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 two 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 [4]. 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 [1]. Our research is targeted at the implementation of two.

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 this solution to scale free networks, which was left as future work by the original authors. However, as we are not implementing specialized hardware, our research will 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 multi-threaded system [4]. 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 [4].

We will compare both 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 scale free network generator GDBench [2].

II. Related Work

A Scalable Distributed Parallel Breadth First Search Algorithm on BlueGene/L[4]

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[1]

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[2]

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.

III. Technical Approach

Our projects technical approach will focus on comparing three versions of the BFS algorithm. The original implementations of our two parallel BFS algorithms were intended for use on super-computing hardware. The reason we have implemented two versions of parallel BFS algorithms is to allow for additional performance comparisons of the algorithms, since they are both abstracted away from their specialized hardware. Accordingly we believe that the PPBFS algorithm's performance will be the most dependent on the hardware, and the MTBFS algorithm will be more hardware agnostic. These parallel algorithms will both 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[3], 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 large 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 generic scale-free graph generated by the social network data generator GDBench. The GDBench tool is capable of generating scale-free graphs with millions of nodes, and saving it in a common GraphML format. The GraphML file generated by the benchmark will be imported using NodeXL into a graph consumable by the algorithms. These social networking benchmarks should both be sufficiently large to incur the cache performance challenges associated with many parallel BFS solutions.

IV. Technical Details

Our research is based on a distributed parallel approach to BFS, originally implemented on the IBM BlueGene/L supercomputer. We hope to attain performance increases using the same methods on a single CPU using multiple cores. Additionally, we will attempt to find optimizations to the algorithm such as allowing the processor to continue without waiting for send/receive messages before continuing the search for a given level/depth.

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.

Algorithm 1 Distributed Breadth First expansion with 1D Partitioning[4]

```
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 $\infty$ 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
```

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 for 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 simply dequeuing an vertex v_1, and enqueuing its neighbors $v'_1 \to v'_k$ 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.

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
```

Algorithm 3 Level-synchronized Parallel BFS[1]

```
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)                 $Enqueue\ v\ \leftarrow\ Q;$
```

Multi-threaded BFS (MTBFS)

The MTBFS implementation used for reference in this paper was implemented on a Cray MTA-2 multithreaded architecture (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 time 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 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 follows:

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 queue and looping over each node's set of neighbors. This multi-threaded BFS technique along with the hardware architecture design offer a considerable speedup advantage over any sequential implementation of BFS on graphs of similar size.

V. Challenges

We have noticed that the PPBFS algorithm[4] waits for all other threads to complete a search level before moving on to the next level. This places some sequential constraint on the algorithm and can potentially limit performance.

The benchmark files have been created for both Facebook and GDBench, but both benchmarks produce graphs with disjoint vertices. Disjoint graphs with BFS won't allow the algorithm to visit all the nodes in the graph. We are currently attempting to make the graphs larger while still maintaining connectivity.

Completed work:

1) Thus far we have successfully implemented and modified the NodeXL class libraries (used for scale-free graph creation) to support additional properties for BFS
2) Implemented a sequential BFS algorithm capable of running on a graph imported from Facebook with NodeXL.
3) We have also implemented a sequential version of the PPBFS Algorithm.

Remaining work:

1) Resolve disjoint sets in the benchmarks, so all vertices in the graph are connected.
2) Finish the parallelization modifications to the PPBFS Algorithm Implement the MTBFS Algorithm
3) Execute the algorithms and collect experimental results

References

[1] 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.

[2] 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.

[3] 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&#38;T '09, pages 255–264, New York, NY, USA, 2009. ACM.

[4] 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.