Parallel Breadth-First Search

Brooke Godinez

***Abstract*—**

Breadth First Search based algorithms are an important tool to solve graph related problems but as graphs grow, this runtime of BFS algorithms can grow exponentially. One way to distribute the work of traversing a graph is through making a parallel implementation. In this project a version of the parallel breadth first search algorithm is explored and optimizations to both its building blocks, the scan, pack, and filter and the BFS algorithm itself. Through these optimizations, improvements in speeds ups over the sequential version have been demonstrated.

I. Introduction

Breadth first search is a foundational algorithm in algorithm development, playing a large part in many more complex graph based algorithms and software systems. Parallel systems are increasingly becoming necessary as needed computational power pushes up against the hardware limits of modern computers. An optimized, parallel breadth first search algorithm is a powerful tool to increase the speed of graph traversal algorithms. There are some fundamental differences between a sequential breadth first search algorithm and a parallel version. The first major obstacle for translating a sequential algorithm to a parallel version is avoiding data races. A data race is characterized as an attempt by two different threads to write to the same variable or, more specifically, memory location. Due to the non deterministic nature of parallel execution, there is no guarantee when a thread will run, resulting in inaccurate reads of that memory location by other threads or later in the program. To guarantee that data races are avoided is essential to ensure that either every time a data location is accessed it is done outside of the parallel execution or the algorithm can be designed to avoid accessing the same data element while in parallel execution. However some algorithms will make this very difficult so there are some locks that can help regular access to specific memory locations, however, they slow execution time overall.
The Cilk library is used to create the parallel execution for this project. It is based on the C and C++ programming language and extends it with constructs to express parallel loops and the fork-join idiom. In general, for loops in the algorithm can be made parallel by replacing it with the Cilk version. The fork-join idiom is the idea that in parallel computing that execution is branched off in parallel and then joined at some other point within the program [2]. Within Cilk this is represented by calls to the $cilk\_spawn$ and $cilk\_sync$. Between the $cilk\_spawn$ and $cilk\_sync$ those lines of code are executed within different threads in parallel.

As discussed in the previous section a major obstacle in turning a sequential algorithm into a parallel algorithm is data races, requiring that while in a parallel execution the program does not access the same memory location. This results in issues with a standard breadth first search algorithm, which uses a queue to explore a graph, keeping a current node variable and pushing the discovered nodes in the graph to a queue data structure and popping them off the front of the queue when a breadth is fully explored. This can result in conflicts in the parallel environment. Two different threads may attempt to pop the front element from the queue at the same time, which can result in incorrect exploration of the graph if a thread that is exploring farther into the graph is able to set the parent of a node incorrectly.

As a result, another data structure that allows for parallel access and another method of holding which elements have been explored.

II. Literature Review

The breadth first search algorithm is an important field of study and increasing the execution time for growing numbers of nodes and edges are important for many different types of common graph-theoretic problems, and main goal of speeding up these algorithms is too speed up the underlying graph problems with parallel execution [5]. A paper from the Lawrence Berkeley National Laboratory outlines two difference implementations of parallel BFS algorithms. Firstly a vertex-based partitioning of the graph uses distributed adjacency arrays for representing the graph. A second method uses a sparse matrix representation and a two dimensional partitioning among processors. They found that their two-dimensional partitioning-based approach reduced the communication overhead by a factor in 3.5. In this method the edges and vertices are assigned to processors according to a two-dimensional block decomposition, so each node stores a sub-matrix of dimensions in its local memory, thus decreasing the need for communication between nodes. In the one-dimensional variety of their algorithm works by letting each processor own n/p vertices and all the outgoing edges from those vertices. This paper found that their ”hybrid parallel” approach can result in better performance than prior work. This paper also makes use of the compressed sparse row representation of the graph, pointing out that adjacency lists would incur more cache misses without providing additional benefits
Another paper that looks at the parallel breadth first search algorithm [6] works by using a ”bag” rather than a FIFO. This bag must allow for fast insertion while also allowing to be split and unioned efficiently in order to overtake a FIFO data structure in efficiency. These bag data structures have methods such as BAG_CREATE, BAG_INSERT, BAG_UNION, and BAG_SPLIT. A bag is made of pennets,

which are a tree of 2^k nodes. This paper also used the Cilk library to achieve parallization, especially taking advantage of the Cilk++ reducer hyperobject, which allows updates to a shared variable or data structure to occur at the same time without contention. They overall found increases in speed over the sequential version of BFS.

III. Methods

In this program the example graphs are explored using a breadth first search algorithm that uses a frontier, rather than a queue to keep track of those elements that have been found already but whose neighbors have not yet been explored. This frontier is held in a standard array, allocated on the heap. These elements in the frontier are explored in parallel, however to avoid a data race a basic primitive of the $bool\_compare\_and\_swap$ is used in order to set the distance of the member of the frontiers neighbor to the current distance plus one, which is used to keep track of the correctness of the algorithm. The $bool\_compare\_and\_swap$ prevents two processors from setting the distance of the same element in memory. This primitive is necessary as it is possible that two frontier members may attempt to set the distance of their neighbor at the same, but they may share this element, resulting in a data race. This would not necessarily result in incorrect results in the original form of this parallel breadth first search but is necessary when we get into further optimizations of the algorithm. Those elements whose distances are set in this round are then added to the frontier. The data structure these benchmark graphs are stored in is CSR. There are multiple ways a graph can be stored, including: a square matrix in which every element is represented twice and holds either a 1 if two nodes are connected with an edge, or a 0 if they are not. While this method has a very fast lookup to check for an edge between two nodes, this form of graph storage can take up a large amount of memory space for large, sparse graphs. Another form is adjacency lists, in which every node has a linked list that holds all the nodes it shares edges with. However, this storage of the graph can result in longer access times for checking if a node shares an edge with another. A linked list can grow very long in large graphs and it requires traversing along the linked list, which can be inefficient if there are many random accesses to access each data element. In order to also facilitate easy lookup within the parallel setting it is more efficient to use an array data structure that can allow the program to access elements that are stored sequentially. To facilitate this the graphs used in this project are stored in a packed compressed sparse row [4]. This storage format allows for graphs to be stored compactly while also allowing for fast lookup. An offset array keeps the index for every element which indexes into an edges list that holds every neighbor of node i in $Edges[offset[i]]$ to $Edges[offset[i+1]]$ [4]. The compressed sparse row format allows the algorithm to quickly look up the neighbors of a node i using i as an index. In the used example graphs the nodes are assumed to be numbered from 0 to n. One downside of this compressed sparse row representation of the graph is that it is very inefficient for dynamic graphs and is not easily changed once created.

A. Parallel Algorithm Building Blocks

1) Scan: The above description of the algorithm is a generalization but is implemented using a set of algorithmic building blocks. This includes all prefix sum, filter, pack and flatten. The all prefix sum algorithm is a common basis of algorithm design. A straightforward definition of the all prefix sum algorithm is an operation that applies a binary associate operator to an ordered set of n elements [1]

$$[a_0, a_1, a_2 ... a_n] \tag{1}$$

and returns an ordered set

$$[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2 ... a_0 \oplus a_1 \oplus a_2 \oplus a_3 .. \oplus a_n] \tag{2}$$

There are two different forms of scan, scan inclusive and scan exclusive. The scan inclusive was described previously but the exclusive scan takes a [a0,a1,a2..an] array and returns an array of

$$[0, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2 .. a_0 \oplus a_1 \oplus a_2 \oplus a_3 .. \oplus a_{n-1}] \tag{3}$$

If the operator is addition then a starting array

$$[1, 2, 3, 4, 5, 6, 7] \tag{4}$$

would return in the inclusive scan

$$[1, 3, 6, 10, 15, 21, 28] \tag{5}$$

and return

$$[0, 1, 3, 6, 10, 15, 21] \tag{6}$$

in exclusive scan

All prefix sums have many uses within existing algorithms. An all prefix sums on a one dimensional array is referred to as a scan operation and its parallel implementation is one of the main building blocks of the breadth first search algorithm in this project. Its sequential implementation is easy to understand and code, and can be found by looping through all the elements of an array and summing from 0 to i for every element scan[i]. However, the parallel version of the scan can be slightly more complex and require additional planning. Another important factor to consider when designing scan algorithms is its complexity. For this project the model of parallel computation is called the parallel random access machine (PRAM). More specifically the exclusive read exclusive write (EREW) that does not allow the simultaneous read or write of the same memory location by two different threads/processors [1]. In this model two different methods of measuring complexity are considered, work and span. Work is the total number of operations the algorithm performs. Span is known as the step complexity, the number of steps that the algorithm executes. Span is also sometimes referred to as depth, as it can also be considered as the deepest chain of dependent operations within the program as represented by a directed acyclic graph of program execution. When designing parallel algorithms that have sequential counterparts the goal is to design a work-efficient parallel algorithm, in which the work of the parallel algorithm is no greater than the complexity of its sequential version asymptotically [1]. The span or depth algorithm restricts how quickly different threads can finish their execution and the advantage of parallelism is a reduction in the number of steps of an algorithm.

2) Filter, Pack, and Flatten: The other essential utility functions for this project filter, pack, and flatten utilize the scan algorithm as an essential building block, which makes the span and work of the scan algorithm a large factor in the overall efficiency of the breadth first search algorithm. A filter function takes a flag array bitmap, that corresponds to another array with other elements shows which elements are meant to be included in the new array. Within the filter algorithm the pack algorithm is also used to move all these elements to their index within the new array, using the exclusive scan algorithm to find these new indexes. The flatten algorithm is used to flatten a two dimensional array into a one dimensional array that includes all the elements of the separate arrays, thus flattening it to a single dimension.

Overall, a parallel algorithm can have multiple different implementations. In general, the goal of turning a sequential algorithm into a parallel algorithm is to ensure that the parallel algorithm is work efficient, as it does no more work than its sequential version [2].

B. Parallel Breadth First Search Algorithm

As stated earlier the sequential breadth first search algorithm is implemented by starting at a source node and exploring all neighboring nodes. Those nodes are then pushed to a queue, whose first in first out structure allows the algorithm to explore the first level of a graph before moving onto subsequent levels.

```
$d[i] \leftarrow -1$    ▷ All nodes distances are initialized to -1
frontierSize $\leftarrow 1$    ▷ The first frontier is always of size one
curr_dist $\leftarrow 1$ ▷ This the number of edges from the source node to the current node
dist[s] $\leftarrow 0$    ▷ Source distance is zero
frontier[0] $\leftarrow s$    ▷ Holds the current frontier
curr_dist $\leftarrow 1$
while $frontier \neq 0$ do
    effective_neighbors $\leftarrow$ [frontierSize]▷ Initalize an array of arrays to store which neighbors will be apart of the next frontier
    for $i \leftarrow 0$ to frontierSize do
        current_vertex $\leftarrow$ frontier[i]
        $k \leftarrow$ the number of current vertex neighbors    ▷ Number of neighbors the current node has
        flag $\leftarrow$ new int[k]    ▷ Initialize the flag array
        for $j \leftarrow 0$ to $k$ do    ▷ For every neighbor of the current node
            if neighbor has not been visited then    ▷
                dist[neighbor] $\leftarrow$ curr_dist    ▷ Set neighbors distance to the current distance
                flag[j] $\leftarrow 1$    ▷
            else
                flag[j] $\leftarrow 0$
            end if
        end for
        curr_dist+=1
    end for
end while
neighbors_array_length[i] $\leftarrow$ next frontier length
effective_neighbors $\leftarrow$ filter(flag)
frontier $\leftarrow$ pack_flatten(effective_neighbors)
```

The parallel implementation of breadth first search requires the use of a frontier to allow the concurrent exploration of multiple nodes at a time in order to get improvements in speed over the sequential algorithm. The algorithm begins with a single element in the frontier, the source node. Here, it is picked based on the most connected component, the node who has the greatest number of connections to other nodes, thus getting the largest possible graph to test the breadth first search on, since some nodes within the benchmark graphs can have fewer connections or none within the graph if they are randomly selected. For every node in the frontier the algorithm then explores each of its node's neighbors (every node that the current node shares an edge with) within a parallel for loop, making sure that only one thread can set the neighbors distance within the distance array at a time, those elements whose distances are not -1 are skipped as already explored or currently in the frontier. Each of these elements whose distances are set in this round are added to the effective neighbors for that element of the frontier. Each frontier element gets an effective neighbor array within a nested effective neighbor array. This is where the pack, filter, and flatten arrays are used. For each element in the frontier a flag array is used, a bit map, in which those elements that will be added to the next frontier are marked by a corresponding true/false or 1/0 within the flag array. This flag and the original array are then filtered and packed creating a new array that only includes those neighbors of a particular current frontier member. After all current frontier members have been explored, those filtered and packed arrays and then flattened into a contiguous array, creating the next frontier. This continues until there are no more elements in the frontier. This is a very straightforward implementation of the parallel breadth first search algorithm, however, tested against a sequential implementation of the breadth first search algorithm, it falls short of achieving any speed up over the original implementation. In fact it is often much slower, often the result of the overhead of creating new threads, how wide the diameter of the graph is, or the size of the frontier.

One major way to decrease the overall number of created threads within the algorithm is to add granularity to these functions that perform the building blocks task, most importantly the scan algorithms as they are used in nearly part of the algorithm and within the filter, pack and flatten functions.

Another optimization within the algorithm is allowing for the algorithm to switch between when the current frontier is sparse or dense. A very large frontier could result in many calls to the scan, pack, and flatten functions which, with or without granularity control can be a drag on the overall runtime of the breadth first search. Additionally with a granularity control on these functions of a constant value can allow smaller frontiers to skip generating threads and help avoid too much overhead for function calls. Adding a dense mode to the algorithm can speed up the overall execution time of the algorithm by skipping these extra function calls. The dense mode of exploring the graph concentrates on finding those nodes not yet explored whose neighbors are currently within the frontier. For every element in parallel, it is checked if its distance has yet to be set in the distance array, if it has not, its neighbors are searched and checked if any of them have yet been explored. If so, the loop searching the neighbors can be broken and the nodes distance can be changed and the node added to the dense frontier. This method also works better when many of the nodes have already been explored and it is quick to find those nodes that have not yet been explored. A well chosen heuristic to determine when switch between a dense and sparse mode is necessary to get the best speed ups. Currently the algorithm operates by switching from the sparse and dense mode if the frontier becomes larger than the square root of n, the number of nodes.

IV. Results

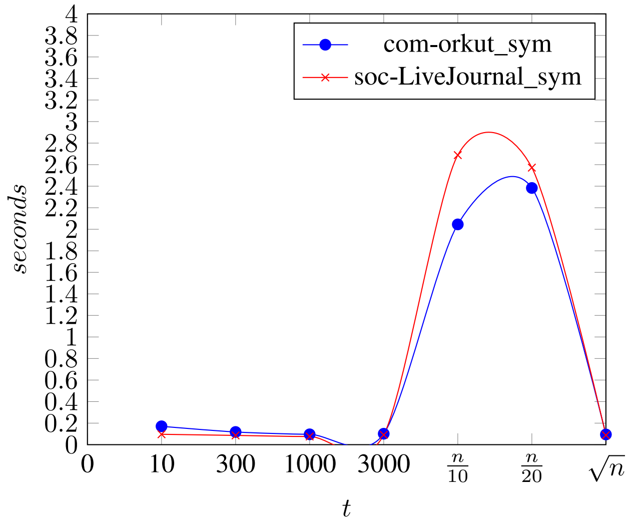
As discussed above, the initial runtime of the parallel breadth first search algorithm was much slower than the sequential version. Requiring some optimizations within the algorithm to speed execution and avoiding some of the pitfalls of straightforward parallel computing. The benchmarks in this project are: $com-orkut\_sym$ and $soc-LiveJournal\_sym$, both real world graphs. The first major optimization used in the project is adding granularity to the parallel building block algorithms, this includes scan, filter, pack and flatten. The more simple granularity added to the filter, pack and flatten are simple translations of the parallel for loops that make up the parallel version of these algorithms to sequential for loops. A constant is used to determine when the algorithm should switch between these parallel and sequential implementations, for each, $n == 200$ is used but there are potential investigations of constants that could result in greater speed ups in execution. Using this, the algorithm can now execute a correct breadth first search on the given benchmarks without being killed after running out of memory due to the many parallel calls.
The scan algorithm requires additional work to speed up its initial execution. The completely parallel scan operation divides the scan into two separate functions, a scan up and scan down operation. The scan up operation building a second array, the left sum array, using another parallel algorithm building block reduce. Reduce splits the initial array into a left and right halves, recurse on each until on a single element remains in the left and right halves and returning the sum between each left and right to the previous call eventually returning the sum off all elements in the array. During this operation, many partial sums are generated, and it is these partial sums that are used to generate this left sum array to be used in the scan down algorithm. At each level the left sum of this operation is saved and used later in the scan down operation, which uses these left sums as offsets to the full scan operation, allowing the algorithm to generate the prefix sum array with a smaller span/depth. As mentioned earlier the scan operation is a major component of the overall algorithm and is used as a building block of many of the other functions so its overall efficiency is important for the overall speed of this breadth first search algorithm. Adding granularity to the scan up operation in particular was a large challenge, however, there are multiple ways to add coarsening to this scan algorithm. On of the main is to split the array and running the scan algorithm on each split, thus shrinking the overall overhead of generating threads.
The results of adding granularity are displayed in this table:

	com-orkut_sym	soc-LiveJournal_sym
sequential BFS	0.78641s	0.519516s

	com-orkut_sym	soc-LiveJournal_sym
No granularity	223.366	201.6109
granularity	121.822s	72.2982s

The results of the addition of the dense mode to the algorithm had the greatest benefits overall the execution time of each test. Based on a certain heuristic or constant different run-times using the dense mode can be observed. The main function call within this dense execution is the filter operation.

Value of t	com-orkut_sym	soc-LiveJournal_sym
10	0.170589s	0.0963665s
300	0.117847s	0.0859367s
1000	0.0961098s	0.0759367s
3000	0.101551s	0.0913236s
$\frac{n}{10}$	2.04526s	2.68852s
$\frac{n}{20}$	2.38332s	2.57237s
$\sqrt{n}$	0.0946087s	0.0879946s



V. Discussion

Overall it appears that the largest effect on speed of execution for this parallel breadth first search algorithm is if the building block algorithms of scan, filter, pack and flatten have added granularity and the switch from a dense to sparse frontier mode. Based overall on these results it appears that there is a range of values of t (the value in which the algorithm switches between dense and sparse modes) that can result in significant improvements in speed. To avoid declaring constant values it appears that $\sqrt{n}$ is a good setting for this value.

VI. Continuing work

There is continuing work in this same section, one of the main ones are the breadth first search on wide diameter graphs. A wide diameter graph is a graph who has fewer connectivity between nodes, resulting in much longer runtime to explore the graph in the parallel breadth first search. Running this algorithm on these wide diameter graph results in creating many instances of the frontier of similar size unlike the other test graphs which begin with smaller frontier sizes, peaking at a very large frontier, and finishing in a few rounds with smaller frontiers. One way to increase the speed of execution for these wide diameter graphs is to add shortcuts between nodes, increasing the number of edges, and shrinking the diameter [4]. There are further optimizations within this to most efficiently generate shortcuts between nodes.
Additional further work in this same section is in creating a parallel version of single source shortest path algorithms, such as Dijkstra's and Bellman-Ford which have yet to develop a work-efficient SSSP algorithm. This could be considered an extension of the work in parallel breadth first search.

Acknowledgment

Very big thanks to my Advisor Dr. Yihan Sun at the University of Calfornia, Riverside for advising me for this project.

References

[1] Blelloch, G. E. (n.d.). Prefix sums and their applications - Carnegie Mellon University. Retrieved April 13, 2023, from https://www.cs.cmu.edu/ guyb/papers/Ble93.pdf

[2] Chatterjee, S., &amp; Prins, J. (n.d.). Comp 203: Parallel and Distributed Computing Pram Algorithms. Retrieved April 13, 2023, from https://homes.cs.washington.edu/ arvind/cs424/readings/pram.pdf.

[3] Dong, X., Gu, Y., Sun, Y., &amp; Zhang, Y. (2021, December 14). Efficient stepping algorithms and implementations for parallel shortest paths. arXiv.org. Retrieved April 12, 2023, from https://arxiv.org/abs/2105.06145.

[4] Wheatman, B., &amp; Xu, H. (n.d.). Packed compressed sparse row: A dynamic graph representation. Retrieved April 13, 2023, from http://supertech.csail.mit.edu/papers/WheatmanXu18.pdf.

[5] Buluç, A., &amp; Madduri, K. (n.d.). Parallel breadth-first search on distributed memory systems. Retrieved April 13, 2023, from https://people.eecs.berkeley.edu/ aydin/sc11_bfs.pdf

[6] CSAIL, C. E. L. M. I. T., Leiserson, C. E., Csail, M., CSAIL, T. B. S. M. I. T., Schardl, T. B., Paderborn, U. of, Laboratories, S. N., Contributor MetricsExpand All Charles E. Leiserson MIT Computer Science &amp; Artificial Intelligence Laboratory, &amp; Charles E. Leiserson MIT Computer Science &amp; Artificial Intelligence Laboratory Publication Years1980 - 2019Publication counts107Available for Download58Citation count15. (2010, June 1). A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers): Proceedings of the twenty-second annual ACM Symposium on parallelism in algorithms and Architectures. ACM Conferences. Retrieved April 13, 2023, from https://dl.acm.org/doi/pdf/10.1145/1810479.1810534