for OpenMP for irregular parallel applications on shared-memory multicore systems. Its key innovations include adaptive self-scheduling using distributed queues per thread and adaptive chunk size tuning that adjusts based on a thread's estimated iteration throughput. It also incorporates an efficient work-stealing mechanism to balance loads. These core ideas, particularly local queues and work-stealing, are directly applicable and beneficial for Breadth-First Search (BFS) implementations, which are also inherently irregular.

The ideas of Sense-Reversal Centralized Barriers, thread-local queues, and work stealing have been implemented in a single algorithm, described in Section 3.2.

2.6 Parallelization Frameworks

The majority of high-performance BFS implementations for multicore CPUs utilize implicit parallelization through frameworks such as OpenMP [13] or Cilk++ [25]. Examples include the reference implementation for the Graph500 benchmark up to version 2.1.4 [32], the direction-optimizing algorithm by Beamer [5], the Ligra framework [40], and the implementations by Tithi et al. [42] and Gonzaga et al. [21]. In contrast, more comprehensive graph processing systems such as Polymer [49] and X-Stream [39] are implemented using the pthreads library[1], which provides explicit thread management and synchronization. This approach is necessitated by the requirements of these systems, which support numerous graph kernels beyond BFS and often handle dynamic graph updates. Such features demand fine-grained control over memory layout, thread scheduling, and data structure synchronization, a level of control not directly offered by higher-level parallelization frameworks.

Both graph processing systems are more than 10 years old and do not include the direction-optimizing approach. It is therefore of interest to examine the performance of standalone BFS when implemented using explicit parallelization through the pthreads execution model. One such implementation, which includes also the optimizations described in Section 2.5, is discussed in Section 3.2.

[1] `https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html`

3 Methodology

This chapter presents two parallel BFS implementations. Section 3.1 describes a cache-optimized implementation using the MergedCSR data structure. The code is implemented in C++ using the OpenMP parallelization framework. Section 3.2 describes an explicitly parallelized BFS implementation in C using the pthreads parallel execution model.

3.1 Cache-optimized BFS using the MergedCSR data structure

The performance of a Top-Down BFS traversal on a standard Compressed Sparse Row (CSR) representation is often constrained by memory latency. For each vertex explored, the algorithm must perform scattered memory accesses to three distinct data structures: the `row_ptr` array to find the offset of the adjacency list, the `col_idx` array to retrieve the neighbors, and a separate distances array to check the visited status and store the new distance. These disjoint accesses frequently lead to cache misses, as the data required to process a single vertex is not co-located in memory.

3.1.1 Design and Memory Layout of MergedCSR

To mitigate scattered memory accesses, the MergedCSR format merges the `row_ptr` and `col_idx` arrays into a single contiguous structure. The format proposed in this work extends this data structure by integrating algorithm-specific metadata directly into the graph's memory layout. In this new format, the adjacency list for each vertex is prepended with two metadata fields: the vertex's degree and its distance from the source vertex. The entire graph is thus stored in a single, large array where each vertex's data block is structured as: `[degree, distance, neighbor_1_idx, neighbor_2_idx, ...]`. Since each vertex's data block in MergedCSR is expanded by two elements (for degree and distance), the offsets for each subsequent vertex will be larger. A new `row_ptr` array is therefore necessary to correctly index into the start of each vertex's block within the new, larger structure. The space complexity for the MergedCSR array is thus $\mathcal{O}(2|V| + |E|)$ elements. An example of the MergedCSR structure is shown in Fig. 3.1, the pseudocode for converting from a standard CSR representation to MergedCSR is outlined in algorithm 2, and the pseudocode to extract the distances from the MergedCSR is outlined in algorithm 3.

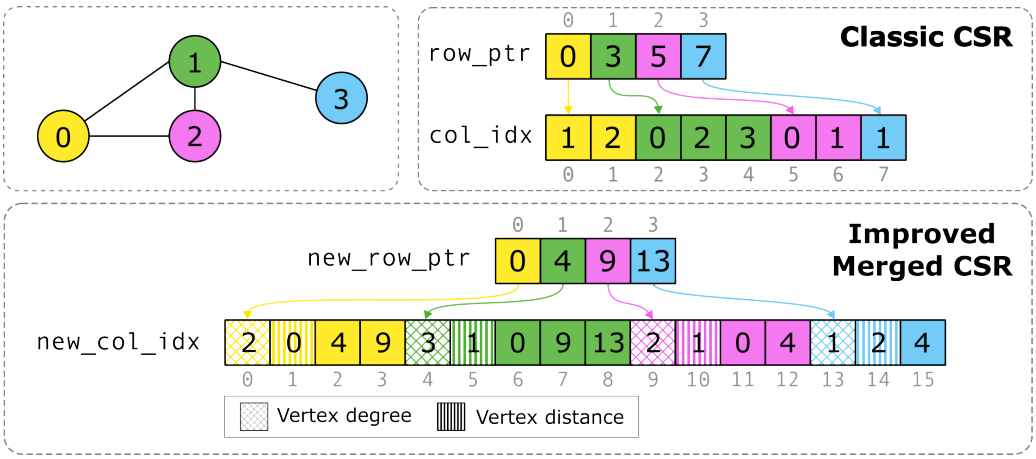


Figure 3.1: An example of the improved MergedCSR structure.

Algorithm 2: Conversion to a Merged CSR Representation

Input: *row_ptr*: An array of row pointers for a standard CSR graph.
col_idx: An array of column indices for a standard CSR graph.
num_vertices: The total number of vertices in the graph.
Output: *new_row_ptr*: The new row pointer array for the MergedCSR structure.
merged_csr: The new data structure containing degrees, distances, and neighbors.

```
merged_csr ← new int[2 × num_vertices + |col_idx|];
new_row_ptr ← new int[num_vertices + 1];
cursor ← 0;
for i ← 0 to num_vertices − 1 do
    new_row_ptr[i] ← cursor ;   // Update position of vertex i in the new structure
    degree ← row_ptr[i + 1] − row_ptr[i];
    merged_csr[cursor] ← degree ;                                  // Initialize degree
    merged_csr[cursor + 1] ← ∞ ;                                  // Initialize distance
    cursor ← cursor + 2 ;                // Offset cursor by number of metadata cells
    // Copy neighbors of vertex i
    for j ← row_ptr[i] to row_ptr[i + 1] − 1 do
        /* Retrieve the ID of this neighbor from the original col_idx array */
        neighbor_id = col_idx[j];
        /* Calculate the starting index of this neighbor's data block.  Account
           for its original offset (row_ptr[neighbor_id]) plus the two metadata
           slots added for every vertex that comes before it (2 * neighbor_id)
           */
        new_position ← row_ptr[neighbor_id] + 2 ∗ neighbor_id;
        /* Store the calculated pointer in the new structure */
        merged_csr[cursor] ← new_position;
        cursor ← cursor + 1 ;  // Advance the cursor to the next available position
new_row_ptr[num_vertices] ← cursor;
return new_row_ptr, merged_csr;
```

Algorithm 3: Distance Extraction from MergedCSR

Input: *merged_row_ptr*: The row pointer array for the MergedCSR structure.
merged_csr: The MergedCSR data array containing metadata and neighbors.
num_vertices: The total number of vertices.
Output: *distances*: An array populated with the distance of each vertex from the source.

```
for i ← 0 to num_vertices − 1 do
    offset ← merged_row_ptr[i] ;        // Get start index for vertex i's data block
    distances[i] ← merged_csr[offset + 1] ;      // Copy final distance from metadata
    merged_csr[offset + 1] ← ∞ ;        // Reset distance in MergedCSR for next run
return distances;
```

This layout is explicitly designed to exploit spatial locality. When the BFS algorithm accesses a vertex to check or update its distance, the CPU hardware will load the corresponding cache line. Because the vertex's degree and its first few neighbors are now stored immediately after the distance field, it is highly probable that this single memory fetch will also load the beginning of the vertex's adjacency list into the cache. This effectively prefetches the data required for the next frontier expansion, thereby reducing or eliminating subsequent cache misses that would have been necessary with the standard CSR format.

3.1.2 Traversal Algorithm and Implementation Details

The BFS algorithm implemented using the MergedCSR data structure is level-synchronous. This means that all vertices at a given distance d from the source are fully explored before the algorithm

proceeds to discover vertices at distance $d + 1$.

Due to the structure of the MergedCSR, this design is optimized exclusively for the Top-Down traversal step and precludes an efficient Bottom-Up implementation. The core operation of the Bottom-Up approach involves iterating over all unvisited vertices and checking if any of their neighbors reside in the current frontier. This check is performed efficiently using a bitmap of the frontier, which requires a direct mapping from a vertex ID to its corresponding bit in the bitmap. The MergedCSR format changes the indices in the adjacency lists and breaks the index-to-vertex-ID mapping. While one could search for each neighbor's original ID to query the bitmap, the computational overhead of this $\mathcal{O}(|V|)$ lookup would negate any performance benefits of the Bottom-Up strategy. Therefore, to maximize the benefit of the improved data locality, only the Top-Down exploration strategy is employed.

Parallelism is introduced at each level of the traversal by partitioning the vertices of the current frontier among a team of OpenMP threads. The implementation follows the "Thread-Local Frontiers" with a "Final Merge Step" parallelization strategy (see Section 2.4 and Fig. 2.4b). This strategy is realized efficiently through a custom OpenMP reduction. Instead of having all threads contend for locks or use atomic operations to write to a single shared next-frontier, each thread populates its own private, thread-local version of the next frontier. At the end of the parallel exploration for that level, OpenMP automatically performs a final merge step to combine all the thread-local frontiers into a single, unified frontier for the next iteration.

```
#pragma omp declare reduction(vec_add \
    : frontier : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

This custom reduction instructs OpenMP on how to combine the thread-local frontiers at the end of the parallel loop: the contents of each thread's private frontier (`omp_in`) are appended to the master frontier (`omp_out`). This direct vector concatenation avoids the need for manual synchronization. The custom reduction is then applied to the for loop that iterates over the vertices in the frontier, as shown the the following listing.

```
#pragma omp parallel for reduction(vec_add : next_frontier) schedule(static) if
↪  (this_frontier.size() > 50)
for (const auto &v : this_frontier) {
  eidType end = v + 2 + DEGREE(v);
  // Iterate over neighbors
  for (eidType i = v + 2; i < end; i++) {
      eidType neighbor = merged_csr[i];
      // If neighbor is not visited, add to frontier
      if (DISTANCE(neighbor) == std::numeric_limits<weight_type>::max()) {
        if (DEGREE(neighbor) != 1) {
          next_frontier.push_back(neighbor);
        }
        // Set the neighbor's distance
        DISTANCE(neighbor) = distance;
      }
  }
}
```

The outer loop is parallelized using the `schedule(static)` clause, which pre-allocates fixed-size chunks of the frontier to each thread. While graph traversal workloads are inherently irregular and can lead to load imbalance, empirical evaluation demonstrated that the low overhead of static scheduling consistently outperformed the more complex dynamic scheduling strategies offered by OpenMP.

Furthermore, parallelization is enabled only when the frontier size exceeds a threshold of 50 vertices. This optimization prevents the runtime overhead associated with creating and managing the thread team from outweighing the computational speedup on small workloads, for which sequential execution is faster.

3.1.3 Postprocessing and Generality

Once the BFS traversal is complete, the distance values are embedded within the MergedCSR structure. A final postprocessing step is required to extract these values into a standard distances array. This is accomplished with a simple parallel loop that iterates from 0 to $|V| - 1$, using the updated `row_ptr` to locate the distance field for each vertex and copy its value. The pseudocode of this procedure is outlined in algorithm 3.

The same framework can be adapted to produce a parent array instead of a distance array. In this variation, the distance field in the MergedCSR structure is repurposed to store the parent ID of each vertex. The logic of the traversal remains largely the same, with the algorithm storing the ID of the discovering vertex instead of the new distance level.

3.2 Case study: Explicit parallelization of BFS using pthreads

This section presents an alternative BFS implementation that avoids high-level parallelization frameworks and instead uses explicit thread management with the POSIX Threads (pthreads) library in C. This approach provides fine-grained control over thread lifecycle, work distribution, and synchronization. The design is built upon three core components: a persistent thread pool to minimize thread creation overhead, a chunk-based frontier with a work-stealing mechanism to dynamically balance the load, and a low-overhead, centralized barrier to ensure level-synchronous traversal. For its underlying graph representation, this implementation utilizes the cache-optimized Merged CSR data structure.

3.2.1 The Chunk-Based Frontier

The parallel frontier is managed by a set of data structures designed to minimize lock contention, reduce dynamic memory allocation overhead, and facilitate a dynamic work-stealing load balancing strategy. The design is composed of three primary structures: the `Chunk`, the `ThreadChunks` queue, and the top-level `Frontier` container. A schematic representation of the combined structures is shown in Fig. 3.2.

The `Chunk` structure This structure represents the atomic unit of work distributed among threads. The structure consists of a fixed-size block containing an array of vertex identifiers. The `CHUNK_SIZE` constant defines the capacity of each block. This design choice amortizes the overhead associated with managing the frontier; instead of threads manipulating individual vertices in a shared queue, they acquire and process entire blocks of work at a time. The `next_free_index` field serves as a stack pointer, allowing the `Chunk` object to function as a Last-In, First-Out (LIFO) buffer. Vertices are pushed by placing them at this index and incrementing it, and popped by decrementing the index. When `next_free_index` is zero, the chunk is empty. This LIFO access pattern is cache efficient, as vertices that were just written to the chunk are also the first ones to be read and processed.

```
typedef struct {
  ver_t vertices[CHUNK_SIZE];
  int next_free_index;
} Chunk;
```

The `ThreadChunks` structure This structure manages a dynamic array of pointers to a set of `Chunk` structures. The `top_chunk` index makes the array function as a stack, while the `chunks_size` field tracks the total allocated capacity of this array. If a thread's queue runs out of space (i.e., `top_chunk` reaches `chunks_size`), the chunks array is reallocated to double its previous capacity. This expansion strategy, which is implemented using the `realloc()` function, ensures that the cost of adding new chunks has amortized cost $\mathcal{O}(1)$. Similarly to the `Chunk` structure, this structure also acts as a LIFO stack to exploit temporal cache locality. During the frontier expansion phase of a BFS level, a thread creates and populates new chunks with newly discovered vertices. In the subsequent BFS level, the thread will pop the chunks to process them. Because the top chunks were the last to be inserted, they also have a higher probability of still residing in the thread's private L1 or L2 caches, therefore optimizing the cache usage. Each `ThreadChunks` structure contains also its own `pthread_mutex_t` lock. A thread must acquire this lock to add or remove a `Chunk` from its own queue. This same lock must be acquired also by any other thread attempting to "steal" a Chunk from this queue.

```
typedef struct {
  Chunk **chunks;
  int chunks_size;
  int top_chunk;
  pthread_mutex_t lock;
} ThreadChunks;
```

The `Frontier` structure This structure consists of an array of pointers to `ThreadChunks` objects, providing a centralized point of access to each thread's individual work queue. This allows a work-stealing thread to iterate through all other threads and attempt to acquire work from their respective queues. The `Frontier` structure contains also a `thread_chunk_counts` array, which stores the number of active chunks per thread. This array provides a fast, lock-free snapshot of the entire system's work distribution. An idle thread can scan this array to quickly identify the busy threads and steal work from them, without incurring the high contention cost of acquiring each thread's mutex simply to inspect its queue size.

```
typedef struct {
  ThreadChunks **thread_chunks;
  int *thread_chunk_counts;
} Frontier;
```

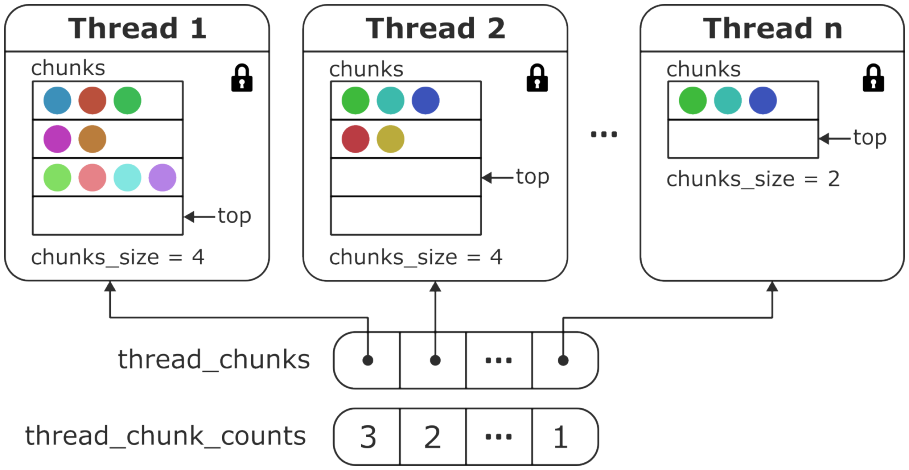


Figure 3.2: An overview of the `Frontier` data structure for parallel BFS. The top-level `Frontier` structure contains pointers to each thread's private `ThreadChunks` work queue. Each `ThreadChunks` contains a stack of `Chunks`, with the top pointer indicating the next available slot in the stack. The colored circles represent the vertices contained in each chunk. Access to the `chunks` array is serialized by a dedicated mutex (represented by the lock icon). The auxiliary `thread_chunk_counts` array provides lock-free access to the number of chunks of each thread.

3.2.2 Dynamic Load Balancing via Work-Stealing

To counteract the inherent workload imbalance characteristic of irregular graph traversal, a dynamic work-stealing mechanism is implemented. This protocol allows idle threads to acquire work from busy threads.

A thread transitions from a "worker" to a "thief" state as soon as it has processed all the chunks in its own `ThreadChunks` queue. Once its own work queue is empty, it initiates the work-stealing protocol. First, the stealing thread iterates through the `thread_chunk_counts` array, looking for any thread whose chunk count is greater than one. This heuristic prevents excessive lock contention, particularly during phases of the BFS where the frontier is small. If the entire frontier consists of only a single chunk, this rule prevents all but one thread from repeatedly attempting to lock the same mutex, a phenomenon which would serialize execution and negate the benefits of parallelism.

Upon identifying a potential victim that satisfies this condition, the thief proceeds to attempt a steal. It first tries to acquire the victim thread's `pthread_mutex_t` lock. Once the lock is successfully acquired, the thief performs a second check on the victim's chunk count. This check is necessary to prevent a race condition; it is possible that another thief stole a chunk in the time between the initial lock-free check and the successful acquisition of the lock. If the victim thread still has more than one chunk available, the steal is executed. The thief decrements the victim's `top_chunk` index to claim a chunk, and the corresponding value in the shared `thread_chunk_counts` array to ensure the global

view of work distribution remains consistent. After these modifications, the thief releases the victim's lock and begins processing the vertices within the stolen chunk.

Upon finishing the stolen chunk, the thief thread re-initiates the work-stealing protocol, starting a new scan of the `thread_chunk_counts` array to find another victim. This loop continues until the thread performs a full scan of the `thread_chunk_counts` array and finds that all entries are zero. This state signifies that the entire frontier for the current level has been processed. At this point, the thread stop its execution for the current level and proceed to the synchronization barrier. The flowchart of the work-stealing protocol is shown in Fig. 3.3.

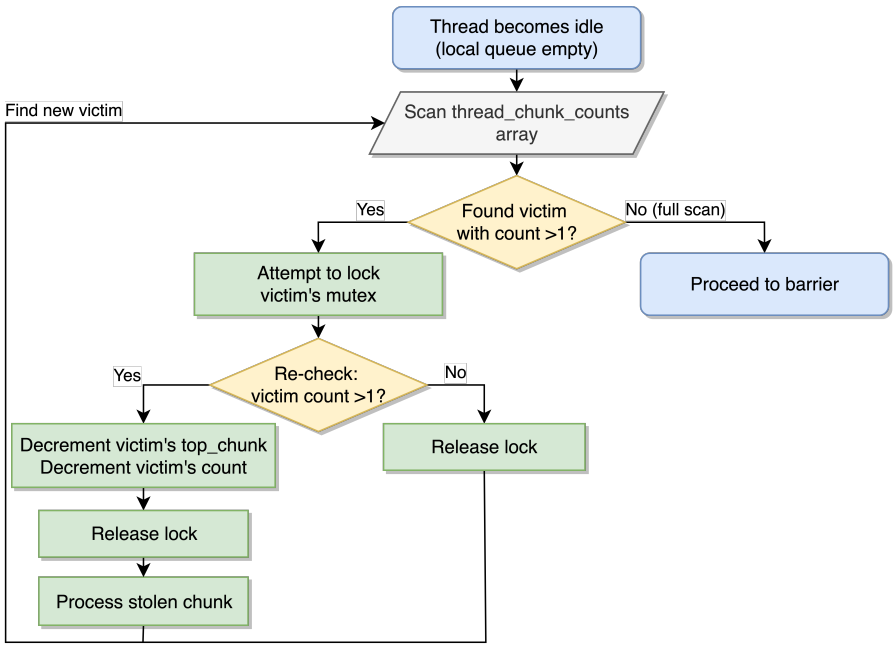


Figure 3.3: The work-stealing protocol.

3.2.3 Thread Pool and Work Dispatch

The explicit parallelization of the BFS algorithm is built upon a persistent thread pool. This design is motivated by the high performance cost associated with the creation and destruction of operating system threads. By creating a fixed-size pool of worker threads once at program initialization and reusing them for each subsequent BFS traversal, the significant overhead of thread management is amortized all BFS runs. The `thread_pool_t` structure serves as the central control block for the entire pool, managing thread handles, state, and the synchronization primitives required for coordination between the main (parent) thread and the worker (child) threads.

```
typedef struct {
  pthread_cond_t cond_children;    // Used to signal child threads
  pthread_mutex_t mutex_children;
  pthread_cond_t cond_parent;      // Used to signal the parent thread
  pthread_mutex_t mutex_parent;

  pthread_t threads[MAX_THREADS];  // Set of spawned threads
  int thread_ids[MAX_THREADS];     // Set of spawned threads' IDs
  atomic_uint run_id;              // Counter for work cycles
  atomic_bool stop_threads;        // Flag to signal threads to terminate
  atomic_bool children_done;       // Flag to signal parent that workers are done

  void *(*routine)(void *);        // Stores the worker function pointer
} thread_pool_t;
```

Synchronization Primitives The implementation employs two distinct pairs of mutexes and condition variables to manage the two-way communication between the main thread and the worker threads. The `cond_children` condition variable and its associated `mutex_children` are used exclusively to manage the state of the worker threads. The main thread signals `cond_children` to wake the workers and dispatch a new work cycle. Symmetrically, the `cond_parent` condition variable and `mutex_parent` are used by the main thread. This allows it to enter an efficient wait state, blocking on `cond_parent` until a worker signals that the entire BFS task is complete. This separation makes the synchronization logic simple, preventing potential deadlocks or race conditions that could arise from using a single set of primitives for bidirectional communication.

State and Signaling Variables The coordination between threads is orchestrated through a set of atomic state variables that ensure thread-safe communication without requiring locks for every state change. The primary signaling mechanism is the atomic `run_id` integer, which functions as a work cycle counter. Using a counter instead of a simple boolean flag provides a more robust design, as each work dispatch is identified by a unique ID. This prevents workers from accidentally performing a work cycle multiple times in the case of a spurious wakeup, as they explicitly check their local counter against the global one. The `stop_threads` atomic boolean variable is used to signal a graceful shutdown of the pool. Finally, the `children_done` atomic boolean variable is used to signal that the children have terminated the BFS.

Thread pinning Upon creation, each thread is pinned to a specific CPU core using the Linux-specific `pthread_setaffinity_np` function. This overrides the default behavior of the operating system's scheduler, which may otherwise migrate threads between cores to balance the overall system load. The primary benefit of this is the enhancement of data locality. As a thread executes, it populates its core's private L1 and L2 caches with frequently accessed data, such as its stack and the graph data from its assigned work chunks. If the thread were to be migrated, these warm caches would be lost, and it would have to repopulate the cold caches of the new core, incurring significant latency from fetching data from the L3 cache or main memory [19].

3.2.3.1 Synchronization Lifecycle

The interaction between the main thread and the worker threads follows a specific order which ensures minimal busy-waiting and efficient CPU usage through the use of condition variables. The state diagram of the main and worker threads is shown in Fig. 3.4.

Initialization and Creation The pool is first initialized with `init_thread_pool()`, which sets up the mutexes and condition variables. Subsequently, `thread_pool_create()` spawns `MAX_THREADS` worker threads. Each thread is assigned a unique ID and it immediately enters a wait state. In this state, each worker thread acquires the children's mutex and enters a while loop that blocks on the `cond_children` condition variable. A worker is only released from this wait when the global `run_id` (controlled by the main thread) is equal to or greater than the worker's own local `run_id`. Upon waking, the worker increments its local `run_id`, effectively "consuming" the work signal, which prepares it to wait for the next cycle after its current task is complete.

Work Dispatch The main thread initiates a BFS traversal by calling `thread_pool_start_wait()`. This function acts as the dispatcher. It first acquires the necessary locks, then increments the global `run_id`, and finally calls `pthread_cond_broadcast()`. This broadcast wakes up all waiting worker threads simultaneously, causing them to exit their wait state and begin executing the main BFS routine.

Completion and Notification After dispatching the work, the main thread immediately blocks by calling `pthread_cond_wait()` on the parent's condition variable, `cond_parent`. It remains in this sleep state, consuming no CPU cycles, until the entire BFS traversal is complete. The last worker thread to finish the final step of the BFS is responsible for calling `thread_pool_notify_parent()`.

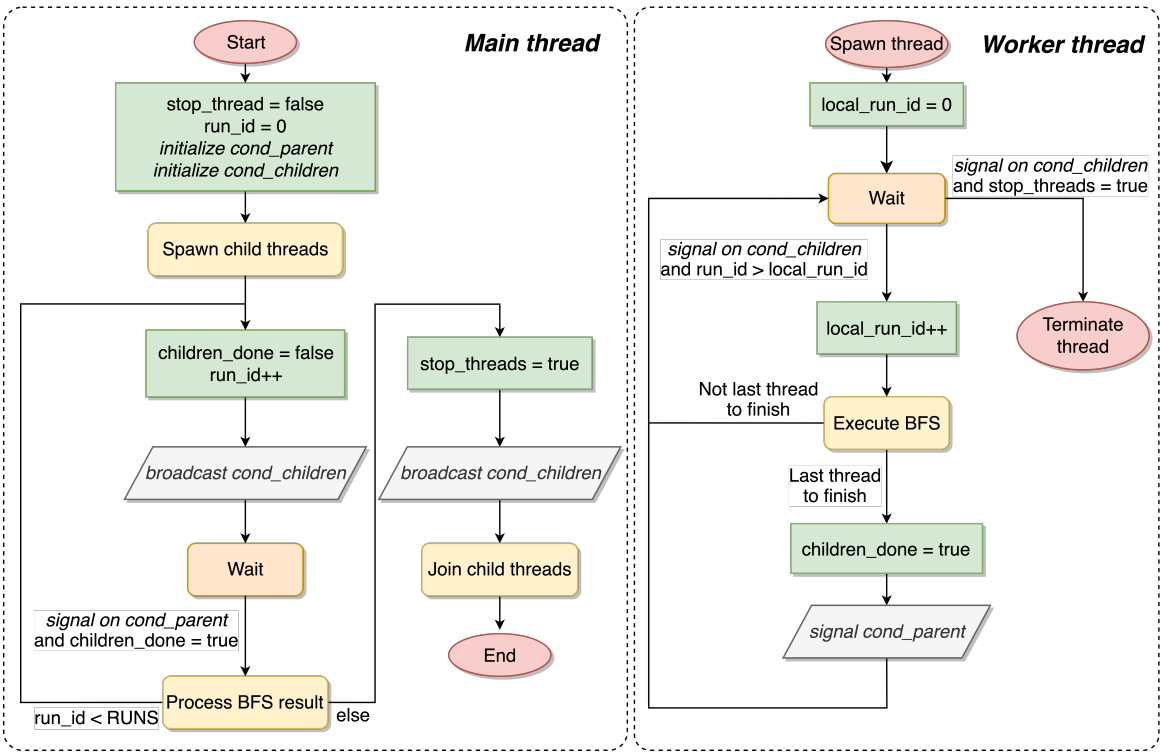


Figure 3.4: State diagrams of the main and worker threads. The `RUNS` variable contains the total number of iterations of the BFS.

This function sets the `children_done` predicate to true and signals `cond_parent`, waking the main thread and allowing the program to proceed.

Termination To shut down the pool, the main thread calls `thread_pool_terminate()`. This function sets the `stop_threads` flag to true and broadcasts one final signal on the `cond_children` condition variable to ensure any sleeping threads wake up. The awakened threads check the `stop_threads` flag and call `pthread_exit` to terminate gracefully. The main thread then calls `pthread_join` on all worker threads to guarantee they have all exited before the program's resources are deallocated.

3.2.4 Level Synchronization and Barrier Implementation

The level-synchronous BFS requires a barrier, which ensures all threads have completed their work for the current level before any thread begins processing the next. This implementation uses a custom barrier that is a variation of the Sense-Reversal Centralized Barrier.

The synchronization is managed using an atomic counter, `active_threads`, and the `distance` variable. At the beginning of each level, `active_threads` is reset to `MAX_THREADS`. When a thread finishes processing all of its work for the current level (including any stolen work), it atomically decrements this counter.

The last thread to finish its work is designated the "leader" for that level. The thread notices that it is the leader because the `active_threads` variable is equal to one when it arrives at the barrier. This leader thread is responsible for performing the serial tasks required between levels: swapping the pointers for the current and next frontiers, checking if the new frontier is empty to determine if the entire BFS should terminate, and finally, incrementing the global `distance` variable. If there are no vertices left in the frontier, instead of incrementing the global `distance` variable, it sets the `exploration_done` boolean variable and notifies the main thread that the BFS has terminated (as explained in Section 3.2.3.1).

The other, non-leader threads, after decrementing the counter, enter a spin-wait loop, continuously checking if the distance has been incremented by the leader. The act of incrementing distance serves as the "sense-reversal" signal, releasing all waiting threads from the barrier simultaneously to begin work on the next level. This design avoids the overhead associated with `pthread_barrier_t` objects

as it relies on a single atomic counter and a shared variable for coordination.

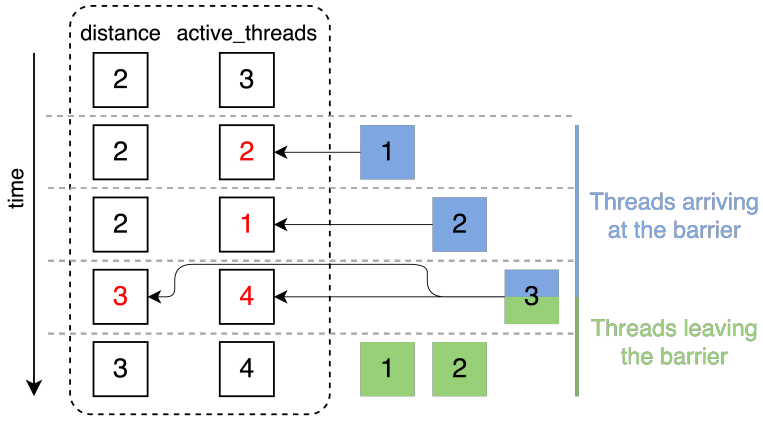


Figure 3.5: The barrier during a BFS exploration at `distance = 2` with 3 worker threads. When threads arrive at the barrier (shown in blue), they decrement the `active_threads` variable (changes to variables are marked in red) and spin-wait until the `distance` variable changes. When the last thread arrives at the barrier, it increments the `distance` and resets `active_threads`, thus releasing all the waiting threads (shown in green).

4 Evaluation

4.1 Evaluation Tools

Conducting a comprehensive performance evaluation of parallel algorithms across multiple High-Performance Computing (HPC) systems presents significant logistical challenges. Each system may have a different architecture, operating environment, and job scheduler configuration. Ensuring that experiments are run consistently, and that results are collected in a structured and reproducible manner, is essential for producing an accurate evaluation. To address these challenges, this work utilized SbatchMan, a Python-based command-line tool designed to automate and manage the submission of batch jobs to HPC clusters running the Slurm Workload Manager and the PBS Workload Manager. The tool is open-source and is available on GitHub[1]. The tool was developed as a joint effort by the author and the thesis co-supervisor, Thomas Pasquali.

4.1.1 SBatchMan design philosophy

The core design philosophy of SbatchMan is the explicit separation of the execution environment from the experimental logic. This is a powerful abstraction that allows the same set of experiments to be deployed across different HPC clusters with minimal changes. This decoupling is achieved through two distinct types of configuration files, both written in the human-readable YAML format.

- Cluster Configuration File: This file defines the hardware and scheduler specifics of a target machine. It contains all the environment-dependent directives, such as the cluster type (e.g., Slurm, PBS, or a local Linux machine), the target partition or queue, the requested number of nodes, CPUs per task, and the wall-clock time limit. By encapsulating these details, the cluster configuration file acts as a reusable profile for a specific HPC system.
- Job Configuration File: This file defines the actual suite of experiments to be run. It references a specific cluster configuration to determine where the jobs will run, but its primary focus is on what to run. This includes the command template for the executable and, most importantly, a powerful variable system for defining the parameter space of the experiments. The user can define a matrix of variables, such as a list of input graph files or a range of thread counts. SbatchMan automatically computes the Cartesian product of all specified variables, generating a unique job for every possible combination. This file also allows for the setting of environment variables and the definition of pre-execution or post-execution commands, providing full control over the job's lifecycle.

In the context of this thesis, SbatchMan was used to conduct the evaluation of the OpenMP and the pthreads-based BFS implementation. It was used to systematically vary the number of threads, the input graph datasets and the implementations' configuration parameters. By using a single job configuration file and multiple cluster-specific files, SbatchMan ensured that the evaluation conditions were identical across all runs on all target systems, thereby eliminating a significant source of potential experimental error.

4.1.2 Other tools

To facilitate the efficient loading of graph data, both implementations utilize the `distributed_mmio` library[2]. This C++ library provides a simple and high-performance interface for reading and parsing matrices stored in the Matrix Market (.mtx) file format. Many common benchmark graphs, such as those from collections like the SuiteSparse Matrix Collection[3], are available in this format, simplifying

[1] `https://github.com/LorenzoPichetti/SbatchMan`
[2] `https://github.com/HicrestLaboratory/distributed_mmio`
[3] `https://sparse.tamu.edu/`

the process of importing real-world data for evaluation. The library is designed for efficiency, capable of quickly parsing large graph files into a memory-efficient representation. To enable its use in the C-based pthreads implementation, a C wrapper was developed as a contribution to the project, exposing the core C++ parsing functionality through a C-compatible API.

The acquisition and generation of graph datasets for the evaluation were managed using MtxMan[4], a command-line tool designed to streamline graph data management. This tool simplifies the process of downloading and extracting graphs from the SuiteSparse Matrix Collection. Additionally, MtxMan includes generators for creating synthetic graphs with specific structural properties, which is valuable for testing algorithmic performance under controlled conditions.

4.2 Evaluation Datasets

The practical relevance of a Breadth-First Search algorithm extends beyond being a benchmark of computing power. The graphs chosen for evaluation in this work apply BFS as either a direct solution or a fundamental building block for solving real-world problems. Given that the MergedCSR structure adapts well only to graphs with smaller frontiers, the selection of graphs classes was restricted to the ones that have a consistent frontier size (see Fig. 2.2). The selected graphs come from the following domains: Road Networks and Random Geometric Graphs (RGGs). The selected graphs are taken from the SuiteSparse Matrix Collection and their characteristics are summarized in Table 4.1.

Road networks For road networks, the application of BFS is intuitive: it is the foundational algorithm for pathfinding and reachability analysis. In this context, vertices represent intersections and edges represent road segments [41]. On these graphs, the BFS is applied to solve problems involving navigation, logistics, and emergency services.

Finite Element Models (FEM) In the context of finite element analysis, vertices represent nodes in a physical mesh and edges represent the connections between them [10]. Here, BFS is used to model the propagation of physical phenomena. This is applied to perform stress and strain analysis and crack propagation simulations.

Random Geometric Graphs (RGGs) Random geometric graphs serve as theoretical models for real-world networks where connectivity is determined by physical proximity [36]. For example, RGGs are used to model wireless networks, where nodes can only communicate with others within a certain range or for epidemiology and contagion modeling in a population, since the spread of a disease occurs through close contact among individuals [24, 17].

Name	Graph class	Vertices	Edges	Notes
GAP-Road	Road Network	23.9M	57.7M	Road network of the USA.
europe_osm	Road Network	50.9M	108M	Road network of Europe.
asia_osm	Road Network	11.9M	25.4M	Road network of Asia.
Geo_1438	FEM	1.4M	60.2M	Geomechanical model of Earth's crust.
Hook_1498	FEM	1.4M	59.4M	3D model of a steel hook.
PFlow_742	FEM	0.7M	37.1M	3D pressure-temp in porous media.
rgg_n_2_22_s0	RGG	4.2M	60.7M	Random Geometric Graph.

Table 4.1: Datasets used in the evaluation.

4.3 Evaluation Environment

The C++/OpenMP implementation presented in Section 3.1 and the C/pthreads implementation presented in Section 3.2 are compared against the BFS implementation from the GAP benchmark.

[4] `https://github.com/ThomasPasquali/MtxMan`

The evaluation was conducted on the following platforms:

- `Amd`: AMD EPYC 7543 CPU @ 2.8 GHz (32 cores), with 32 KiB L1d cache, 32 KiB L1i cache and 512 KiB L2 cache per core, 256 MiB shared L3 cache, and 32 GB DDR4 RAM @ 3.2 GHz, nominal TDP 225W;
- `Pioneer` board: Sophon SG2042 RISC-V CPU @ 2.0 GHz (64 cores), with 64 KiB L1d cache and 64 KiB L1i cache per core, 1 MiB L2 cache shared per 4-core cluster, 64 MiB shared L3 cache, and 128 GB DDR4 RAM, nominal TDP 120W [8];
- `Grace`: NVIDIA Grace CPU Superchip @ up to 3.0 GHz (144 cores), with 64 KiB L1d cache, 64 KiB L1i cache and 1 MiB L2 cache per core, 228 MiB shared L3 cache, and 960 GB LPDDR5X RAM, nominal TDP 500W (including memory) [16].

For each dataset, performance was measured by starting the BFS from 32 random source vertices, with the first two runs discarded to mitigate warm-up effects. The geometric mean of the subsequent runtimes was taken as the final metric for each graph. When aggregating results, the geometric mean of these individual metrics was used to represent the performance for each graph class.

4.4 Results

This section presents a comparative performance evaluation of the two parallel BFS implementations developed in this thesis: the cache-optimized OpenMP version from Section 3.1 (hereafter `openmp`) and the explicit pthreads version from Section 3.2 (hereafter `pthreads`). To contextualize their performance, these two approaches are benchmarked against the BFS kernel from the GAP Benchmark Suite (`gapbs`). The `gapbs` implementation serves as a widely recognized baseline, providing a standard point of comparison common in graph analytics literature. The implemented code along with reproducibility instructions is available at `https://github.com/sasso0101/thesis`.

The performance analyses presented in Sections 4.4.1 to 4.4.4 are conducted on the `Amd` platform. Section 4.4.5 then provides a comparative evaluation, benchmarking the performance of the AMD platform against the RISC-V-based Pioneer board and the ARM-based NVIDIA Grace CPU.

4.4.1 Evaluation of the optimal chunk size

The `CHUNK_SIZE` is a key parameter in the pthreads implementation, defining the granularity of the work partitioning and load balancing system. A sensitivity analysis was conducted to find an optimal value, testing sizes from 4 to 1024. The results, shown in Fig. 4.1, reveal that the optimal granularity is highly dependent on the degree of parallelism.

In the single-threaded case, where no parallel work-stealing occurs, performance is dictated by the overhead of the chunking mechanism, with larger chunk sizes often proving more efficient. As the thread count increases, a clear trade-off between management overhead and load balancing effectiveness emerges. A `CHUNK_SIZE` of 64, highlighted by the hatched bars, proves to be the most effective choice across the majority of multi-threaded scenarios. Smaller chunks increase the frequency of lock contention and queue management, while very large chunks minimize this overhead but provide too few opportunities for the work-stealing mechanism to effectively balance the load. Based on this analysis, a value of 1024 was used for single-threaded executions, while a value of 64 was used for all multi-threaded configurations in subsequent evaluations.

4.4.2 Strong scaling analysis

Fig. 4.2 presents a strong scaling analysis of the `gapbs`, `openmp`, and `pthreads` implementations. All three implementations exhibit effective strong scaling at lower core counts. However, their behavior diverges at 16 and 32 threads. The `gapbs` and `openmp` implementations, which share similar framework-based designs, show performance degradation, particularly on the large Road Network datasets. This negative scaling is likely due to the synchronization overhead overwhelming the computational work at high core counts. In contrast, the `pthreads` implementation demonstrates more robust scaling, due to the lightweight barrier and dynamic work-stealing, which mitigates contention. The `asia_osm` graph does not scale well because its frontier sizes are the smallest of the evaluated graphs (the `asia_osm` graph has the lowest frontier size line in Fig. 2.2), which limits the amount of parallelism that can be exploited.

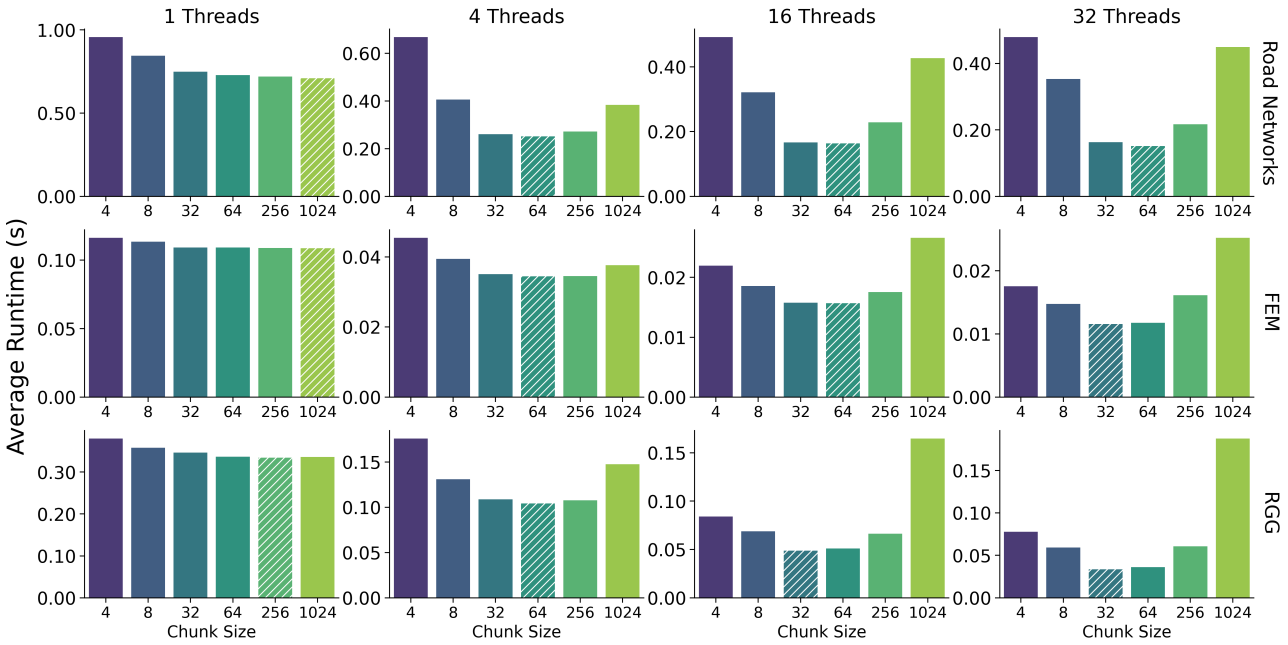


Figure 4.1: Average runtime of the pthreads BFS implementation across various graph classes (rows), thread counts (columns), and `CHUNK_SIZE` configurations (colored bars). The hatched bar in each subplot indicates the `CHUNK_SIZE` that yielded the best performance for that specific configuration.

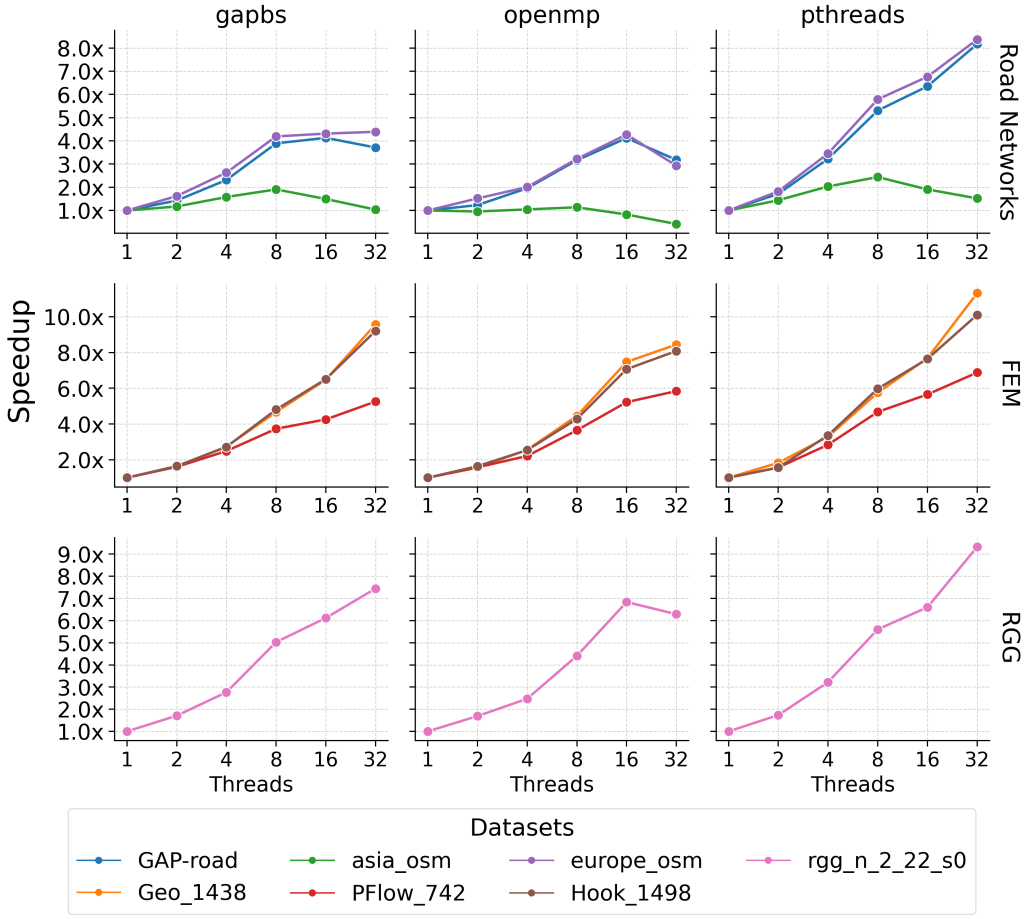


Figure 4.2: Scaling comparison of the evaluated implementations (columns) across various graph classes (rows). The plots show the speedup of each configuration as the number of CPU threads increases, relative to the one-threaded performance.

4.4.3 Performance of the MergedCSR implementation

To quantify the performance impact of the cache-optimized MergedCSR data structure, the `openmp` implementation was compared directly against the `gapbs` baseline. Fig. 4.3 shows the speedup of the `openmp` version relative to `gapbs` across the three graph classes.

For the FEM graphs, the `openmp` implementation is consistently slower than the `gapbs` baseline, with a speedup below 1. The FEM graphs, while originating from mesh-based models, exhibit small-diameter characteristics such as rapid frontier growth. The `gapbs` implementation, being direction-optimizing, detects this and leverages the more efficient Bottom-Up traversal strategy during the dense, intermediate phases of the search. For instance, on the `Hook_1498` dataset, the Bottom-Up step is activated five times. In contrast, the `openmp` implementation is restricted to a Top-Down-only traversal due to the design of its MergedCSR data structure. Consequently, it is at a significant algorithmic disadvantage on these specific graphs, as it is forced to use a less optimal strategy.

On the Road Networks and the RGG datasets, the frontiers instead remain small, and both `gapbs` and `openmp` exclusively use the Top-Down strategy. With the traversal algorithm being identical, the performance difference is now dictated by the efficiency of the underlying data structure. In this context, the benefits of the MergedCSR format are clearly visible. By improving spatial locality and reducing cache misses, the `openmp` implementation significantly outperforms the baseline. This results in a geometric mean speedup of 1.47x on the Road Network datasets and 1.5x on the RGG dataset. This demonstrates that for workloads that are purely Top-Down, the cache-optimized data layout provides a substantial performance improvement over a standard CSR implementation.

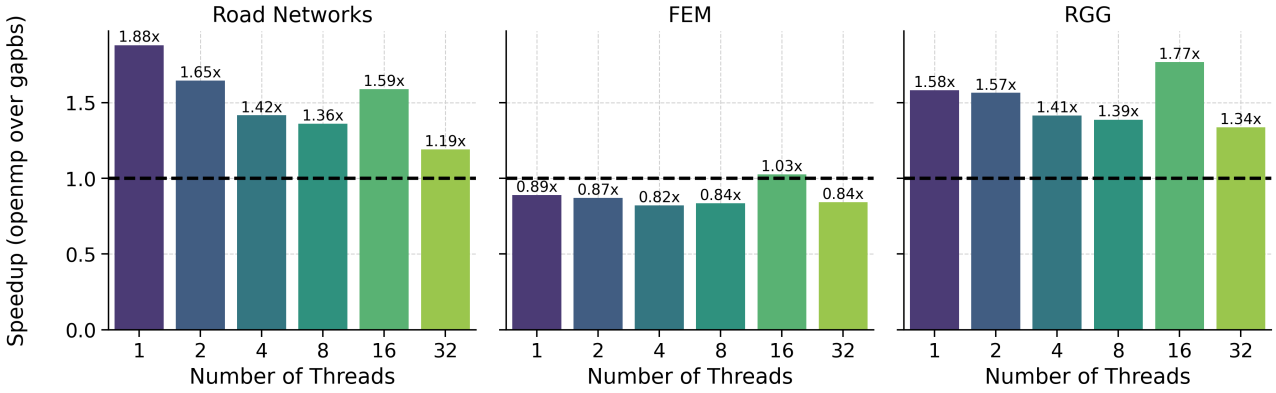


Figure 4.3: Speedup of the `openmp` implementation relative to the `gapbs` baseline, grouped by graph class.

4.4.4 Performance of the pthreads implementation

To isolate the performance impact of the parallelization strategy, this section directly compares the explicit `pthreads` implementation against the framework-based `openmp` version. Both implementations utilize the same cache-optimized MergedCSR data structure, making this a direct comparison of their respective parallel execution models. Fig. 4.4 illustrates the speedup of the `pthreads` version relative to the `openmp` implementation across the three graph classes.

The results again show a clear distinction in performance based on the graph type. On the FEM datasets, the performance of the `pthreads` implementation is on par with the `openmp` version. This is an expected outcome, as both implementations are constrained to a Top-Down-only traversal.

In contrast, on the large-diameter graphs where the Top-Down strategy is optimal, the benefits of the explicit `pthreads` implementation become evident. The lightweight barrier and the dynamic work-stealing mechanism of the `pthreads` version prove to be significantly more efficient and scalable than the general-purpose primitives provided by the OpenMP runtime. This results in a substantial performance advantage that grows with the number of threads, particularly visible at 32 threads for the Road Networks where the `pthreads` version is nearly 2.8x faster. Quantitatively, this performance advantage translates to a geometric mean speedup of 1.55x for the `pthreads` implementation across the Road Network datasets and 1.25x on the RGG dataset. Compared to the GAP benchmark, the

achieved geomean speedup was 2.28x for the Road Network graphs (up to 3.6x for the `osm_europe` graph) and 1.87x for the RGG dataset.

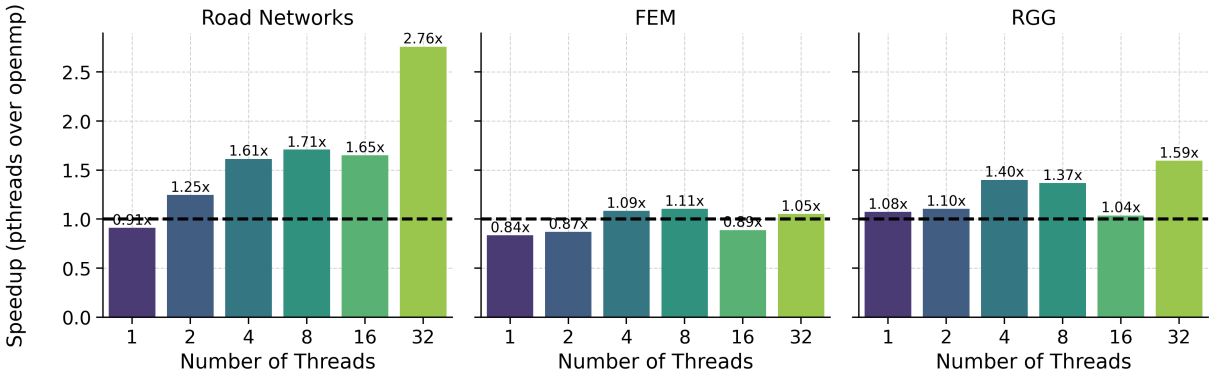


Figure 4.4: Speedup of the `pthreads` implementation relative to the `openmp` implementation, grouped by graph class.

4.4.5 Performance on diverse hardware platforms

To provide a comprehensive analysis of the pthreads implementation's performance and portability, its strong scaling was evaluated on three distinct hardware platforms: the x86-based AMD EPYC, the ARM-based NVIDIA Grace CPU, and the RISC-V-based Sophon SG2042 (Pioneer board). The `europe_osm` graph was selected for this comparison as it is the largest dataset evaluated, and the pthreads implementation was chosen as it demonstrated the most robust scaling in the previous analyses. Figure Fig. 4.5 presents the absolute runtime and relative speedup for this configuration.

The results reveal significant architectural differences in both performance and scalability. The single-threaded performance is a direct reflection of each CPU's single-core processing power. The NVIDIA Grace and AMD EPYC CPUs exhibit comparable performance, with Grace being slightly faster. In contrast, the Pioneer board's RISC-V CPU is initially much slower, taking over 10 seconds. This is attributable to its lower clock frequency and lower Instructions Per Clock (IPC) compared to the more mature x86 and ARM server architectures. However, as the thread count increases, the Pioneer board's massive parallelism allows it to dramatically reduce its runtime achieving an almost linear speedup up to 8 threads, and a 15.2x speedup for 32 threads. The Grace and AMD CPUs exhibit similar performance up to 16 threads, but the Grace CPU's performance drops at 32 threads, likely due to the custom sense-reversal barrier's poor interaction with the cache coherence protocol of Grace's massively parallel architecture.

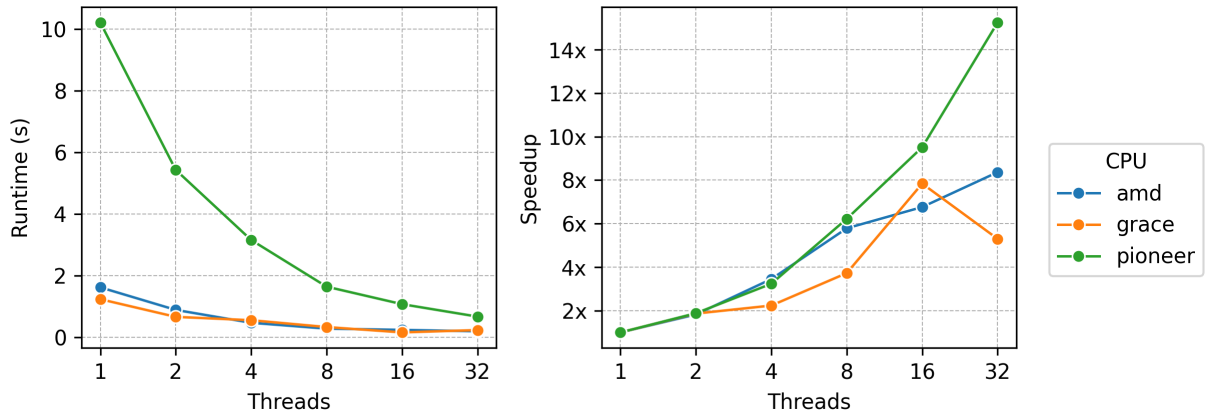


Figure 4.5: Strong scaling performance of the `pthreads` implementation on the `europe_osm` graph across three different hardware platforms. The left panel shows the absolute average runtime in seconds. The right panel shows the corresponding speedup relative to the single-threaded performance on each respective platform.

5 Conclusions

The efficient execution of Breadth-First Search, a foundational algorithm in graph analytics, is a significant challenge on modern multicore architectures. The primary performance bottleneck is not computational complexity but rather the irregular memory access patterns inherent in traversing large-scale graphs, which leads to poor cache utilization. This thesis addressed this challenge by investigating, implementing, and evaluating two distinct parallel optimization strategies specifically tailored for large-diameter graphs, where the Top-Down traversal approach is most effective. The central hypothesis was that a holistic optimization approach, which considers both the data structure's memory layout and the parallel execution model, is necessary to achieve high performance and scalability.

This work presented two primary contributions. The first was a cache-optimized BFS implementation in C++ using OpenMP, built upon a novel MergedCSR data structure. This format improves spatial locality by co-locating a vertex's metadata directly with its adjacency list. The second contribution was a case study in explicit parallelization, an implementation written in C using the pthreads library. This version provided fine-grained control over the parallel execution model, featuring a persistent thread pool, a dynamic work-stealing mechanism, and a lightweight, custom sense-reversal barrier.

The performance evaluation, conducted across a diverse range of hardware platforms and real-world datasets, yielded several key findings. Firstly, the MergedCSR data structure proved highly effective for purely Top-Down workloads. The OpenMP implementation using this structure achieved a geometric mean speedup of up to 1.5x over the GAP Benchmark Suite baseline on road networks and random geometric graphs, validating the hypothesis that aligning the data structure with the memory hierarchy is critical in performance optimization.

Secondly, the comparative analysis of parallelization strategies revealed the limitations of high-level frameworks for this specific problem domain. While the OpenMP implementation was effective at low core counts, it exhibited negative scaling at higher thread counts due to the overhead of its general-purpose synchronization primitives. In contrast, the explicit pthreads implementation demonstrated superior scalability, achieving a geometric mean speedup of up to 1.55x over the OpenMP version. This finding underscores the importance of a custom parallel execution model for algorithms with fine-grained synchronization. The study also highlighted the limitations of a specialized, Top-Down-only approach, as both implementations were outperformed by the direction-optimizing gapbs on datasets with small-diameter characteristics.

Finally, the evaluation on diverse hardware platforms highlighted the importance of architectural awareness. While the pthreads implementation scaled robustly on the x86-based AMD platform and showed excellent scaling on the RISC-V-based Pioneer board, it suffered from a sharp performance degradation at high core counts on the ARM-based NVIDIA Grace CPU.

Several directions for future research emerge from this work. The most immediate is the extension of these implementations to support a hybrid, direction-optimizing strategy. This would involve adapting the MergedCSR format to support the Bottom-Up traversal's bitmap-based lookups. Another promising direction is the application of these principles, namely, cache-aware data layouts and scalable custom synchronization, to other memory-bound graph algorithms, such as Single-Source Shortest Path or PageRank.

Beyond algorithmic extensions, the OpenMP implementation and the pthreads implementation could serve as a benchmark for evaluating the overhead and scalability of the two parallelization models. Furthermore, for the pthreads implementation, the synchronization mechanisms themselves can be implemented with different primitives, such as atomics or NUMA-aware locks. This modularity allows the implementation to serve as a benchmark for evaluating the performance of these synchronization constructs on different architectures. Also, the existing sense-reversal barrier could be replaced with

other mechanisms, such as tree-based or tournament barriers, to analyze how different synchronization strategies interact with a given platform's cache coherence protocol. As demonstrated by the performance difference on the NVIDIA Grace CPU, this approach can reveal performance bottlenecks in both hardware architectures and compiler implementations.

Finally, a more in-depth investigation into the cache behavior using low-level performance counters could further determine the reasons for the observed performance differences. Such an analysis, combined with a direct evaluation of the energy consumption of these implementations, would provide valuable insights, explicitly linking reduced runtimes and fewer memory stalls to improvements in energy efficiency on modern processors.

Bibliography

[1] Luis A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. Classes of small-world networks. *Proceedings of the national academy of sciences*, 97(21):11149–11152, 2000.

[2] Salvatore Domenico Andaloro, Thomas Pasquali, and Flavio Vella. Cache-optimized bfs on multicore cpus. In *Proceedings of the 1st FastCode Programming Challenge*, pages 23–27. 2025.

[3] Junya Arai, Masahiro Nakao, Yuto Inoue, Kanto Teranishi, Koji Ueno, Keiichiro Yamamura, Mitsuhisa Sato, and Katsuki Fujisawa. Doubling graph traversal efficiency to 198 terateps on the supercomputer fugaku. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2024.

[4] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[5] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[6] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[7] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558, 2010.

[8] Nick Brown and Maurice Jamieson. Performance characterisation of the 64-core sg2042 risc-v cpu for hpc. In *International Conference on High Performance Computing*, pages 354–367. Springer, 2023.

[9] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, 23(89):1–64, 2022.

[10] Xiaolin Chen and Yijun Liu. *Finite element modeling and simulation with ANSYS Workbench*. CRC press, 2018.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[12] Limeng Cui, Haeseung Seo, Maryam Tabar, Fenglong Ma, Suhang Wang, and Dongwon Lee. Deterrent: Knowledge guided graph attention network for detecting healthcare misinformation. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 492–502, 2020.

[13] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[14] Joshua Dennis Booth and Phillip Allen Lane. An adaptive self-scheduling loop scheduler. *Concurrency and Computation: Practice and Experience*, 34(6):e6750, 2022.

[15] Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*, pages 287–290. 2022.

[16] Anne C Elster and Tor A Haugdahl. Nvidia hopper gpu and grace cpu highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022.

[17] Ernesto Estrada, Sandro Meloni, Matthew Sheerin, and Yamir Moreno. Epidemic spreading in random rectangular networks. *Physical review E*, 94(5):052316, 2016.

[18] Lester R Ford Jr and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[19] Brahmaiah Gandham and Praveen Alapati. Occ pinning: Optimizing concurrent computations through thread pinning. In *Proceedings of the 2024 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, pages 1–5, 2024.

[20] Chen Gao, Yu Zheng, Nian Li, Yinfeng Li, Yingrong Qin, Jinghua Piao, Yuhan Quan, Jianxin Chang, Depeng Jin, Xiangnan He, et al. A survey of graph neural networks for recommender systems: Challenges, methods, and directions. *ACM Transactions on Recommender Systems*, 1(1):1–51, 2023.

[21] SL Gonzaga de Oliveira, MI Santana, DN Brandão, and C Osthoff. An openmp-based breadth-first search implementation using the bag data structure. *Concurrency and Computation: Practice and Experience*, 36(16):e8119, 2024.

[22] Keita Iwabuchi, Hitoshi Sato, Yuichiro Yasui, Katsuki Fujisawa, and Satoshi Matsuoka. Nvm-based hybrid bfs with memory efficient data structure. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 529–538. IEEE, 2014.

[23] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C Hart. Edge v. node parallelism for graph centrality metrics. In *GPU Computing Gems Jade Edition*, pages 15–28. Elsevier, 2012.

[24] Hichem Kenniche and Vlady Ravelomananana. Random geometric graphs as model of wireless sensor networks. In *2010 The 2nd international conference on computer and automation engineering (ICCAE)*, volume 4, pages 103–107. IEEE, 2010.

[25] Charles E Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, 2009.

[26] Charles E Leiserson and Tao B Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 303–314, 2010.

[27] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[28] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 30(10):1852–1872, 2018.

[29] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[30] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[31] Benjamin W Mezger, Douglas A Santos, Luigi Dilillo, Cesar A Zeferino, and Douglas R Melo. A survey of the risc-v architecture software support. *IEEE Access*, 10:51394–51411, 2022.

[32] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19(45-74):22, 2010.

[33] Mark EJ Newman. Community detection and graph partitioning. *Europhysics Letters*, 103(2):28003, 2013.

[34] Yuyao Niu and Marc Casas. Berrybees: Breadth first search by bit-tensor-cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 339–354, 2025.

[35] H Timucin Ozdemir and Chilukuri K Mohan. Flight graph based genetic algorithm for crew scheduling in airlines. *Information Sciences*, 133(3-4):165–173, 2001.

[36] Mathew Penrose. *Random geometric graphs*, volume 5. OUP Oxford, 2003.

[37] Tahereh Pourhabibi, Kok-Leong Ong, Booi H Kam, and Yee Ling Boo. Fraud detection: A systematic literature review of graph-based anomaly detection approaches. *Decision Support Systems*, 133:113303, 2020.

[38] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[39] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[40] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[41] Robert C Thomson and Dianne E Richardson. A graph theory approach to road network generalisation. In *Proceeding of the 17th international cartographic conference*, pages 1871–1880, 1995.

[42] Jesmin Jahan Tithi, Dhruv Matani, Gaurav Menghani, and Rezaul A Chowdhury. Avoiding locks and atomic instructions in shared-memory parallel bfs using optimistic parallelization. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1628–1637. IEEE, 2013.

[43] Ryan Torok. Improving graph workload performance by rearranging the csr memory layout. 2020.

[44] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory performance of amd epyc rome and intel cascade lake sp server processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 165–175, 2022.

[45] Stephan M Wagner and Nikrouz Neshat. Assessing the vulnerability of supply chains using graph theory. *International journal of production economics*, 126(1):121–129, 2010.

[46] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. Understanding graph sampling algorithms for social network analysis. In *2011 31st international conference on distributed computing systems workshops*, pages 123–128. IEEE, 2011.

[47] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, Bo Long, et al. Graph neural networks for natural language processing: A survey. *Foundations and Trends® in Machine Learning*, 16(2):119–328, 2023.

[48] Hai-Cheng Yi, Zhu-Hong You, De-Shuang Huang, and Chee Keong Kwoh. Graph representation learning in bioinformatics: trends, methods and applications. *Briefings in Bioinformatics*, 23(1):bbab340, 2022.

[49] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 183–193, 2015.