Parallel Breadth First Search for Scale-Free Social Network Graphs

James Ihrig
University of Central Florida

Josh Eberst
University of Central Florida

Johnathan Pecoraro
University of Central Florida

***Abstract*—Graphical representations are one of the most commonly used abstractions to model large networks. Social media networks like Facebook and Twitter contain massive scale free networks with billions of nodes. The scale free nature of the connections between nodes presents several challenges to the performance of traditional searching algorithms. In this paper, we present three implementations of Parallel Breadth First Search algorithms on scale free networks without the use of specialized hardware.**

I. Introduction

Graph abstractions are commonly used in large scale network analysis [6]. With the proliferation of "Big Data" applications, parallelized versions of common graph analysis algorithms have a very high demand.

Many graph analysis problems make use of Breadth First Search (BFS), as the efficiency of the algorithm scales linearly with the number of nodes and edges.

Parallelizing Breadth First Search algorithms offers an excellent opportunity to increase the efficiency of this common graph analysis technique. However, scale free networks such as those created by social media connections are challenging to parallelize efficiently as the high distribution of edges to a few nodes is difficult to balance in memory [2]. Our research is targeted at the implementation of three variations of Parallel Breadth First Search algorithms without the use of specialized hardware on scale free networks. The first algorithm, Parallel Breadth First Search with Partitioning (PPBFS), was originally designed to improve BFS performance in distributed memory architectures used by many modern supercomputers. Our implementation will specifically apply applied this solution to scale free networks, which was left as future work by the original authors. However, as we are did not implementing specialized hardware, our research will was be performed without the use of a distributed memory architecture.

The second algorithm we will implement, Multithreaded Breadth First Search (MTBFS), is designed to support fine-grained, low overhead synchronization in a massively multithreaded system [6]. This algorithm will be used for gauging the performance of PPBFS, as MTBFS was intended to support scale free graphs. Since the focus of MTBFS is on increased parallelism through multi threading, the differences between the Cray MTA-2 system MTBFS was designed for and a standard multi-core CPU should be less drastic than the shift from the distributed memory architecture used by PPBFS. Memory access by BFS and other graph algorithms is typically fine-grained and irregular. This leads to poor cache performance, especially in parallel versions of BFS as parallelization relies heavily on the cache performance. Some performance improvements can be made, but they can not be sufficiently generalized since cache performance depends largely on the structure of the graph [6].

The final algorithm we implemented is a level synchronous breadth-first search. This implementation of this algorithm forces the threads to process the graph iteratively during the breadth first search. The iterative nature of this algorithm ensures the same node is not visited multiple times. The paper uses this algorithm as part of their hybrid breadth-first search solution.

We will compare all of these algorithms with the sequential version of BFS across two datasets. One dataset will be a sufficiently large import of a Facebook social network, and the other a generic dataset based on internet architecture.

II. Related Work

A Scalable Distributed Parallel Breadth First Search Algorithm on BlueGene/L[6]

This paper talks about searching networks that are too large to fit into memory of a single machine. To handle this, it divides the graph into partitions where each node processes a set of vertices assigned to it. If it finds a vertex that does not belong to it, the node that owns it is notified. One major drawback this approach seems to have is that it waits for all processes to reach the same level before moving on. While this is important to truly follow a breadth first pattern, it may not be necessary to limit it in this way.

Designing Multithreaded Algorithms for Breath First search and st-connectivity on the Cray MTA-2[2]

The Multithreaded BFS algorithm implemented by Bader and Madduri will serve as our baseline parallel BFS algorithm. Although initially implemented on a massively multithreaded shared memory system without a data cache, we believe the fine-grained parallelization of this algorithm will still perform efficiently in a single CPU multi-core system. We have chosen this algorithm to serve as baseline parallelized implementation because it was specifically designed for the traversal of large scale-free graphs similar to those used in our benchmarks.

Parallel Breadth First Search on Distributed Memory Systems[3]

The partitioning Parallel Breadth First Search algorithm for distributed memory systems implemented by Buluc and Madduri extends the research performed by Yoo on the Scalable Distributed Parallel Breadth First Search, and highlights inefficiencies in the supporting structure of the other research. This research provided prior work summaries and criticism on the two algorithms we will adopt, but implementing the improvements made here is outside the scope of our research.

Efficient Parallel Graph Exploration on Multi-Core CPU and GPU[4]

The breadth-first search algorithm presented in this paper was proposed by Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Their algorithm uses a hybrid approach to dynamically determine the best search method to use. They combine a level synchronous approach with a partitioning approach. For their combined approach, they determine whether to use a queue based search or an array based search based on criteria observed from the graph.

III. Technical Approach

Our projects technical approach focuses on comparing the graph traversal time of the three versions of the BFS algorithm. The original implementations of our parallel BFS algorithms were intended for use on super-computing hardware. The reason we have implemented multiple parallel BFS algorithms is to allow for additional performance comparisons between the algorithms, since they are abstracted away from their specialized hardware. Additionally all of our parallel algorithms will have their performance compared with a traditional sequential implementation of BFS.

The performance comparison of the algorithms will be determined by the amount of time each algorithm takes to fully traverse every node in two scale-free graphing benchmarks. We have implemented our solution using a third party application, NodeXL[5], to generate large scale-free graphs within our application. The NodeXL program is an open source application written in the C# language. In general, it is used as a network visualization tool which gives users the ability to view a graph's connectivity on a graphical interface. Additionally, there is a social network plugin for the NodeXL application which provides the capability of importing a social network graph from Facebook. We have utilized the class libraries for NodeXL and the associated social network plugin within our application to assist with the generation of the graphs.

Our first benchmark will be based on a graph of a sufficiently large social network imported through the use of the Facebook social network plugin for NodeXL. Our second benchmark is a based on an internet network dataset with over 124,000 nodes and 200,000 edges. Both of these datasets will be imported using NodeXL into a graph consumable by the algorithms. These benchmarks should both be sufficiently large to incur the cache performance challenges associated with many parallel BFS solutions.

We had intended to include the generic scale-free graph generated by the social network data generator GDBench [1]. The GDBench tool is capable of generating scale-free graphs with millions of nodes, and saving it in a common GraphML format, but the graphs generated by the tool were significantly disjoint. This made inclusion of the benchmark difficult as over half the nodes were unreachable.

IV. Technical Details

All of the algorithms we have implemented were originally implemented on specialized supercomputing hardware. A difference in our approach is to implement these algorithms on a single CPU with multiple cores. Two of the algorithms, MTBFS and LSBFS will be implemented as closely as possible to their original design on the super computing hardware. The PPBFS algorithm will be significanly modified from its original design for distributed computing hardware.

Parallel With Partitioning BFS (PPBFS)

The original algorithm used as a reference for our project was implemented as a distributed BFS on the BlueGene/L architecture. The algorithm takes advantage of the architecture's structure to develop efficient inter-processor communication, which is generally the bottleneck in distributed systems. Our implementation uses the techniques described in the paper to develop our parallel version of the algorithm, but with modifications that make more sense for working on a shared memory machine.

The primary difference between our implementation and the method used by the paper is how it transmits neighbor nodes to other threads. While a thread is executing the breadth first search, it may come across nodes that it does not own when searching adjacent nodes. The original paper solves this problem by having each thread create lists of neighboring nodes that belong to other processes and send them over a network to their respective processes and wait to receive lists from other processes. This sending and receiving of node lists forces the processes to need to wait for each other before descending into the next level.

Our implementation uses the same general methodology of partitioning the graph, but without the overhead of a network to perform on, we decided that creating queues for each thread, and enqueueing nodes as they were found made a lot more sense. This way each thread does not have to wait for all other threads to finish before searching the next depth level. The tradeoff is that the path found may not be the shortest possible path to a node and the path found will depend on the scheduler.

The algorithm starts by partitioning the graph equally between the available threads. All of the vertices will be assigned an ID indicating which thread is responsible for searching it. One node will be marked as depth zero. This marks the starting node for the search, and is enqueued into the queue of its owner. Each thread takes the partitioned graph as input of size n/p. (Where 'n' is the number of vertices in the graph, and 'p' is the number of threads.) The algorithm works by

Algorithm 1 Distributed Breadth First expansion with 1D Partitioning[6]

1) Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{where } v_s \text{ is a source} \\ \infty & \text{otherwise} \end{cases}$
2) **for** $l = 0$ *to* ∞ **do**
3) $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
4) **if** $F = \emptyset$ for all processors **then**
5) Terminate main loop
6) **end if**
7) $N \leftarrow$ {neighbors of vertices in F (not necessarily local)}
8) **for all** processors q **do**
9) $N_q \leftarrow$ {vertices in N owned by processor q}
10) **Send** N_q to processor q
11) **Receive** $\bar{N}_q$ from processor q
12) **end for**
13) $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
14) **for** $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
15) $L_{v_s}(v) \leftarrow l + 1$
16) **end for**
17) **end for**

Algorithm 2 Modified Breadth First search with 1D Partitoning

```
subgraph<Vertex>[numVerts/numThreads]
subgraph[k].level = 0
localQueue[numThreads]
bfs(subGraph, threadID)
    finishMask |= 1 << threadID
    for Vertex v in subgraph
        v.id = threadID
        if (v.level = 0)
            localQueue[v.id] = threadID
    while ( finishMask != currentMask )
        while ( localQueue[threadID].size > 0 )
            srcVert = localQueue[threadID].dequeue()
            for Edge e in srcVert.edges()
                localQueue[e.dest.id].enqueue(e.dest)
                currentMask = 0
        currentMask |= 1 << threadID
```

dequeuing a vertex v_1, and enqueuing its neighbors $v'_1 \rightarrow v'_k$ into the local queues of the thread it belongs to. This process is repeated until all queues are empty. This modified version is fully illustrated as Algorithm 2.

Multi-threaded BFS (MTBFS)

The MTBFS implementation used for reference in this paper was implemented on a Cray MTA-2 multithreaded architecture [2](Reference paper 2). Similar to our approach, the algorithm was tested against scale-free graphs. When tested with a graph containing 400 million nodes, a 40 processor system showed the multi-threaded algorithm to have a system speedup of about 30 times over the sequential implementation.

The algorithm takes in a graph with source node s, and returns a shortest-path array d such that d[v] contains the length of the shortest path from source node s to destination node v, where v is a node in the graph. The algorithm uses the standard approach for implementing a sequential BFS where you start by adding the source node to a queue, then continue looping until the queue is empty. Each iteration

Algorithm 3 Level-synchronized Parallel BFS[2]

1) *for all_v* $\in$ *V in parallel do*
2) $d[v] \leftarrow -1$;
3) $d[s] \leftarrow 0$;
4) $Q \leftarrow \phi$;
5) *Enqueue* $s \leftarrow Q$;
6) *while* $Q \neq \phi$ *do*
7) *for all* $u \in Q$ *in parallel do*
8) *Delete* $u \leftarrow Q$;
9) *for each* v *adjacent to* u *in parallel do*
10) *if* $d[v] = -1$ *then*
11) $d[v] \leftarrow d[u] + 1$;
12) *Enqueue* $v \leftarrow Q$;

dequeues the first node, finds it's neighboring nodes, and adds all unvisited neighbors to the queue. The main difference is that looping through the queue and through the neighboring nodes is done in parallel. In detail, the algorithm works as shown in Algorithm 3.[2]

This algorithm takes full advantage of the Cray MTA-2 system's architecture by using its fine-grained parallelism and zero-overhead synchronization while looping through the queue and looping over each node's set of neighbors. This multi-threaded BFS technique implemented on the Cray MTA-2 hardware architecture design offers a considerable speedup advantage over any sequential implementation of BFS on graphs of similar size.[2]

MTBS has been implemented using nested Parallel.ForEach commands on methods that perform the functions highlighted above. Our implementation is significantly slower than the version implemented on the Cray MTA-2 as the nested parallelization limits the performance of the algorithm. The parallelization of the edges is responsible for most of the performance degradation as the workload of collecting adjacent edges is a relatively small.

Level Synchronous BFS (LSBFS)

The LSBFS algorithm uses level synchronization to keep the processing of nodes in order. A level property with an initial value of infinity is applied to each of the vertices in a graph. This property represents the distance a given vertex is away from root node. When the LSBFS starts, it sets the level of the root node to zero (0) and adds the root node to the list of visited nodes. Then for each iteration of the breadth first search, all nodes at the current level are searched in parallel. For each vertex in the current level, all of the vertex's neighbors are obtained and visited. For every visited vertex, the level property is set to the current level plus one (level + 1). After all the neighbors have been visited, the current level counter is incremented and all the neighboring nodes we just visited are the next set of nodes to continue searching on. All threads will execute this task in parallel.

The bottleneck in this algorithm comes from the fact that a finished thread must wait until all other threads finish their portion of the search at the current level. This is because

all threads operate on a single level at a time. A general observation is that the number of nodes increases at higher levels of the graph. The algorithm benefits from this property as the parallel threads have more nodes to execute on as the level increases. However, another drawback of this method is that the algorithm does not take full advantage of breadth-first search properties. It would benefit from a partitioning scheme that allowed for multiple threads to operate on subgraphs of connected vertices. According to the parallel exploration paper[4], the algorithm still performs well in real-world systems despite it's synchronization overhead.

V. Experimental Results

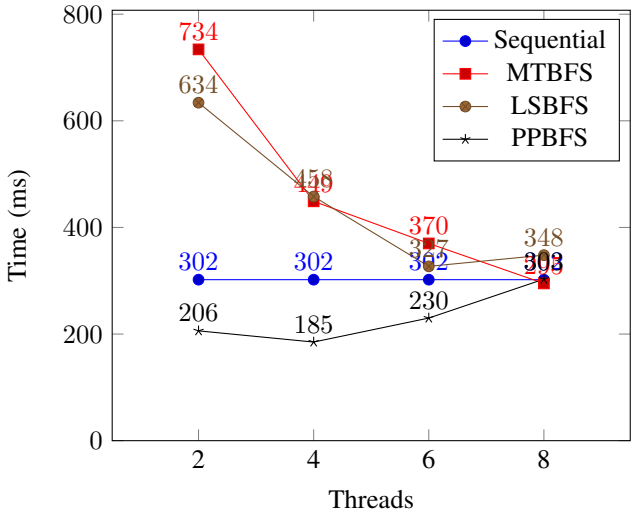
Our experiments were implemented on an Intel i7 system with 4 cores. Our first benchmark included an import of a FaceBook social network with 7737 nodes and 12,694 edges. The second benchmark is a scale free dataset based on internet architecture with 124,651 nodes and 207,214 edges. The experiments were run multiple times over 2, 4, 6, and 8 threads.

Our experimental results show that the PPBFS algorithm has a 62% improvement over the performance of the sequential algorithm when it is executed on the larger scale free graph with four threads. The algorithm was not able to run with two threads, and the execution time in this case is marked as zero in our results.

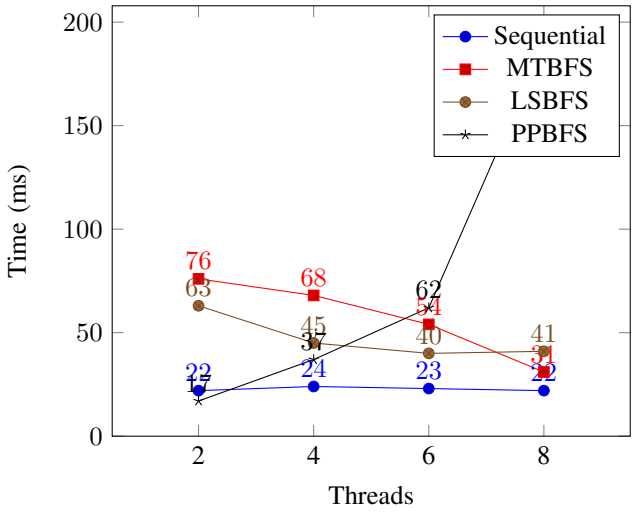
The PPBFS algorithm also does not perform nearly as well on the smaller facebook graph. In order to achieve the performance benefits of partitioning the graph, the number of nodes must be sufficiently high to amortize the penalty of the graph partitioning process.

The MTBFS and LSBFS algorithms do not perform as well as the sequential BFS algorithm, but the performance gap narrows as the number of threads increases. These algorithms performance results are based on the balancing of work among the threads, and both graphs show good performance until 8 threads. With 8 threads, MTBFS continues its performance improvement, but LSBFS's performance starts to decrease. This is likely because the overhead of the additional threads is no longer contributing to a balanced workload.

Scale-Free Graph



Facebook



VI. Conclusions

Our experimental results show that PPBFS algorithm can improve the performance of scale free graph traversals, with a sufficiently high number of nodes, and a balanced number of threads. The algorithm consistently has the best performance when it uses four threads, and if the dataset is large enough to encourage this type of parallel graph analysis it can perform very well. The MTBFS and LSBFS algorithms have generally lower performance than their sequential counterpart, but this was partially expected as it has been reported that scale free graphs don't benefit much from these simpler forms of parallelization.Our implementation of PPBFS outperforms the other algorithms we used as benchmarks of parallel BFS performance, making it a suitable choice for scale free graph analysis.

VII. Challenges

Many benchmark datasets contain graphs with disjoint sets of vertices. Disjoint graphs with BFS won't allow the algorithm to visit all the nodes. The GDBench tool that we

had intended to use for the production of substantially large datasets would produce disjoint graphs with roughly half the nodes in each set. This made the datasets produced nearly unusable as only half the graph could be traversed by the algorithms.

These datasets may have been usable, but the NodeXL architecture we implemented becomes sensitive to the amount of memory used as the number of nodes increases above 100,000.

VIII. Future Work

There are a few minor optimizations we would like to focus on for the future works of this project. First, we would like to compare the differences in performance between restricting the algorithm to visiting nodes on a level synchronous bases, versus running a level asynchronous version. The asynchronous version would have the possibility of visiting nodes multiple times, but it would eliminate the overhead of threads needing to synchronize their execution after each level. Another small optimization we would like to implement is attempting to use array based queues. The size of the array for the queue could be easily determined by observing the number of nodes rationed to each partition of the graph. This would give us the benefit of spatial locality over the current list based queue that is used.

We would also like to use larger graphs to test our algorithm. We believe the algorithm would benefit from larger graphs since the parallelism could take full advantage of the larger range of nodes. As we observed from our test results, sometimes it's beneficial to use a sequential approach over a parallel approach, and sometimes it's beneficial to use a partitioned approach over a full graph search. With this in mind, we would like to optimize our implementation by taking a hybrid approach to determine which algorithm would be best to use based on the graph provided and the capabilities of the system which is being used to run the search.

References

[1] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 15:1–15:7, New York, NY, USA, 2013. ACM.

[2] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[4] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88, Oct 2011.

[5] Marc A. Smith, Ben Shneiderman, Natasa Milic-Frayling, Eduarda Mendes Rodrigues, Vladimir Barash, Cody Dunne, Tony Capone, Adam Perer, and Eric Gleave. Analyzing (social media) networks with nodexl. In *Proceedings of the Fourth International Conference on Communities and Technologies*, C&T '09, pages 255–264, New York, NY, USA, 2009. ACM.

[6] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.