Accelerating Breadth-First Search

EC527: High Performance Programming

Po Hao Chen, Carlton Knox

March 27, 2025

Abstract

This report is submitted as the final project for EC527. We explore different implementations for one of the fundamental building blocks of classic graph algorithms - BFS; Accelerated with pThread, OpenMP, and MPI, we demonstrated the performance increase against generic version in sample problems. In addition, we analyze the scaling of our implementation on a High-Performance Computing (HPC) cluster.

Contents

Chapter 1: Introduction

The *Breadth-First Search* (BFS) algorithm is one of the basic and essential systematic approach of traversing a graph data structure. In contrast to another classical method, *Depth-First Search* (DFS) which iteratively follows an arbitrary node and backtracks. BFS explores its neighbors in "rings" and level-ordered manner.

Breadth-First guarantees the shortest path on an unweighted graph; it has applications in computing optimal solutions for games with least number of moves, finding connected components, shortest cycle, and more. In short, it is the basis for numerous advanced graph.

In scientific computing, graph-based computation is ubiquitous and often requires highly efficient methods of processing the large-scale operations. We espouse to study the parallelization of the BFS algorithm in subsequent chapters by presenting different approaches through multi-processors and Message Passing Interface (MPI) and evaluate their performances in a distributed setting.

Preliminaries

Traditionally, graphs are represented in the format of adjacency matrix and adjacency list. The former can potentially be useful in matrix-multiply based graph search algorithm involving stochastic matrices and Graphics Processing Units (GPUs). Our implementations chose to implement the adjacency list because of it's known efficient structure. Asymptotically, the matrix has a $O(|V|^2)$ search time and space complexity while the list is $O(|V| + |E|)$.

To accurately benchmark the scalability and efficiency of our algorithms, We decided to generate the Kronecker graphs which have properties similar to real world network.

Kronecker Graphs

Expander networks such as the internet grows over time and the distance between each nodes slowly decreases. Running experiments on real data can lead to new empirical findings which allows us to create new network models, vice versa. Inspired by the real network models, Kronecker graphs are [LKF05] generated to contain the properties and patterns in these models.

The graph generation compute the *Kronecker products* of two matrices every iteration until a desirable size is reached. With each multiplication, the size exponentially increases. Precisely, G_k has N_1^k nodes and E_1^k edges and we get densification as a result. definition

Definition 0.0.1 (Kroncker products) *Given matrix $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times q}$.*

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}N \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} \in \mathbb{R}^{pm\times qn}$$

We used the generator provided by Graph500 which allows flexibility in choosing the number of nodes in powers of 2 and the degree d of each node. In this work, we will refer to the level of difficulty as $log_2(n)$ where n is the total number of nodes.

Graph500 Benchmarks

Graph500 is a standard used to benchmark and rate state-of-arts supercomputers. Popular HPC benchmarks such as HPL measures double-precision floating point. However, these systems are designed to deal with much more complex problems which raw performance metrics do not take account into. Graph500 focuses on the capability to handle data-intensive workload through the communication subsystem of the clusters. The implementation is based on performing Breadth-First Search on large-scale Kronecker graphs. Officially, there are six possible scales: toy (226 vertices; 17 GB of RAM), mini (229; 137 GB), small (232; 1.1 TB), medium (236; 17.6 TB), large (239; 140 TB), and huge (242; 1.1 PB of RAM). [Wik22] However, it is possible to customize the sizes and number of edges.

The benchmark involves three computation kernels. First, it generates the graphs and compresses them into sparse structures (Compressed Sparse Row/Column). Second, randomly samples 64 sources and run parallel BFS search. Third, a single-source shortest path is computed. We will further explore the scalability and efficiency of BFS with this benchmark in Chapter 5.

Randomly Generated Graphs

In addition to the generator provided by Graph500, we also implement and use a simple random graph generator function for our tests. This generator can quickly generate an adjacency list for a graph given a desired number of nodes, minimum and maximum number of neighbors, and a random seed for reproducibility. Using this random generator for some of our testing allowed us to vary specific graph properties such as size and degree of connection.

Breadth-First Search (BFS)

Given a graph $G(V, E)$, BFS algorithms runs in $O(|V| + |E|)$ time on an adjacency list. The algorithm can be understood as an expansion of the source node in the directions of all of its neighbors. With every iteration, a ring-like pattern is increased by one unit.

The idea is to maintain a queue containing the verticies to be processed and we have a boolean structure that allows us to check whether the node has been visited. At initialization, we push the source node onto the queue to begin our traversal. We push all the neighbors to the queue in arbitrary order and pop the first vertex at the front of the queue to repeat the process. Assume the algorithm is synchronize, we can guarantee shortest distance traversal between the source and any other node. The pseudo-code as follows:

Algorithm 1 Breadth-First Search (BFS)

```
Require: $G(V, E)$
  $q \leftarrow$ Queue
  $q$.push(source)
  while q $\neq \emptyset$ do

    FrontOfQueue $= v =$ q.front()
    q.pop()
    for all neighbors $u$ of $v$ do
        if not visited then
            $q$.push($u$)
            mark $u$ as visited
```

Observe that since the neighboring nodes can be pushed in arbitrary order, there is a potential for paralellelization. In this work, we investigate different methods we can implement to speed up the graph search and compare their performances.

Chapter 2: OpenMP

The easiest way to parallelize the *Breadth-First Search* algorithm is by taking advantage of the natural ring-level parallelism of BFS. Since the set of adjacent nodes comprising each ring can be traversed independently in any order, we can perform the traversal of each ring in parallel, while maintaining BFS's ring level order. In this chapter, we describe our parallel BFS algorithm and implementation through the Multithreading API, OpenMP.

Parallel Breadth First Search Algorithm

As we described, the parallel BFS algorithm take advantage of ring-level parallelism to traverse each ring via multiple threads. For each ring-level, the nodes can be visited independently, so the available nodes can simply be divided among the available threads or CPU cores. However, each thread cannot simply add the visited threads neighbors to a shared queue, because it would cause a race condition. Since queues are inherently serial, only one element can be safely added at a time without causing errors. Forcing each thread to wait in line to push the next neighbors to the queue is possible through semaphores or mutual exclusion locks, but it would greatly decrease the parallel efficiency of the algorithm, potentially resulting in worse performance than the serial BFS code.

Our solution to this problem is to simply assign each thread their own private queue to push neighbors onto, which avoids any potential race conditions. Now, each thread can perform BFS on their assigned nodes in parallel. After the threads have finished exploring the current ring-level, all of the private queues are merged together to form the next ring to traverse. The parallel BFS algorithm continues like this, dividing the ring among the threads, and merging the resulting private queues back together, until all the connected nodes have been explored and the global ring queue ends up empty.

Implementation in OpenMP

For the OpenMP implementation of Parallel BFS, we first describe the inputs to the BFS function. First, the function takes as an input the graph adjacency list,a vector of vectors of nodes, and the origin node to search the graph from. Specific to our parallel BFS functions, the desired number of CPU threads is also taken as an input. Lastly, our all of BFS implementations include an action, a pointer to a function which acts on the visited node, as an input so that any of our graph traversal algorithms can be integrated with other algorithms built on top of BFS. Action functions can be combined with global data structures to perform complex tasks without having to change our BFS implementations.

Next, we overview the data structures used. First, to keep track of which nodes have been visited, we use a vector of booleans to represent a bitmap. For the global queue itself, we actually impliment it using a std::set, and a std::vector. Since we check for duplicate neighbors added to private queues in the merge step, using a set makes sure that each node only gets visited once, even if it is adjacent to multiple nodes in a given ring. Next, when each ring is traversed in parallel, the ring set is converted into a vector because OpenMP requires a data structure with elements that can be accessed in O(1) time so each element can be accessed in parallel.

Algorithm 2 Parallel Breadth-First Search with OpenMP (BFS$_omp$)

```
Require: $G(V,E)$, $origin$,N$_{THREADS}$, $action(node)$
  $q \leftarrow$ Queue
  $seen \leftarrow$ vector<bool>$\leftarrow false$
  $q$.push(source)

  while q $\neq \emptyset$ do
   initialize and assign $N_{THREADS}$ private queues $tq \leftarrow \emptyset$

   #pragma omp parallel for
   for all nodes $node$ in $q$ do
       if $action$ $\neq$NULL then
           action(node)
       end if
       for all neighbors $u$ of $node$ do
           if $u$ not seen then
               $tq$.push($u$)
               mark $u$ as seen
           end if
       end for
   end for
    $q \leftarrow \emptyset$
   for each private queue $tq$ of N$_{Threads}$ do
       merge $q \leftarrow tq$
   end for
end while
return =0
```

Lastly, we describe how the data structures are used to implement the parallel BFS algorithm. When the BFS function is called, it begins with an empty global queue(set), into which the origin is inserted. Then, while the global queue is not empty, the function performs BFS on each ring. First, empty private queues are allocated for each thread. Then, the global queue is converted into a vector, which is divided into the available threads using the OpenMP preprocessor directive "pragma omp parallel for". Then, in parallel, the action, if defined, is performed on each node in the ring, and then that node's neighbors are explored. Those neighbors are added to each thread's private queue. finally, the global queue is emptied, and replaced by the merged private thread queues, and the algorithm repeats this until the global queue remains empty.

Chapter 3: Load-Balancing Multi-thread Graph Traversal with pthreads

In this section, we describe an alternate parallel graph traversal algorithm using PThreads (POSIX Threads). Unlike the OpenMP version of BFS which takes advantage of ring-level parallelism to maintain ring-order when traversing through the graph, this algorithm ignores the ring-order hierarchy of the graph in favor of traversing through discovered neighbors as quickly as possible.We accomplish this by designating one manager thread, and several worker threads. Each worker thread is allocated a subset of the global queue, and allowed to traverse the graph independently, adding new neighbors to their own local queue as they go. Since this method does not include any synchronization between the worker threads in regards to the ring-order, the whole graph can be explored in a non-BFS order.

Mutual Exclusion

Although the worker threads do not need to be synchronized in terms of ring-order, we do still need to ensure that each graph node is only visited once. Simply using a seen vector is not sufficient to ensure mutual exclusion, as the multi-threaded algorithm results in a race condition between checking if a node has been seen, and setting it's seen value as true. In other words, it is possible for two or more parallel threads to add a node to their private queue simultaneously, before any of them have labeled that node as seen. The way we avoid this race condition is by introducing an array of mutual exclusion locks (mutex), each corresponding to one of the graph's nodes. In the POSIX Threads library, a mutex is a thread-safe lock that can only be acquired atomically by a single thread. So, assigning a mutex to each node ensures that each node can only be added to any queue a single time. Then, when the next thread attempts to take the mutex for a node that has already been taken, it is forced to give up and try the next node in the adjacency list. As a result, each node can only be enqueued a single time.

Load Balancing

When our pthreads algorithm begins, it starts by performing serial BFS until there are enough nodes in the global queue that can be divided into the number of available threads.

Then, the worker threads are created, and continue performing BFS on their local queues until the graph has been fully traversed. Although each worker thread begins with at least one node, there is no guarantee that all the worker threads will remain busy for the entire run time of the algorithm. In fact, it is likely that some threads will run out of nodes to explore early on in the algorithm's run time. This imbalanced work load is impossible to predict at the initialization of the worker threads, and can result in low efficiency and vastly reduced performance. So, our proposed solution to this problem is to include a load-balancing parallel graph traversal algorithm.

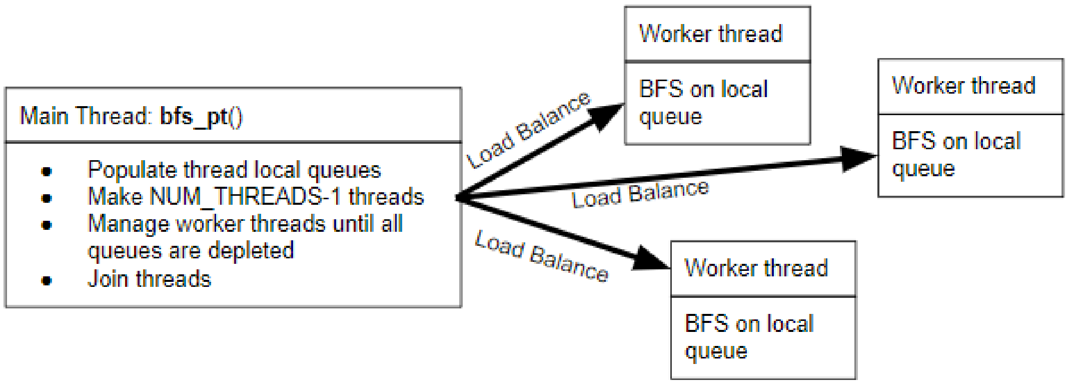


Figure 1 High level diagram of PThreads-based Load-Balanced Parallel Graph Traversal Algorithm

As described before, this algorithm begins by performing serial BFS to fill out a global starting queue that can be divided into each thread's local queue. Then, the algorithm creates $N_{THREADS} - 1$ worker threads, each with their own private queue. However, now the main thread is designated as the "manager" thread, which has the function of monitoring the loads of each of the worker threads, and redistributing nodes to threads that have depleted their queue and need more work. In order to achieve this, we introduce a second array of mutual exclusion locks, where each mutex corresponds to one of the worker threads. Now, Whenever a worker thread accesses its private queue, it must acquire it's own corresponding mutex first. If the manager thread acquires the mutex first, then the worker thread will wait until the manager thread releases it's lock before proceeding. If the manager thread tries to acquire a worker lock that the worker thread already owns, then the worker thread will pause after it has finished it's current iteration of BFS. Now, with an array of worker thread locks, the manager thread can monitor the worker threads and detects and counts threads that have entered a "waiting" state. Then, if the number of "waiting" threads is less than the number of worker threads, but greater than 0, the manager will attempt to lock one of the non-waiting threads. If the non-waiting thread still has more than one element in it's queue when the manager locks it, it will distribute that thread's queue's nodes to the waiting threads. The manager will distribute the load, one node at a time until either all the waiting threads have work to do, or there are no more threads that can donate work. Then, the manager unlocks the donor, and sets the waiting threads to a non-waiting state. Once a worker thread is no longer waiting, it continues performing BFS on it's local queue. The manager continues checking for waiting threads and condtionally load balancing until the number of waiting threads equals the number of worker threads. This means that there is no more work to be done, and the graph has been fully traversed.

Finally, the manager thread sets a flag for all of the worker threads, allowing them all to exit, and join back to the main thread.

Chapter 4: Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standard designed to be the de-facto programming model for scientific parallel computations. It is essential in High Peformance Computing (HPC) applications, the protocal offers synchronization, communication and virtual topology between a set of processes.

Programming model

By convention, a MPI program is written in sequential programming language and commonly computes on a single source of data. (SPMD) We assign consective *ranks* to each process in the topology and exchange information between them across the *communicator*. The communicator is an argument to MPI subroutines, and it is used to send (MPI_Send) and receive (MPI_Recv) data with the respective source rank and destination rank.

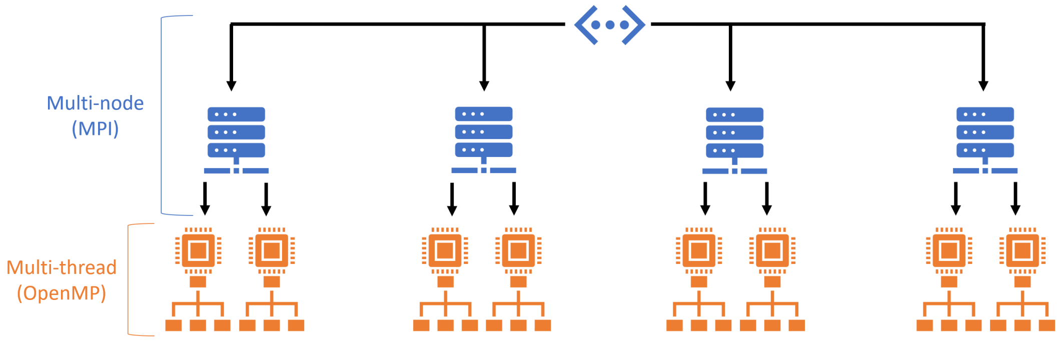


Figure 2 Topology of MPI communication

Our experiment

There are various implementations available (OpenMPI, IntelMPI, MVAPICH), however, we will not focus on the details of the programming model in this chapter. In general, MPI incurs a larger overhead than the methods we discussed before. We are interested in the performance of MPI-based BFS implementation because it will be needed in high-scale graphs with billions of verticies for inter-node communication.

As highlighted in the introductory chapter, Graph500 offers a referenced BFS algorithm implemented with MPI. Their algorithm follows closely with the Parallel Breadth-Firt Search that maintains a local and global queue we discussed in Chapter 2 with adjustment to the corresponding MPI directives. We perform a simple level-synchronized BFS and transfer the information of nodes visited with MPI_Send and MPI_Recv. We take advantage the zero-length message as an implicit barrier and use MPI_AllReduce to count the number of elements in the local queue and merge onto the global queue when synchronization is met.

In our experiment, we fixed the graph to be a 16-regular (degree) graph with 2^{20} verticies and tested the implementation on 16/32/64/128 processes or 2/4/8/16 compute nodes.

Chapter 5: Results

In this chapter, we analyze the performances of methods from previous chapters to determine their scalability and efficiency.

Benchmarking Serial BFS, Parallel OpenMP BFS, and PThreads Parallel Graph Traversal

First, we benchmark our graph traversal algorithms on the Graph500 graph. To compare the performance of these graph traversal algorithms, we measure the time taken to traverse the same graph, starting from the same origin node, and with the $action(node)$ inputs set to $NULL$. Additionally, these tests were all run on a 32-core Intel Xeon Gold 6242 CPU on the Shared Computing Cluster with $N_{THREADS}$ set to 32.

Table 1 Results comparing graph traversal time for 16-regular 2^{20} node graph generated by Graph500 benchmark

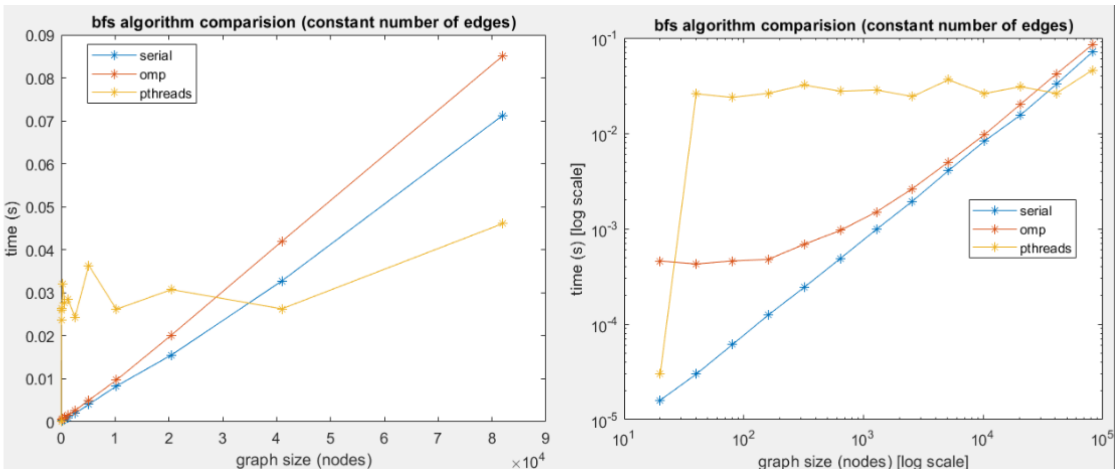
BFS Algorithm	Serial BFS	Parallel BFS with OpenMP	Parallel Load-Balanced Graph Traversal with PThreads
Time (s)	1.119076	0.943081	0.38678

As seen in Table 1, the Parallel Load-Balanced Graph Traversal Algorithm provides 2.9x speedup over the serial algorithm for this graph, while the OpenMP code provides a 1.2x speedup. This shows that both parallel algorithms are faster than the serial code for this graph. However, the speedup for OpenMP is rather low, especially when the code has access to 32x the CPU cores.

The reason for this unexpected result is that the OpenMP algorithm inherently benefits from a high degree of ring-level parallelism, while the PThreads algorithm ignores this ring-order structure, and traverses the graph in any parallel order.

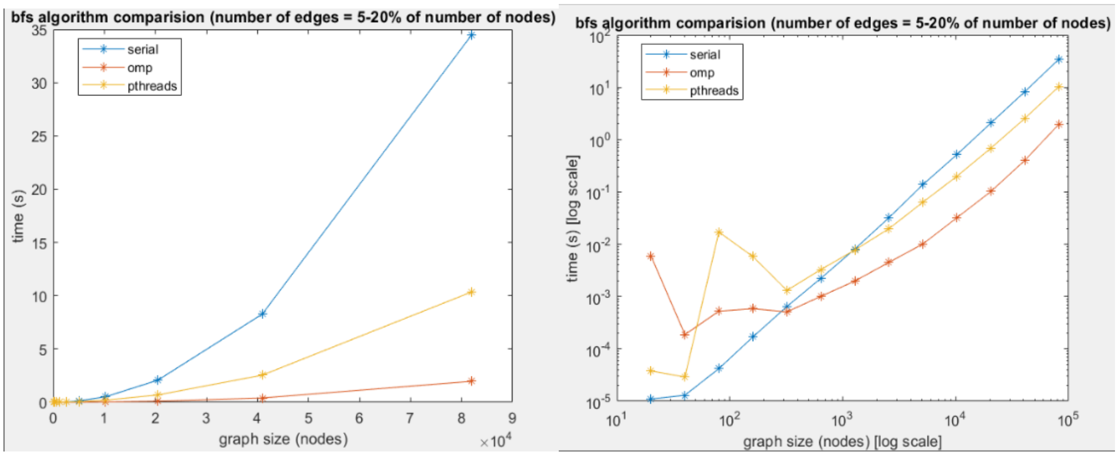
To test this, we run two additional benchmarks, this time using our own graph generator. First, we randomly generate graphs of increasing size, and a constant connectivity percentage range of 5% - 20%, such that for each graph size, each node is connected to between 5% and 20% of the nodes. The results can be seen in Figure 4.

Next, to test graphs with low degrees of connectivity, we randomly generate graphs of increasing size, but a constant degree, with each node having between 10 and 15 edges. The results can be seen in Figure 3.



(a) linear scale **(b)** log scale

Figure 3 Graph Traversal time for randomly generated graphs of increasing size, but constant degree range



(a) linear scale **(b)** log scale

Figure 4 Graph Traversal time for randomly generated graphs of increasing size, and increasing degree

As shown in our experiments, The pthreads algorithm provides high speedup for significantly large graphs with low degrees of connectivity. Meanwhile, the OpenMP code can actually perform worse than the serial code for such graphs. On the other hand, for graphs with very high degrees of connectivity, the OpenMP code is able to outperform both the serial and pthreads code by a lot. Overall, the most efficient graph traversal algorithm depends on both the size and degree of the graph being traversed.

Table 2 Time for 16-regular 2^{20} graph with MPI implementation from Graph500

procs	nodes	sec
128	8	0.0209307
64	4	0.0308374
32	2	0.0586262
16	1	0.0688323

MPI on Graph500

Table 2 demonstrated the performances scales sublinearly with the number of nodes. Although we are limited to the number of available cores on our cluster to have a definite conclusion, this result indicates good scalability characteristics. Modern High-Performance Computing (HPC) clusters adopts a scale-out design, and our parallel BFS algorithm is largely benefited which speeds up the array of scientific applications they are implemented in. Theoretical estimate such as Amdahl's Law does not take parallel overhead into account. We must be aware of the communication bottleneck present in the program. Consider that our routine is not optimized, we can remove some of the overheads in the send-recv communication to further increase the efficiency. Bottom line here is that if our problem size is smaller, we shouldn't use MPI.

Conclusion

We remark our thoughts on this study of Breadth-First Search here. We believe that it is crucial to understand the scale of the input that the application is designed to handle. Our implementations of BFS has their respective advantages and disadvantages. Parallel Graph Traversal algorithms that respect the ring-level order of BFS work best in graphs with a high degree of ring-level parallelism, while methods like our Load-Balanced Parallel Graph Traversal algorithm may be more flexible. If properties of the graph's topology is known beforehand, the most efficient algorithm can be selected accordingly. In practice, we expect to see a hybrid of the implementations used in practice. For example, MPI communication can be used to exchange information of each compute node running parallel BFS with OpenMP directives. We hope to study and evaluate the algorithm at even larger cales in the future and consider it in other application scenarios in the future.

Bibliography

[LKF05] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations”. In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining.* KDD ’05. Chicago, Illinois, USA: Association for Computing Machinery, 2005, pp. 177–187. ISBN: 159593135X. DOI: `10.1145/1081870.1081893`. URL: `https://doi.org/10.1145/1081870.1081893`.

[Wik22] Wikipedia. *Graph500 — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Graph500&oldid=1030684201`. [Online; accessed 05-May-2022]. 2022.

Source Code

All original source code referred to in this paper can be found at:
https://github.com/du00d/ParallelFirstSearch