Steven Cabral: 6801179
Bronwyn Perry-Huston: 5537360

Final Project Report

Introduction

For our project, we opted to implement a parallel breadth-first search algorithm. The breadth-first search algorithm has been around for a long time, but it is becoming more and more relevant as the demand for graph analytics increases. Graphs are becoming more and more widely used, in anywhere from social media to process management. Breadth-first search is the easiest and most common way to analyze such graphs, but it is notoriously difficult to parallelize efficiently. Our goal was to take an existing breadth-first search algorithm, parallelize it, and improve the efficiency as much as possible.

The Data Structure

```
PENNANT-UNION(x,y)
1  y.right = x.left
2  x.left = y
3  return x
```

```
PENNANT-SPLIT(x)
1  y = x.left
2  x.left = y.right
3  y.right = NULL
4  return y
```

From Leiserson & Schardl

Our initial approach followed in the footsteps of Leiserson and Schardl to create a specialized structure called a *bag* [1]. We also chose to use the parallel library Cilk so that we could implement a special feature of Cilk called a reducer, which is effective at aggregating and synchronizing data across many processors automatically. The bag itself is designed to have three main operations: union, split, and insert. Each bag maintains an array of pointers to an auxiliary structure called *pennants*, which are merely unordered binary trees with an empty right subtree at the root. This property allows the pennants to be easily unioned and split in constant time. The pennants are also maintained so that each tree (beyond the root node) is filled to a constant depth.

```
BAG-INSERT(S,x)
1  k = 0
2  while S[k] ≠ NULL
3      x = PENNANT-UNION(S[k],x)
4      S[k++] = NULL
5  S[k] = x
```

```
BAG-SPLIT(S1)
1  S2 = BAG-CREATE()
2  y = S1[0]
3  S1[0] = NULL
4  for k = 1 to r
5      if S1[k] ≠ NULL
6          S2[k-1] = PENNANT-SPLIT(S1[k])
7          S1[k-1] = S1[k]
8          S1[k] = NULL
9  if y ≠ NULL
10     BAG-INSERT(S1,y)
11 return S2
```

From Leiserson & Schardl

Because of this requirement, each bag maintains an array of pennants, such that, for pennant indices 0 through n, the i'th pennant contains exactly 2^i items. This gives the bag itself a structure analogous to a bit string. Because of this property, the bag operations work similarly to

```
BAG-UNION(S1,S2)
1  y = NULL    // The "carry" bit.
2  for k = 0 to r
3       (S1[k],y) = FA(S1[k],S2[k],y)
```

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	PENNANT-UNION(x,y)
1	0	1	NULL	PENNANT-UNION(x,z)
0	1	1	NULL	PENNANT-UNION(y,z)
1	1	1	x	PENNANT-UNION(y,z)

From Leiserson & Schardl

integer arithmetic operations. For example, adding an item is done much the same as adding 1 to an integer: insert the item at the lowest level, and if that level is already occupied, "carry" that level and attempt to insert it at one level higher. Additionally, unioning works similarly to integer addition - each "bit" of the insert takes the same level of both bags plus a carry-in and outputs a bag of the same level plus a unioned bag of a higher level as a carry-out. Likewise, split is the equivalent of splitting each of the pennants to add to a new bag, shifting all the entries in the old bag down by a power of 2. However, unlike integer shifting, the remainder is added in to avoid losing entries.

Next, we planned on implementing a reducer structure which would use these bag operations automatically to maintain the data across processors. However, reducers proved incredibly difficult to debug effectively, so we opted for a different approach. In place of a reducer, we had each strands maintain its own bag, and the processors manually merged these bags at the end of each iteration of the main loop.

The Algorithm

```
for i = 1 to maxlevel
  v = queue.pop()
  for each neighbor w of v
    if level[w] == -1
      parent[w] = v
      level[w] = thislevel+1
      levelsize[thislevel+1]++
      queue.push(w)
```

Original sequential algorithm from Prof. Gilbert's code

For the main breath-first search kernel itself, our goal was to take an existing breadth-first search and parallelize it as well as possible. This approach has a few main advantages - first, breadth-first search code is very common and easy to come by. Another advantage is that having a functional version of breadth-first search is good for running correctness tests alongside the parallel version to verify that the parallel algorithm is giving the correct output on any new graph. Additionally, a serial breadth-first search can also give excellent timing data as well to use as a benchmark alongside the parallel version to measure the effectiveness of the optimizations empirically. The specific code that we used for the serial version was the sample code given to us by Prof. Gilbert.

Creating Parallelism

The first step for our implementation was the actual parallelization. For this step of development, we weren't concerned with efficiency, but instead were only focused on allowing the algorithm

```
cilk_for i = 0 to levelsize
```

Initial parallel loop

to run on multiple processors. Specifically, what we did was overall rather simple once we had the bag class working. First, we changed the per-level loop into a cilk_for loop to enable the parallel processing. Since the original algorithm used a queue to add the next vertices, we had to change to a parallelizable structure - namely, the bag class. Additionally, we had to divide the sets so that we had two sets: one set that is currently being traversed and one frontier set to contain the untraversed neighbors of the current set. The frontier set was instantiated as a set of bags, while the current set was the previous frontier set converted to array format.

However, though this form was technically in parallel, each thread only explores a single vertex, which is a minimal amount of work compared to the overhead needed. Additionally, a large amount of bag unions is required in an approach like this. The next step became increasing the amount of work done by each processor to allow for a smaller number of threads. To solve this issue, we converted the main loop into two loops, the outer of which is parallel. The outer loop was given a fixed amount of iterations dependent only on the number of workers (processors) and a set constant. Exactly one bag was given to each thread, drastically reducing the number of bags to be unioned. The work was divided distributed to each thread in the inner loop, with the remainder being handled by the head node.

```
cilk_for i = 0 to numchunks by 1
  for j = 0 to chunksize by 1
```

First optimization - replaces initial parallel loop

```
BAG_MERGE(array, size)
  if size > GRANULARITY
    a = cilk_spawn BAG_MERGE(array, size/2)
    b = cilk_spawn BAG_MERGE(array + size/2, size/2+size%2)
    cilk_sync;
    return BAG_UNION(a,b)
  else
    for i = size - 2 to 0 by -1
      BAG_UNION(array[i], array[i+1])
    return array[0]
```

Second optimization - implement BAG_MERGE

To improve the breadth-first search even further, we focused on the process for merging the bags, which until this point was done in serial by a single processor. Since this had a linear time that could not benefit from parallelism, we decided to create a recursive parallel merging algorithm for logarithmic runtime. The algorithm here was very simple: if the set of bags is larger than a set granularity, then split the array in half and do a merge in parallel on the two halves. If the size is smaller than or equal to the granularity, then the bag is simply merged sequentially. The granularity here was set as a parameter which we could use later in tuning.

The final set of optimizations targeted the other operations in the breadth-first search which could be included in the calculation within the main loop. Most of this step was cleaning up small inefficiencies in the code, though the most significant optimization was including the calculation of the size of the frontier set in the parallel loop. To do this, we gave the bag class a size variable and properly accounted this value in the initialization and operations the bag class can perform. The size of the final merged bag can then be taken as the size of the frontier set. Additionally, we attempted to tune the parameters to achieve the best performance, but our original parameters, recursive coarseness = 100 and threads per worker = 4 appeared to be most optimal.

Experiments

After parallelizing the Breadth First Search algorithm, we quantified its performance, accuracy, and scalability through experiments on graphs of increasing size. To ensure that we solely gathered data on the algorithm and not the graph creation overhead, we only measured the execution time of the our BFS function. Using generated RMAT graphs of increasing size we performed experiments to gauge the accuracy and the scalability of our algorithm with increasing problem size and number of processors.

RMAT graphs are power-law graphs of a specified size that are used by the Graph500 benchmark to test submissions. We generated RMAT graphs of increasing size that ranged from 17 to 15 million edges to create a testing suite for our algorithm. Unfortunately, when generating large RMAT graphs for testing, the lack of adequate connectivity in the graphs made it difficult to find a starting node for the BFS algorithm that reached an adequate number of other vertices. To combat this difficulty, we decided to always begin the breadth first search from vertex one, and we created new graphs until the first vertex was suitably connected.

We first quantified the performance of our parallel algorithm by measuring the scalability with respect to the problem size. Starting at vertex one we timed the execution of our BFS algorithm on graphs with tens, thousands, tens of thousands, hundreds of thousands, millions, and tens of millions number of edges. Execution timing for each graph was collected for the serial, un-optimized parallel, slightly optimized parallel, more optimized parallel, and final optimized parallel version of the algorithm to observe the relative performance of each iteration. We used 1 processor for the serial algorithm and 4 processors for the parallel versions to ensure consistency for the parallel algorithms. The timing data of each algorithm for each graph size was used to calculate the speedup for each iteration of of our parallelized algorithm, producing metrics to gauge the performance of the algorithm with respect to problem size.

After measuring the performance due to increasing problem size, we experimented with the number of processors to determine the scalability of the parallelism of each version of the parallel algorithm. We ran our BFS algorithms on a generated RMAT graph with 7.5 million edges and 8 million vertices to measure the performance on a suitably complex and large problem size. Timing data of each parallelized version of the algorithm was collected from 1, 2, 3, 4, 6, 8, 12, and 16 processors, allowing the quantification of each algorithms relative performance as the number of processors increases. The results generated from this experiment allowed the calculation of parallel efficiency of each iteration of the parallel algorithm. Trends in the changes of parallel efficiency from increasing number of processors quantify demonstrate the degree of scalability of parallelism in the algorithms.

Finally, the accuracy of each iteration of parallel algorithm was confirmed by using the results of the serial algorithm as a baseline. A primary goal of our project was to parallelize and optimize an existing serial algorithm while preserving the algorithms accuracy. The serial version of the breadth first search was used on each of our generated RMAT graphs, and the output was established as the "correct" result for each respective graph. The output of each parallel algorithm was compared to this baseline for each graph used in each experiment to determine accuracy. Unfortunately, it is nearly impossible to prove unequivocally that our algorithm will always produce the correct results. To increase our confidence in the accuracy of the parallel algorithm, we tested each graph multiple times to check that the results were consistent and accurate every time.

Results

The results of the above experiments indicated a significant improvement in performance in the final parallelized algorithm over the initial serial BFS algorithm. The raw data results of the experiments testing increasing problem size are illustrated in Table 1:

N_Edges	Serial_BFS	Not_Opt_Parallel	Slightly_Optimized	More_Optimized	Final_Optimized_Parallel
17	4e-06	0.001746	0.00053	0.00033	0.000361
2561	0.000108	0.00255	0.000706	0.00039	0.000421
12289	0.000493	0.004367	0.001372	0.001197	0.001144
16384	0.000242	0.003369	0.002076	0.000703	0.001329
28673	0.000961	0.007486	0.001499	0.001406	0.000635
32768	0.000389	0.002692	0.001253	0.001213	0.000656
81921	0.002548	0.017255	0.002948	0.001645	0.00188
1.3107e+05	0.000719	0.002579	0.002015	0.001782	0.001013
1.6384e+05	0.005076	0.020176	0.004578	0.002904	0.002397
2.6214e+05	0.008211	0.049993	0.003633	0.003241	0.002999
3.2768e+05	0.009651	0.0303	0.00456	0.0035	0.003006
8.3886e+05	0.017714	0.079691	0.005214	0.004832	0.003174
1.5729e+06	0.018805	0.1207	0.0106	0.009643	0.006386
1.835e+06	0.024403	0.13941	0.013807	0.012613	0.007259
1.8874e+06	0.049877	0.3405	0.047415	0.047478	0.029409
7.5497e+06	0.34077	1.3854	0.22134	0.20898	0.12935
1.5099e+07	0.65836	2.7031	0.45627	0.42187	0.26454

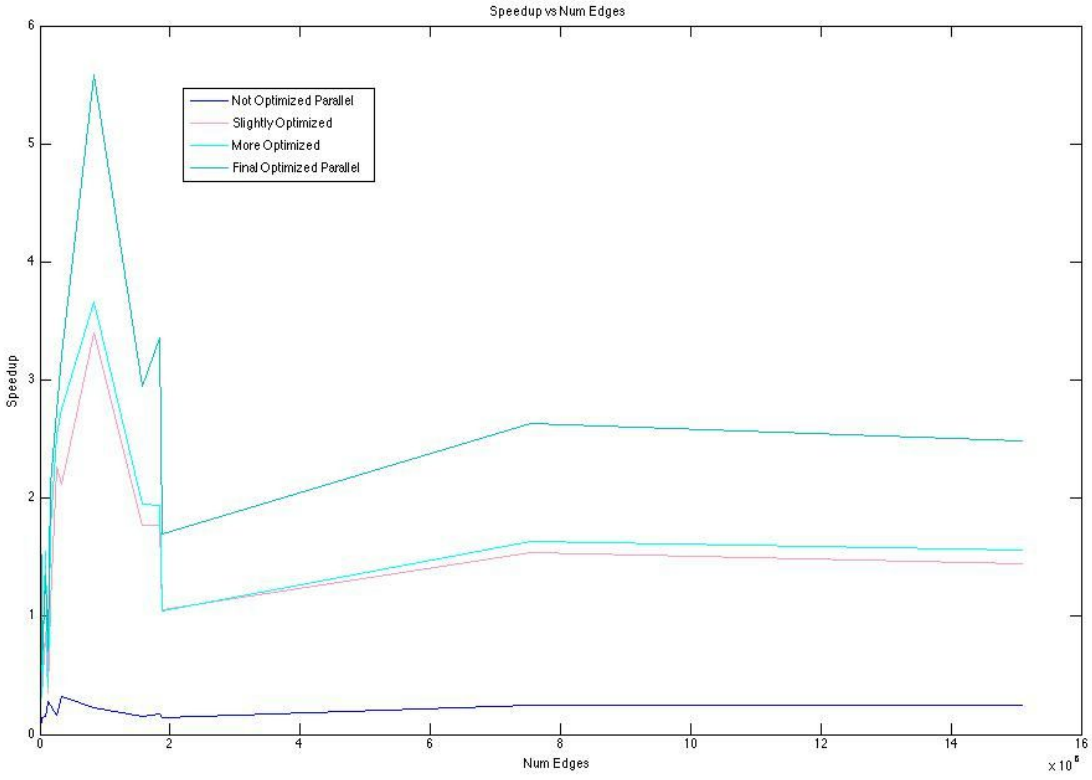
Table 1) Time in seconds of each algorithm on each graph

From this raw data, the speedup of each iteration of the algorithm was calculated and collected into a table:

N_Edges	Not_Opt_Parallel	Slightly_Optimized	More_Optimized	Final_Optimized_Parallel
17	0.002291	0.0075472	0.012121	0.01108
2561	0.042353	0.15297	0.27692	0.25653
12289	0.11289	0.35933	0.41186	0.43094
16384	0.071831	0.11657	0.34424	0.18209
28673	0.12837	0.64109	0.6835	1.5134
32768	0.1445	0.31045	0.32069	0.59299
81921	0.14767	0.86431	1.5489	1.3553
1.3107e+05	0.27879	0.35682	0.40348	0.70977
1.6384e+05	0.25159	1.1088	1.7479	2.1176
2.6214e+05	0.16424	2.2601	2.5335	2.7379
3.2768e+05	0.31851	2.1164	2.7574	3.2106
8.3886e+05	0.22228	3.3974	3.666	5.581
1.5729e+06	0.15579	1.7741	1.9501	2.9447
1.835e+06	0.17505	1.7674	1.9347	3.3618
1.8874e+06	0.14648	1.0519	1.0505	1.696
7.5497e+06	0.24598	1.5396	1.6306	2.6344
1.5099e+07	0.24356	1.4429	1.5606	2.4886

Table 2) Speedup of each version of the parallel bfs algorithm

The relative speedup for each problem size provides a metric to determine the scalability of each algorithm's performance as problem size increases. The changes in speedup create trends that indicate this scalability. (Graph 1)



Graph 1 illustrates the speedup of each iteration of optimization of the parallel BFS algorithm, and demonstrates the significant increase in speedup from the un-optimized parallel algorithm to the final optimized version. These results also indicate the relative performance increase due to each optimization of our parallel BFS algorithm.

The raw data for the experiments quantifying the scalability of parallelism is presented in table 3:

N_Cores	Not_Opt_Parallel	Slightly_Optimized	More_Optimized	Final_Optimized_Parallel
1	1.4731	0.43087	0.46321	0.31942
2	1.9111	0.38015	0.41097	0.30524
3	1.7939	0.36604	0.39103	0.24457
4	1.6492	0.31967	0.35027	0.23334
6	1.5726	0.27089	0.27568	0.17189
8	1.7676	0.29479	0.27482	0.17707
12	1.8861	0.36783	0.36097	0.22317
16	2.0875	0.41748	0.3664	0.26861

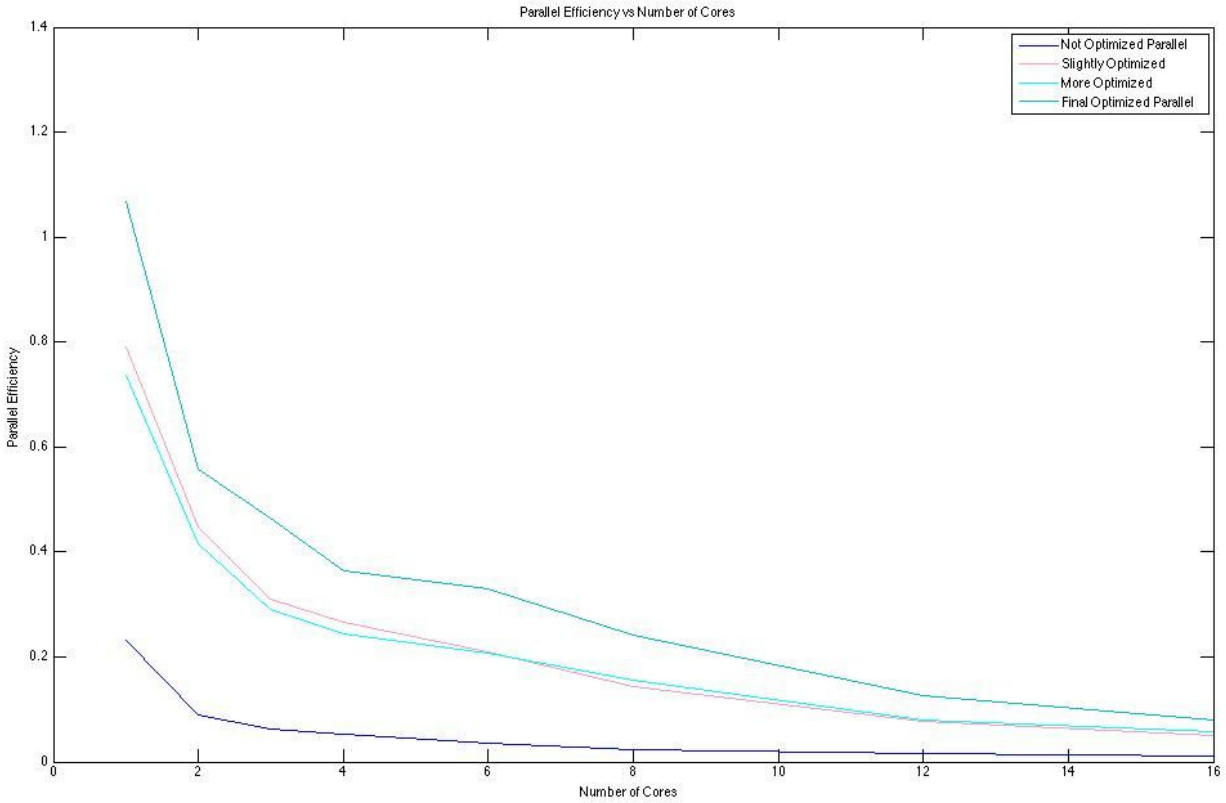
Table 3) Time in seconds to run BFS on a 7.5 million edge graph

This timing data was used to calculate the parallel efficiency of each respective iteration of optimization to provide an informative metric to gauge the parallel scalability.

N_Cores	Not_Opt_Parallel	Slightly_Optimized	More_Optimized	Final_Optimized_Parallel
1	0.23132	0.7909	0.73567	1.0668
2	0.089154	0.44821	0.41459	0.55819
3	0.063319	0.31033	0.29049	0.46445
4	0.051656	0.2665	0.24322	0.3651
6	0.036116	0.20966	0.20602	0.33043
8	0.024098	0.1445	0.155	0.24056
12	0.015056	0.077203	0.07867	0.12725
16	0.010203	0.051016	0.058129	0.079289

Table 4) Parallel Efficiency of a BFS for each number of processors

The change in parallel efficiency of a BFS by each iteration of the parallel algorithm as the number of processors increases can be used to determine the ability of our algorithm to parallelize a specific input graph. The data presented in Table 4 was graphed to visually display the relative parallel efficiencies of each algorithm for each number of processors, allowing interpretation of the results of the experiment.



Graph 2) Parallel Efficiency vs Number of Cores

Discussion

The trends presented in Graphs (1) and (2) illustrate the performance increases of the final optimized parallel algorithm over the serial version and quantitatively demonstrate the improvements of each optimization of our parallel breadth first search implementation.

The speedup information shown in Graph (1) indicates that the optimized parallel BFS algorithm had a significant improvement in performance over the serial version BFS algorithm. Speedup is defined as a metric for relative performance improvement when executing a task. The speedup of >1 of the final parallel version for all problem sizes(Graph 1) means that the optimized parallelization of the BFS algorithm improves its performance. The maximum speedup of the optimized parallel algorithm is 5.58 and occurs when the problem size is 800,000 edges. This means that at this problem size, the parallel version is 5.58 times better than the serial version. The increases in speedup with each optimization, as seen in Graph 1, quantify and demonstrate the improvements of each optimization to the parallel algorithm. Initially, the un-optimized parallel algorithm performed worse than the serial version, with speedups <1 for all

tested problem sizes. However, as the algorithm was optimized, speedup was observed to increase as expected.

This speedup information also allows us to understand the relative performance of the parallel algorithm as problem size increases. For graphs with less than 100,000 edges the optimized parallel algorithm has a speedup less than 1, indicating that it has a worse performance than the serial algorithm. However, for larger problem sizes the speedup increases to values greater than 2, demonstrating that the parallel algorithm has improved performance over the serial version for larger problem sizes. This result makes sense intuitively because for small problem sizes the overhead required to parallelize the problem overshadows the increase in efficiency of the BFS search itself. However, for large problem sizes, the ability to explore every vertex in a level at the same time decreases the overall execution time of the algorithm.

The shape of each speedup curve is also consistent for each iteration of the parallel algorithm's optimization, which is interesting and indicates a key characteristic of our implementation that was not altered by the optimizations. The optimizations to the parallel algorithm didn't decrease the inherent limitations in scalability of our implementation, they just increased the ability of Cilk to parallelize and speedup the BFS operation as a whole. This result is interesting, and indicates that changes to the structure of the implementation itself may need to be changed to improve the algorithm's performance and scalability.

The parallel efficiency trends displayed in Graph (2) demonstrate the performance improvements from each of the optimizations to the initial parallel algorithm. Parallel efficiency is the speedup/number of processors and would be a horizontal line at 1.00 in an ideal program. The negative slope of the parallel efficiency curve is common, and indicates that each processor becomes decreasingly useful as the number of processors increases. Breadth first search is difficult to parallelize, so it is not surprising that our observed parallel efficiency is low. Ideally, we want a parallel efficiency as close to 1 as possible, which indicates that each processor is being used to its full potential. However, in our experiments this value was never attained for more than 1 processor. The decrease in parallel efficiency with the increase in number of processors means that the scalability of parallelism of our algorithm is low. As the number of processors increases, the communication overhead between the large number of processors creates a bottleneck that decreases the efficiency of each processor. The merging of each bag structure in our algorithm requires communication between each processor, so as the number of processors increases, communication increases significantly which decreases the performance and efficiency of our algorithm. Collectively, these point all indicate that further changes and optimizations need to be made to improve the scalability of parallelism for our parallel BFS algorithm.

Although our parallelized algorithm exhibits a low parallel scalability, the relative parallel efficiency curves for each iteration of optimizations demonstrate the performance increases caused by each optimization. The initial, un-optimized parallel algorithm has a consistently very small value, meaning that it is not using each processor efficiently. This conclusion is expected and confirms the results displayed in the speedup tests that all show that the un-optimized parallel algorithm does not effectively increase performance. The next two optimized iterations, however, show a marked increase in parallel efficiency that coincides with the increase in speedup seen in Graph (1). This increase in parallel efficiency signifies that the optimization to the parallel algorithm directly increased performance. The final optimized algorithm has the highest parallel efficiency for each number of processors, indicating that the final version uses each processor to a higher portion of their potential than the previous iterations. Overall, the parallel efficiency may not be ideal, but the trends exhibited by each version of the algorithm confirms that each iteration of optimizations positively impacted performance and improved the efficiency in the use of each processor.

Conclusion

As demonstrated by the results of our experiments, the serial breadth first search algorithm was successfully parallelized and optimized to decrease execution time and increase performance. The algorithm was successfully parallelized to maintain 100% accuracy while achieving a maximum of 5.58x speedup. Although scalability with respect to problem size and number of processors was not ideal for any of the versions, each iteration of optimizations caused a visible, significant increase in performance and efficiency. Our implementation taught us the limitations of simply parallelizing a serial algorithm and revealed the need to completely restructure some portions of the algorithm to reduce unnecessary operations. We also learned the importance of creating data structures that can be easily separated and merged together to increase the ability of Cilk to parallelize it. Overall, we believe that we successfully accomplished the project objective of implementing a parallel breadth first search algorithm by parallelizing and optimizing an existing serial version of a BFS algorithm.

Citations

[1] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In SPAA, pages 303–314, 2010