Nathan Crandall, 3970001
Dane Pitkin, 4085726

CS140 Final Project

Introduction:

Our goal was to parallelize the Breadth-first search algorithm using Cilk++. This algorithm works by starting at an initial vertex and exploring all of its neighbors. It pushes these neighbors into a queue until every vertex one level away is in the queue. Then it pops off the initial vertex and repeats this process for every neighbor in the queue. This algorithm searches a graph's levels 0, 1, 2, 3, ..., to 'k' levels away from the starting vertex in that order, which is a breadth-first search. We started with a sequential version of this algorithm that uses a queue as stated above. In order to parallelize this algorithm, a different data structure must be used in place of the queue. Since the queue can only push one item in or pop one item out at a time, the queue cannot be accessed in parallel. There are several data structures to replace the queue with. The two implementations we tried were the STL lists wrapped in a cilk reducer and our implementation of a data structure called a Bag wrapped in a custom reducer.

Algorithm:

The parallel algorithm we used is fairly similar to the sequential version of the algorithm. We have a while loop that loops once per level of the graph, but instead of computing the neighbors in the while loop, we call a recursive function that in this loop. This recursive function takes two Bags, one containing the vertices of the current level and one for storing all newly explored neighbors at the next level. It first checks the size of the input Bag and compares it with the grain size (this grain size is customizable and changed accordingly to provide maximum runtime efficiency). If the size of the input Bag is larger than the grain size, it splits the Bag in half, spawns a recursive call with the new half Bag and then runs a recursive call on the same thread with the old half Bag. Once the Bag has been split into enough pieces, they will be smaller than the grain size and will then compute the neighbors of the elements in the Bag it has and store the neighbors in the output Bag (which is a hyperobject, so that it can be written to in parallel). There is a cilk for loop that explores the neighbors of each vertex. This is the limit of the parallelism for our parallel algorithm.

Data structure:

The final implementation for our parallel Breadth-first search algorithm uses a Bag data structure containing Pennant trees. A Bag is a multiset and so therefore it supports the union operation. The Bag is an array of uniquely sized Pennant trees. A Pennant tree is a complete binary tree with an extra node above the complete binary tree's root that is used as the Pennant tree's root. Each index i of the Bag corresponds to a Pennant tree of size 2^i, which is a complete binary tree of size 2^i – 1 + the root of the Pennant tree. Because of this property, the Bag works like a binary counter, with each index of the Bag either containing a Pennant tree or null pointer.

Inserting an element into the Bag is just a matter of unioning a single element into the Bag and updating the size of the Bag by 1 like a binary counter, with each Pennant tree

representing a bit. Inserting takes O(1) amortized time. This is because any Bag of size even will have an insert cost of O(1) time (a Pennant tree of size 1 is inserted in the first position of the Bag in O(1) time) and any Bag of size odd may take up to O(log n) to insert one element, but by pushing the Pennant trees farther into the Bag, one O(log n) insert could free up many O(1) inserts before another O(log n) insert needs to happen, which is why the insert function has an amortized time of O(1).

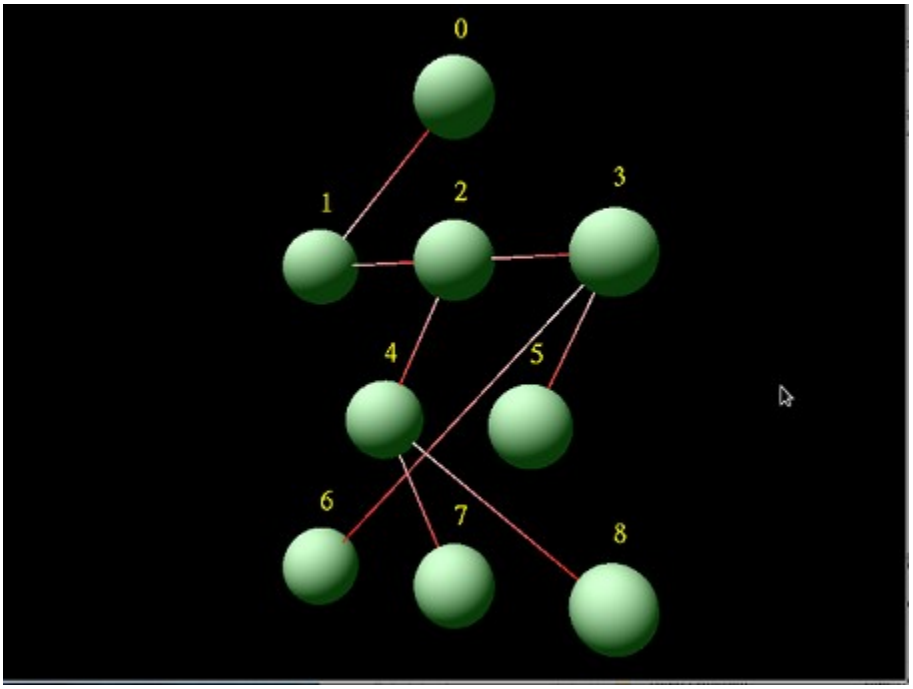
Our Bag not only can merge with another Bag, but it can also split itself in half and return a second Bag containing the other half of the elements. Both of these operations happen in O(log n) time. The merge operation works like a two bit Full Adder. It starts at the beginning index of each Bag and merges the two trees. Three cases may occur here. First, either one of the Pennant trees are empty and the non-empty Pennant tree gets pushed onto the resulting Bag. Second, if both Bags contain trees then they are merged together and carried to the next index in the merge operation. Third, if neither Bag has a tree for this index, then a null pointer is assigned to the resulting Bag. For every subsequent merge, a fourth step must be taken into account. The carried tree must be merged with the two next Pennant trees. The rules of merging still follow that of a two bit binary Full Adder. Split() is the inverse operation of merge(). The split function splits each tree and places the resulting Pennant trees in its index – 1 in each Bag. These operations are O(log n) because they call k operations, where k = number of Pennant trees in Bag = log n (n = number of elements in Bag), and each operation works in O(1) time (merging and splitting Pennant trees only rearranges the roots, which takes O(1) time).

These functions allow the Bag to be added to and split apart at any time, which means it is a great choice for using in a Parallel Breadth-first search algorithm. However, in order for our data structure to be accessed in parallel, it must be wrapped in a cilk hyperobject. Since we wrote our own data structure, there is no pre-made cilk hyperobject that we can use. Therefore, we needed to write our own. The type of hyperobject that we implemented was a cilk reducer. This is the appropriate option because our parallel algorithm requires that our Bag be written to in parallel and the parallel versions must be “reduced” to a single entity when our algorithm reaches a sequential portion of code. The reducer was written in the Bag header file and is essentially a wrapper around the Bag class. This Bag reducer includes a cilk class called a Monoid (this is a Mathematical concept) with our static reduce function, which just merges the Bags from different processes into a single entity. A reducer is very efficient and has very little overhead. The overhead occurs during work-stealing processes, when one instance of a Bag now has multiple copies on multiple processes and the most current version must be reduced to the original memory location, destroying the outdated Bags. In our Bag reducer wrapper class, we use the only data member, a cilk::reducer<Monoid>, to access the Bag's functions. All this does is implement the functionality of the Bag as stated previously, abstracting away the problem of data races.

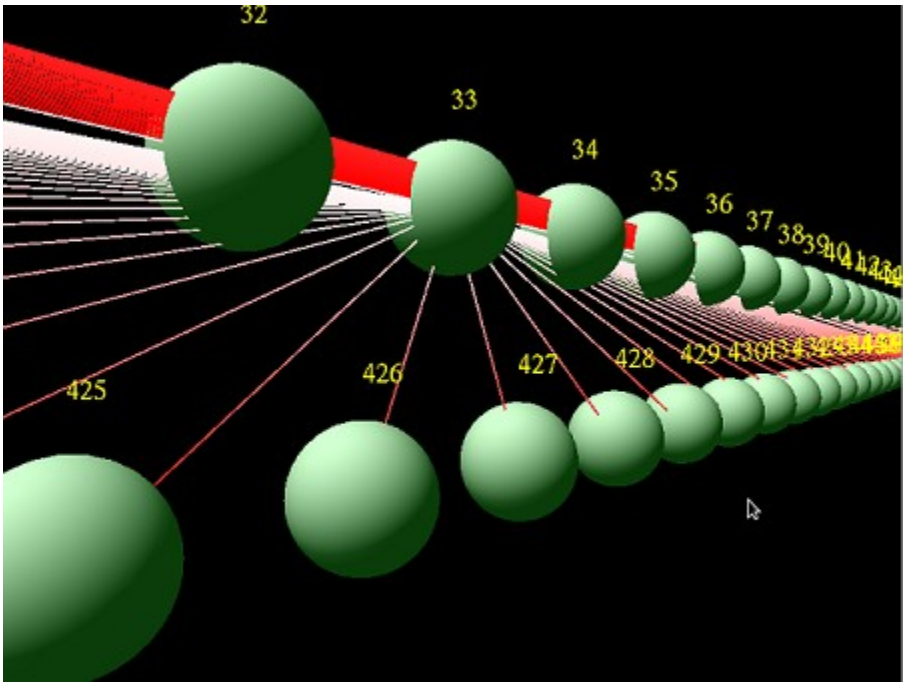
3D Graph Viewer:

Our 3D graph viewer turned out surprisingly well, as it depicts the graphs read in from any edge file with useful information and offers a different way to view and think about the file. When starting the project, looking at an edge file was completely useless when there were more than 10 or 20 edges. It just became a huge, incomprehensible file with large numbers. But when we ran it with our graph viewer, we were easily able to understand how the graph was

represented and how it should be represented in memory. This program was created with OpenGL and Professor Gilbert's code to read in an edge file. Window creation was handled by GLUT (GL Utilities Toolkit). At first, each node was represented by a 2D disk created with the GLU (GL Utilities) quadric library, and edges were drawn as solid red lines. Eventually, we updated the program to add more clarity by drawing the nodes as 3D spheres using GLU quadrics once again, and drawing the tail of an edge as a white vertex, and the head of an edge as a red vertex. This gradient on edges provided an easier way to tell how vertices were connected, and where edges begin and end. After this, we decided to use GLUT bitmap fonts to identify the nodes so that we could tell which vertex was which. We solved this problem by assigning an ID variable to each node, converting this label to a string, and feeding it to the render call for a GLUT bitmap font. We then realized our graph viewer was pretty static, as the user couldn't move around or zoom into the graph. To solve this problem, we added keyboard input so the arrow keys move the camera around accordingly to the x and y-axis, 'W' advances the camera forward into the z-axis (in this case, it would be the negative z-axis), 'S' retreats the camera back from the graph, and 'A' and 'D' move the camera along the x-axis but does not update the look-at vector, therefore giving the user a better depth of the 3D environment, since you can effectively rotate the camera around the y-axis. The last step to make this a nice looking program was to add smooth shading and lighting to the graph. Luckily the GLU quadric library provides a nice function for computing normals on spheres, so those calculations were already done for us. Also, OpenGL easily allows the programmer to add lighting to the scene, which in turn takes care of the shading for the sphere. After this, we felt our graph viewer was as effective as it could be, also balanced with a nice front end for usage. However, one thing that disappointed us was the fact that we could not compile our graph viewer with Cilk++. GLUT takes in a pointer to a function in the form of void* to handle its OS calls, such as glutKeyboardFunc(void(*)(unsigned char, int, int). However, when compiling with Cilk++, function signatures are changed to void (cilk*)(unsigned char, int, int). This broke the GLUT system calls, therefore rendering GLUT useless. This was disappointing since we wanted to run the OpenGL render calls on one thread, and use the rest of the threads to perform the BFS. We thought this would be an interesting experiment to see how OpenGL render calls impact system performance, and more importantly how it would impact our BFS algorithm.



This graph was produced from the small graph file “sample.txt” that was provided by Professor Gilbert.



This graph was produced from the small RMAT graph file “tinyrmat.txt” that was provided by Professor Gilbert. This graph helped show how many edges some vertices could have.

Different Implementations:

1) Unoptimized STL List with predefined Append List Reducer:

This was our first implementation of a BFS. We searched the Cilk include directory and discovered that there was a predefined list reducer made for the push_back function on STL lists. We eventually got our algorithm to run both sequentially and in parallel, only to discover it was painfully slow. Professor Gilbert's code ran in 0.005 seconds on a 130,000 edge graph, whereas ours ran in 11.2 seconds. After discovering this, we both agreed that this was far too slow and needed to find a way to do a faster implementation.

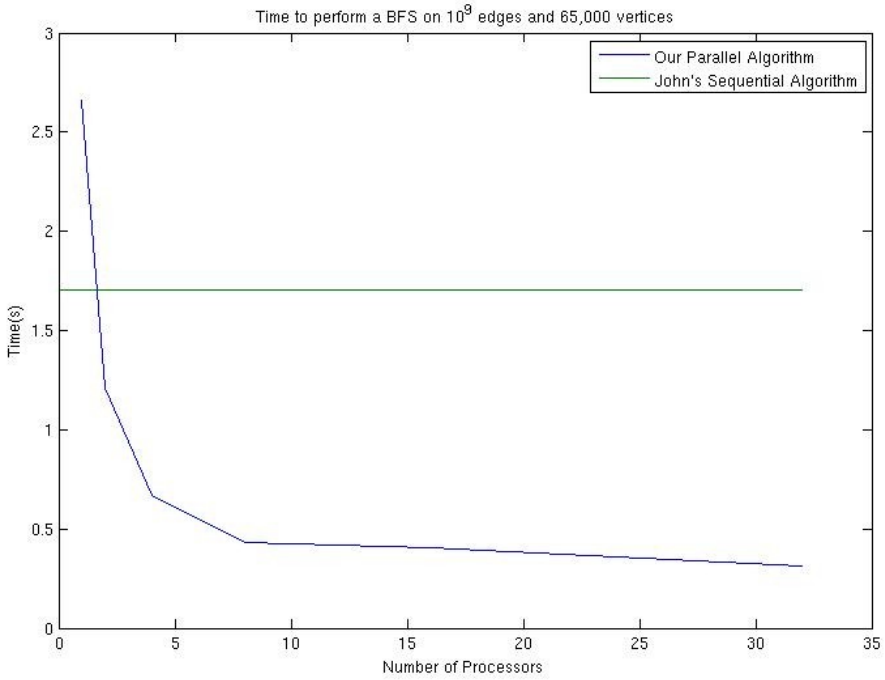
2) Optimized STL List with predefined Append List Reducer:

Our next step was to go through our code and find any unnecessary copy calls on the STL list, or any other steps that could be reduced to a smaller amount of operations. We found several copy constructor calls for STL lists and deleted those. Also, we realized we were searching sequentially through our list, instead of just calling the pop_back() function, which runs in O(1) time. After cleaning up our code, our runtime was reduced from 11.2 seconds to 0.1 seconds. It was a huge difference, but with a big flaw. It was still slower than Professor Gilbert's sequential code, and the more processors we added the slower it would become. We were getting the opposite of speedup, so we decided once again that we needed to go with a different implementation.

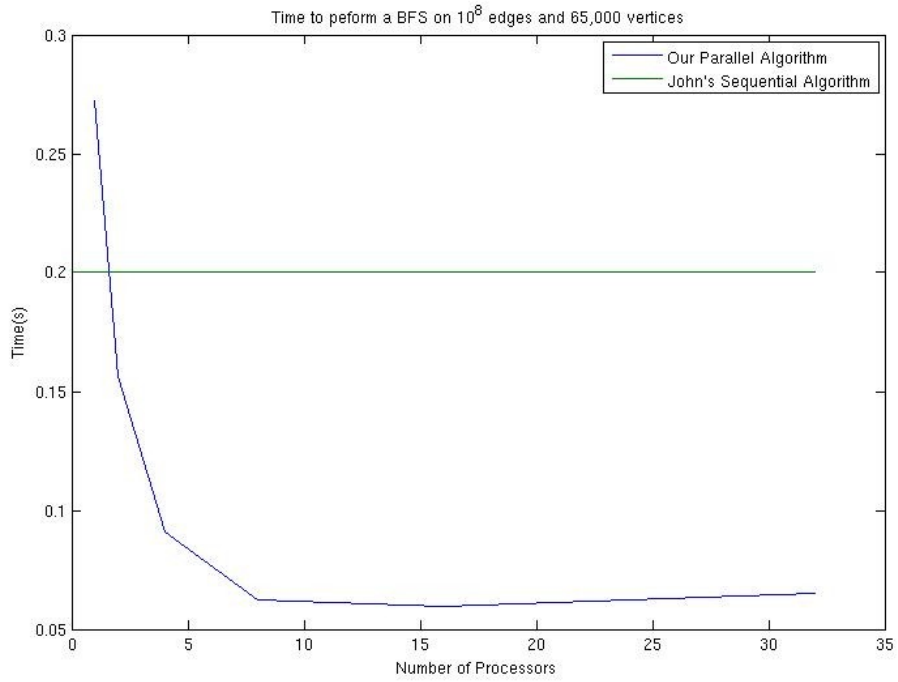
3) Two predefined Append List Reducer hyperobjects:

We decided to go with one last approach before attempting at creating the Bag data structure. Instead of using a data structure that was a wrapper for an STL list, we created two hyperobjects that held the current vertices being explored and the current parents of the vertices being explored. In this way, we thought it would parallelize very well since we could simultaneously explore all the neighbors of a node, and keep track of several nodes at once. Unfortunately, this was not the case since hyperobjects are very slow. This implementation was by far the worst, as it took 90.52 seconds to explore 130,000 edges. Obviously we had to scrap this idea and create the Bag data structure if we wanted to get parallel efficiency.

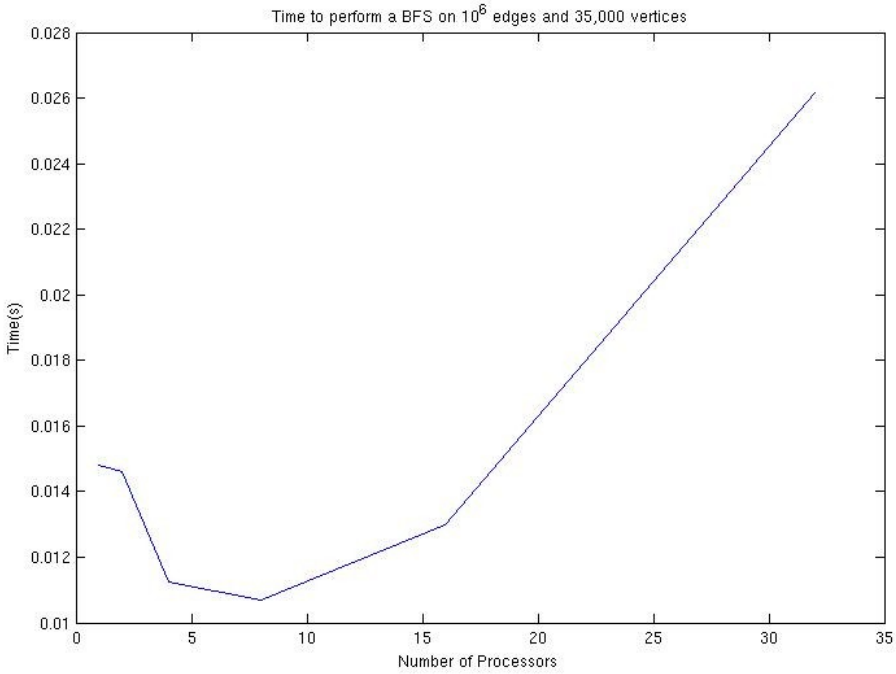
Performance:



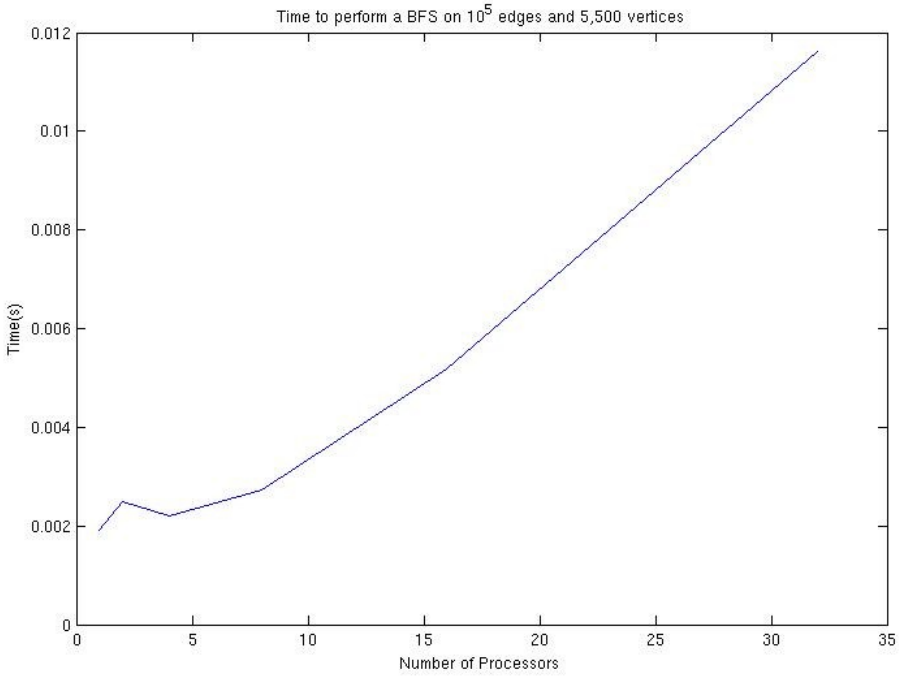
This graph represents the time to run a BFS on one billion edges for 1, 2, 4, 8, 16, and 32 processors. As you can see, the more processors added, the quicker the algorithm runs. This graph has the largest speedup on 32 processors, showing that the larger the dataset, the more efficient the parallel algorithm is with the maximum amount of processors on triton. The line at 1.7 seconds is the time it took for the optimized sequential code to run.



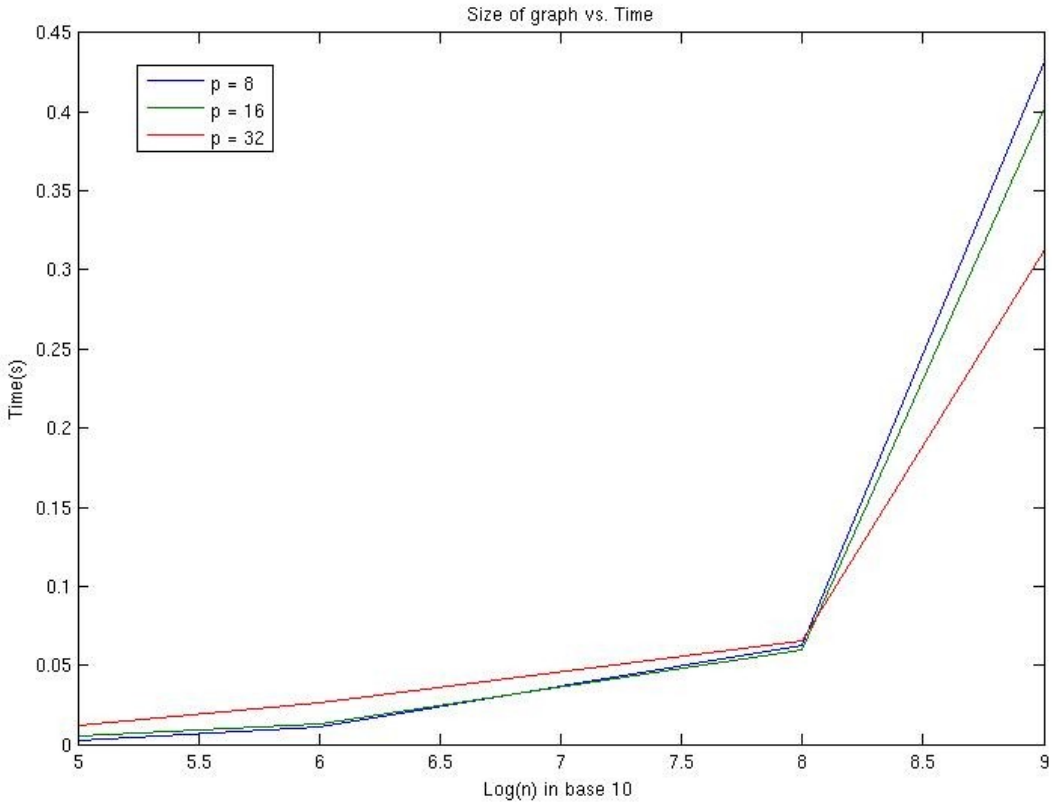
This graph depicts the same data as the above graph, except the parallel BFS was run on 10^8 edges instead of 10^9. As you can see, we acquire parallel speedup, except if you look closely the 16 processor program ran slightly faster than the 32 processor program. This is probably due to Cilk++ overhead and not having enough data to parallelize efficiently.



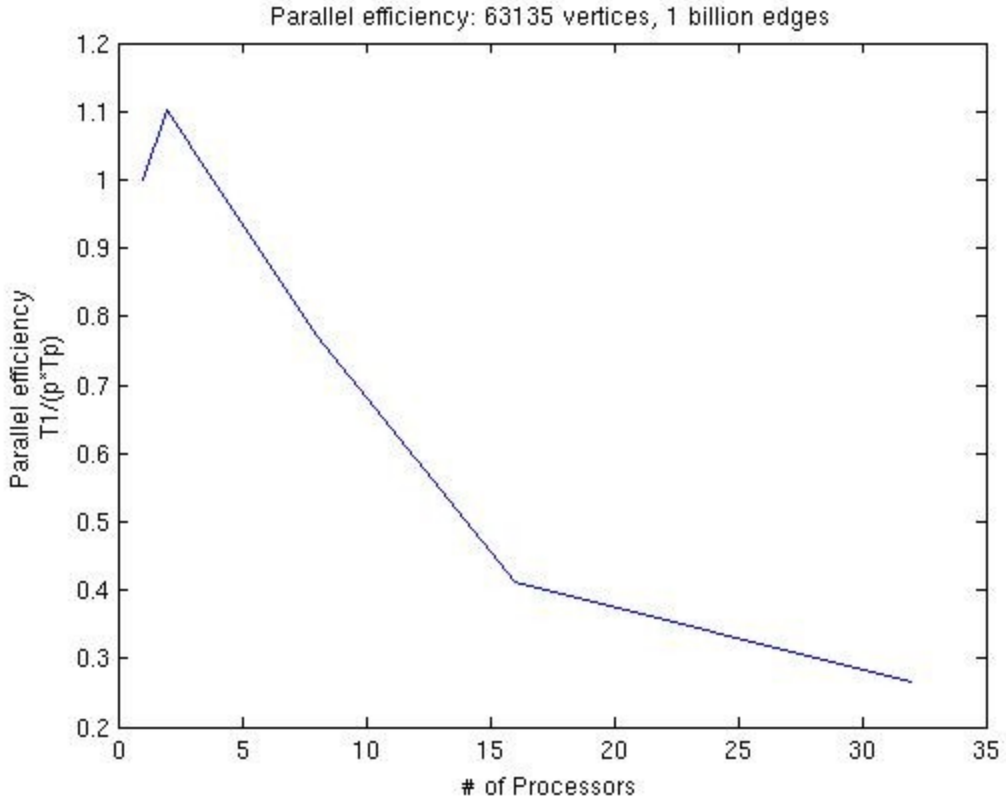
This graph depicts the same data set as the above graph, except on 10^6 edges. This graph shows how our algorithm does not achieve its full potential until having a larger data set. This data set is too small to parallelize efficiently in our algorithm, which is why the 32 processor program runs so slow.



This graph shows almost the same exact behavior as the former graph, except the parallel BFS is run on 10^5 edges. Once again, Cilk++ overhead dominates the runtime, and slows our program down considerably when we add more processors.



This graph represents how the size of the data set affects the time of our parallel BFS algorithm. The smaller data sets run faster with fewer processors. This is because the larger data sets are still doing more than an efficient amount of computations per process. All the data sets appear to converge at eight processors. This is the turning point for our algorithm and when larger data sets start to run faster and smaller data sets begin to have a large amount of overhead compared to the number of computations that occur.



This is the parallel efficiency for our parallel BFS algorithm on our largest data set of 1 billion edges for using 1, 2, 4, 8, 16, and 32 processors. We achieve super linear speed up at 2 processors with almost perfect linear speed up at 4 processors. After this, the parallel efficiency quickly declines. This is most likely caused by comparing our parallel efficiency with the sequential version of our parallel algorithm (that contains a lot of overhead with the type of data structure and cilk reducer we use that is not needed sequentially) instead of comparing our code to the optimized sequential version using a queue.

Conclusion:

Parallelizing the BFS algorithm was quite challenging. Our initial implementations using pre-made hyperobjects and STL list reducers were either too slow or performed inverse speed up (proportionally, the more processors used, the more time the algorithm took). The Bag data structure using Pennants was an excellent choice for parallelizing this algorithm due to its worst case runtime bounds on its functions and the simplicity of the custom reducer that it needed. With this implementation, we observed excellent speedup and parallel efficiency. The larger the data set we used, the more efficient our program ran using the maximum amount of processors (32), which means our program scales excellently. In the end, we tested TEPS (Traversed edges per second) on a graph that traversed ~500 million edges on 32 processors and we received 1,572,160,924 TEPS, putting triton at number 26 on the Graph500 list.