Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/36/20230443

Efficient parallelism in Breadth-First Search: A comprehensive analysis and implementation

Jianlin Li

Computer Science and Technology, Southern University of Science and Technology, Shenzhen, 518055, China

12012221@mail.sustech.edu.cn

Abstract. The Breadth-First Search (BFS) entails a systematic traversal of a given graph, G = (V, E), layer by layer, starting from a specific vertex. Recognized as a cornerstone methodology for graph exploration, the importance of BFS has skyrocketed, especially with the increasing demands of graph-based data processing. However, as the vertex count expands, traditional serial implementations reveal their limitations, faltering in terms of time and space efficiency. This paper aims to contrast the efficiencies of standard BFS with its parallelized iteration. Introducing a shared-memory model of level-synchronous parallel BFS, the approach integrates optimizations to navigate the challenges posed by implicit barriers and critical sections. Employing the Graph500 benchmark, this parallel methodology is meticulously evaluated, highlighting the speedup concerning various thread counts. Initial findings unveil a compelling pattern: speedup generally correlates positively with the number of active threads. However, if the thread count breaches the system's inherent capacity, the speedup hits a plateau, showing only marginal fluctuations without significant increases. These statistical revelations not only vouch for the advantages of BFS parallelization but also emphasize a critical insight: judiciously increasing thread count, up to a system-specified limit, can yield peak efficiency.

Keywords: Breadth-first Search, Parallelization, Efficiency.

1. Introduction

In light of the recent proliferation of high-performance multi-core processors, parallelization has taken a formidable leap forward. Paving the way are mature APIs, notably OpenMP and pthreads, grounded in shared memory, alongside MPI, which embodies distributed-memory parallelism. Particularly potent when harnessed for algorithms characterized by parallelizable code sections or iterative tasks, parallelization has proven its mettle across diverse domains: from image processing and big data analytics to intricate scientific and technological simulations.

The Breadth-First Search algorithm, a cornerstone in algorithmic design, systematically navigates a graph layer-by-layer, commencing from a designated source vertex [1]. While a traditional BFS boasts a time complexity of O(V+E) – where V denotes vertices and E symbolizes edges – its efficiency wanes with burgeoning graph sizes [2]. This is noteworthy, given the pervasive presence of vast graph structures in realms as varied as social network analytics, web semantic evaluations, bioinformatics, and natural language processing (NLP). Addressing the inherently data-intensive BFS amidst escalating graph dimensions demands a rethink: blending BFS with parallelization. Such an amalgamation seeks

DOI: 10.54254/2755-2721/36/20230443

to harness the computational prowess of multiprocessor systems, especially as transistor counts on processors plateau [3]. Consequently, real-world endeavors, be it web crawling, web content indexing innovations, search engine optimizations, or data-centric applications, stand to gain considerably, underscored by comparative work-efficiency analyses [4]. This manuscript introduces a level-synchronous parallel BFS approach, anchored in a shared-memory paradigm. It emphasizes nuanced optimizations, tailored to curtail layer synchronization expenses and mitigate race condition-related overheads. Additionally, an exploration into the speedup of this parallel construct, mapped against thread counts, is undertaken [5]. Complementing this, a detailed analysis probes the interplay between achieved speedup and the number of operational threads.

2. Parallelism and Graph

2.1. Graph-Traversal and BFS Issues

Breadth first search is a fundamental graph traversal algorithm that is widely used in various applications such as social network analysis, web crawling, recommendation systems, etc. Given a graph G = (V, E), the time complexity of Breadth first search is O (|V| + |E|). But as the number of vertices increases, the sequential implementation becomes inefficient in terms of both time and space complexity. For instance, using a standard serial BFS to traverse through a complete graph with 100,000,000 vertices may lead the system to kill the process [6]. So here comes parallelization, and the goal of BFS parallelization is to develop a parallel implementation of BFS to distribute the vertices across all cores to traverse concurrently.

Other complications exist such that there are many implementations for serial BFS (e.g., queue-based, frontier-based, etc., so which version to use determines the effectiveness of the parallel counterpart [7]; graphs are irregular data structures, the dependencies of data of which impose a challenge to parallelization.

2.2. Parallel Implementation of BFS

Table 1. Serial implementation of Breadth-First Search.

Algorithm 1 serial_BFS

```
Input: G = (V, E), src;
levels := a collection of level values for a given vertex v;
for vertex v in V do
    levels[v] := -1;
end for
frontier := {}, next_frontier := {};
levels[src] := 0, curr_level = 0;
insert src to frontier;
while frontier is not empty do
    for vertex v in frontier do
        for neighbor n of v do
            if levels[n] == -1 then
                insert n to next_frontier;
                levels[n] := curr_level + 1;
            end if
        end for
        frontier := next_frontier;
        next_frontier := {};
        curr_level := curr_level + 1;
    end for
end while
```

DOI: 10.54254/2755-2721/36/20230443

Based on common BFS algorithms, there are two containers of vertices that are used in this BFS implementation, called frontier and next_ frontier, where frontier is used to keep track of all vertices that are equally distant from the source vertex src, and next_ frontier is used to store all vertices adjacent to those in frontier that have not been visited. This implementation explores the potential parallelism by separating distinct layers of a graph so that each layer of vertices can be processed in parallel without the interference from the next layer. The table 1 shown below is the pseudocode of the serial implementation of BFS [8].

Then, based on the serial implementation of BFS, it's a necessity to explore the most time-consuming code block to achieve higher improvement. Therefore, code profiling is conducted to analyse the time consumption with Very Sleepy Profiler [9]. And the configurations of the graph used for profiling are listed in table 2, the corresponding profiling results are shown in figure 1.

Table 2. Configuration of the graph used for profiling.

Test Graph	
Number of nodes	54.9K
Number of edges	1.3M
Maximum degree	275
Minimum degree	11
Average degree	24

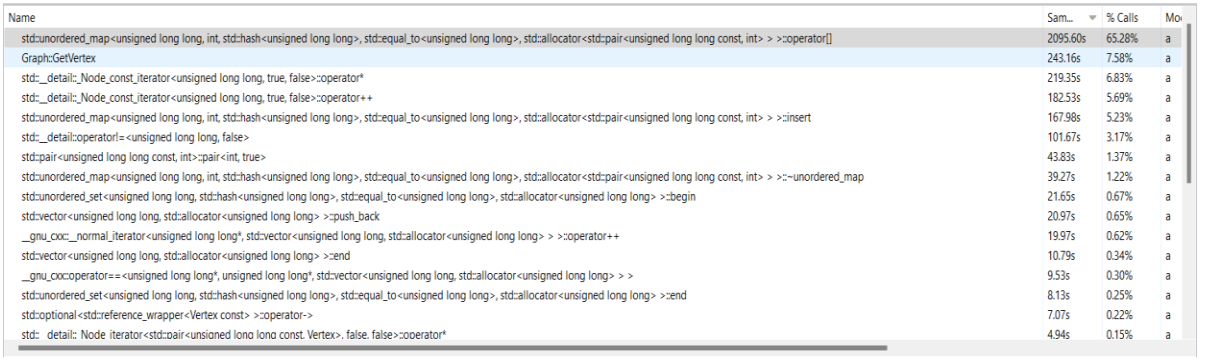

Figure 1. Profiling result (Photo/Picture credit: Original).

It is worth noting that according to the profiling result, checking vertices' level cost the most time (the program check level one vertex at one time in serial algorithm). Then the parallelization can be applied here. #pragma omp for nowait is applied to this process, which not only realizes parallelism, but also removes the implicit barrier at the end of each iteration. Moreover, since the atomic-operation section is introduced in table 1, line 12 and 13, no synchronization problem is supposed to exist.

On top of the parallelization, for the fact that BFS inherently explores neighbouring vertices independently, parallelizing an ordered serial BFS would enforce a specific order of execution among iterations, potentially leading to unnecessary synchronization overhead and limiting the parallelism, which could degrade performance [10]. Consequently, an unordered version of BFS is chose to handle the problem. The problem and its solution stated in the following paragraph clearly explains how the unordered algorithm is proposed and achieved.

Another potential race condition still exists that multiple threads are likely to add vertices synchronously to next_ frontier. Then, the optimal solution is proposed, which is to have each of the threads takes over and monitor its own work by adding a new container local_ next_ frontier, and finally all the vertex are added to next_frontier when every single thread completes its execution. Due to the unordered visiting sequence of the vertex, it's certainly an unordered algorithm.

DOI: 10.54254/2755-2721/36/20230443

3. Results and Analysis

3.1. Dataset and Specifications

To verify the effectiveness and robustness of the parallel BFS algorithm, authentic real-world graph data was procured from an online real-graph dataset networkrepository.com. The chosen data consists of various graphs including sparse or dense graphs, connected or unconnected graphs and directed or undirected graphs in order to ensure the fully-consideration and validity of the result. The table 3 and table 4 shows the configurations of two graphs with considerable size that are mainly used in the experiments, and comparison of serial and parallel BFS with analyses are derived from the results based on them.

Table 3. Configuration of the first sample graph.

Sample Graph 1	
Number of nodes	30.8K
Number of edges	1.3M
Maximum degree	4.6K
Minimum degree	1
Average degree	82

Table 4. Configuration of the second sample graph.

Sample Graph 2	
Number of nodes	18.8K
Number of edges	198.1K
Maximum degree	498
Minimum degree	0
Average degree	21

It can be easily seen in the tables that the second graph is an undirected graph and a unconnected graph with minimum degree of 0. Also, the first graph is comparatively denser and larger than the second graph in size, which promises the diversity of the dataset used to conduct the experiment. Meanwhile, in order to enhance the validity and persuasiveness of the results from the experiments conducted on the graphs, the system specifications of our experiments are supposed to be listed in the table 5.

Table 5. System specification.

System Specifications	
Processor	AMD RYZEN 7 PRO 4750U MOBIEL PROCESSOR WITH AMD RADEON GRAPHICS
Installed RAM	32.0GB (31.2GB usable)
System type	64-bit operating system, x64-based processor
Number of CPU cores	8
Number of threads	16
L1 cache	512KB
L2 cache	4MB
L3 cache	8MB
Max boost clock	<=4.1GHz
Base clock	1.7GHz
Operating system edition	Windows 11 Pro
Operating system version	22H2
Operating system build	22621.2070

3.2. Efficiency Comparison: Serial vs. Parallel BFS

A comprehensive experimental framework encompassing a spectrum of graph sizes and characteristics facilitated an in-depth performance evaluation of the parallel algorithm against the serial version. Generally, there are 500 experiments run on each real graph that has been collected. In each of the experiment, the time taken by the serial BFS (in milliseconds) is measured, and the time taken by the parallel BFS (in milliseconds) is also measured at the same time, both using the omp_ get_ wtime () function from < omp. h >, providing accurate benchmarks across varying graph sizes, machine

DOI: 10.54254/2755-2721/36/20230443

configurations, and thread counts. These recorded times were instrumental in calculating average execution times for both the serial and parallel BFS algorithms. Besides, there are two variables, time_ serial and time_ parallel, that are used to keep track of the accumulated time of serial BFS and parallel BFS, respectively.

Moreover, upon finishing all the experiments, the average execution time for both algorithms is calculated based on the formula: average execution time = total execution time / number of experiments, where number of experiments is set to be 500 here.

Then, the speedup of the parallel BFS algorithm is defined as the ratio between the time (measured in milliseconds) taken by the serial BFS and the time (also measured in milliseconds) taken by the parallel BFS. Then the experiments are conducted based on all the methods introduced in previous contents. And the maximum number of threads is set to be 32 for both graphs, which in sufficient for the fact that the system number of threads is 16. The corresponding results of the two graphs (speedup versus number of threads) are shown in figure 2 and figure 3. Notice that the lateral axis NumThds stands for the number of threads executing in parallel, and vertical axis is set to be speedup that has been previously defined.

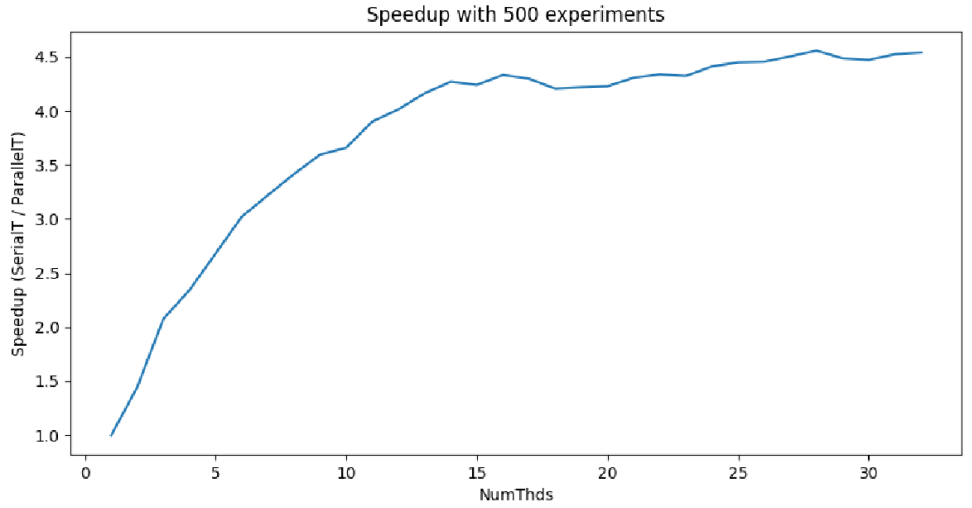

Figure 2. Result based on the first graph (Photo/Picture credit: Original).

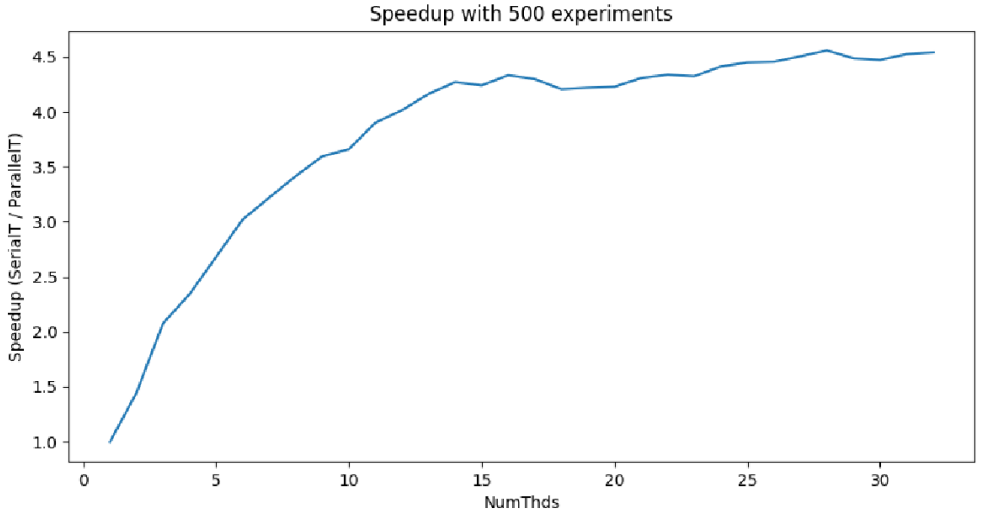

Figure 3. Result based on the second graph (Photo/Picture credit: Original).

DOI: 10.54254/2755-2721/36/20230443

3.3. Analysis of Outcomes

To ensure the authenticity of all the speculations and analyses from the experiment results, all the prerequisites are considered to be satisfied and sufficient, including the enough number of experiments, same BFS and parallelization strategy for both graph, identical system specification for all the experiments and diversity of graphs under examination. From the figure shown above, most of the information can be obtained directly, such as the speedup of execution time of single thread BFS (when the NumThds is set to one) and parallelized BFS with different number of threads. Apparently, for both of the graphs, before the number of threads exceeds 16, which is the number of system threads, the speedup grows steadily with the increase of number of threads. This is because as the number of threads continues to grow, more nodes can be arranged to be traversed at the same time so that total time cost to visit all nodes can consequently be decreased. Moreover, when the number of threads is set too small, if a large number of tasks/requests need to be processed at the same time, the large number of requests/tasks may queue up in the task queue waiting for execution, or even the task/request cannot be processed for the fact that the task queue is full. All the problems occurs on account that CPU is simply not fully utilized.

On the other hand, for both of the graphs, when the number of threads exceeds 16, the value of speedup seems to be stable with slight fluctuation and almost no increase. This is likely to result from the dilemma that when the number of threads is configured too large, the threads will compete for CPU resources for system threads, which will cause frequent context switch.

4. Future Work and Further Exploration

In addition to the experiments and analyses based on the dataset, there are also some related work or improvements to be accomplished. First of all, the reliability of our results can be reinforced by enlarging the size of the dataset and examining the graphs through multiple machines. Considering the method used in parallel BFS, parallelization of the method next_ frontier. extend (local_ next_ frontier) are supposed to be parallelized to achieve higher performance. Moreover, data partition and potential load-balancing-related issues are absolutely able to be added and further analysing.

Among all the possible directions of future work, current parallel implementation of BFS suggests a promising avenue for further exploration: extending the parallel BFS algorithm to dynamic graphs. This would involve accommodating graph structures that evolve over time, which is the most real-like graph and of high value of application, presenting new challenges and opportunities for optimization.

5. Conclusion

This study presents an enhanced parallel Breadth-First Search algorithm designed for the adept traversal of expansive graphs on shared-memory platforms. By delving deeply into the intricacies of the standard BFS algorithm, the research crafts innovative methodologies for parallelization, deftly distributing tasks among multiple threads. Experimental assessments on the chosen dataset reveal a noteworthy observation: as the thread count escalates, there's a corresponding surge in CPU utilization, leading to an increased speedup, provided the count remains within the system's thread limit. Conversely, when the thread count surpasses the system's threshold, the speedup experiences marginal shifts, likely due to the frequent context-switching among threads. From a methodological vantage point, the research undertakes a tri-fold approach: meticulous algorithm examination, strategic optimization, and practical integration. The consistent experimental findings underscore the parallel BFS algorithm's edge over its sequential counterpart, especially when harnessing the full potential of system threads. Such statistical evaluations shine a light on the marked performance variance between the two techniques.

To encapsulate, this study unveils a skillfully optimized parallel BFS algorithm, underpinned by rigorous experimentation and discerning analysis. It offers profound insights, serving as a goldmine for professionals navigating the spheres of data-centric applications, graph analytics, and parallel computation, especially within the parallel graph traversal domain. The anticipated exploration of BFS on evolving graphs adds a promising facet to future research trajectories.

DOI: 10.54254/2755-2721/36/20230443

References

[1] Haley J, Shaia B, Subrahmanyam J, et al. A Breadth-First, Ordered Parallel Tree Traversal [J].
[2] Tödling D, Winter M, Steinberger M. Breadth-first search on dynamic graphs using dynamic parallelism on the gpu[C]//2019 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2019: 1-7.
[3] Tödling D. Breadth-First Search using Dynamic Parallelism on the GPU[J].
[4] Geil A, Porumbescu S D, Owens J D. Maximum Clique Enumeration on the GPU[C]//2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2023: 234-244.
[5] Chen Y, Brock B, Porumbescu S, et al. Atos: A task-parallel GPU scheduler for graph analytics[C]//Proceedings of the 51st International Conference on Parallel Processing. 2022: 1-11.
[6] Dong X, Wang L, Gu Y, et al. Provably Fast and Space-Efficient Parallel Biconnectivity[C]//Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 2023: 52-65.
[7] Gu Y, Napier Z, Sun Y, et al. Parallel cover trees and their applications[C]//Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures. 2022: 259-272.
[8] Ding X, Dong X, Gu Y, et al. Efficient Parallel Output-Sensitive Edit Distance[J]. arXiv preprint arXiv:2306.17461, 2023.
[9] Orenes-Vera M, Tureci E, Wentzlaf D, et al. Massive Data-Centric Parallelism in the Chiplet Era[J]. arXiv preprint arXiv:2304.09389, 2023.
[10] Kaler T, Schardl T B, Xie B, et al. PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation∗[C]//Symposium on Algorithmic Principles of Computer Systems (APOCS). Society for Industrial and Applied Mathematics, 2021: 144-158.