Parallel BFS

I’ve always been struggling with algorithms such as BFS that came up in all the past courses I’ve studied and I was not able to properly implement it and make it work (I don't think my A2 version works….) So when the final project opportunity came, I was determined to actually implement a working function of BFS but not only that, also make it parallel with the things I’ve learnt in “fun”par :)

BFS is just an algorithm to explore the vertices layer by layer and there is an opportunity for this process to be done in parallel.

The main idea of my parallel BFS is to make the function expand the frontier in parallel and merge it together at the end using concurrent data structures.

I came across “flurry” which provides a concurrent hashmap and hashset that I needed and it was also a crate from Java's version of concurrent hashmap. I tried testing it out and it was working completely fine. However, when I try to use it in my version of parallel bfs, it does not work. Because the parallelism I was trying to do was in the “expand” function that takes in an input of a regular std::collection HashSet/HashMap and also returns a std::collection HashSet/HashMap. The fact that I needed it to be a std::collection version because I needed the rayon parallel iterator par_iter which was the function I learnt in the lectures and I didnt know how to use other parallelization methods…. And because of this, “flurry” doesnt work with my code because it doesnt support “rayon”. Therefore, I had to find a new dependency.

I then took a look at Arc and Mutex but was turned away because the fact that I’m using mutexes would only mean that the code I’m running would be stalled or be waiting such that only one thread is allowed in the bathroom one at a time and it would then become a sequential piece of code.

After browsing through for a few days, I came across “DashMap” which supports the extensive feature of “rayon” and allowed me to use par_iter() while also using concurrent data structures!

Par_bfs (“expand” portion) was then implemented by taking input a regular std::collection hashset/hashmap and then converting to dashset and dashmap for local variables (frontier, visited nodes, parent and distance) and returning a regular std::collection hashset and hashmap at the end of the function.

After the code is done and I’ve tested that it works, I move on to benchmark tests. Initially, I was confused why my parallel version of code wasn't faster than the sequential version. After messaging you (Ajarn), I was enlightened by that parallelism != faster code!!!!!

I then tried parallelizing different portions of the code and tested the results, there are two other parts of the code that could be parallelized. One is removing visited nodes from the new frontier and the other part is when iterating the new frontier to update the distance hashmap. I’ve noticed that the only part that could benefit from parallelization is the part when removing visited nodes from the new frontier. Only this part made the code run faster. Parallelizing the update of the distance hashmap only slowed it down so I revert it back to the normal iter().

During benchmark tests, I needed to randomly generate a graph with even MORE number of nodes and edges! I then remembered that we used to have a function that randomly generates vectors in Lecture19 about parallelization techniques so I went back to make use of those codes. With several tests, the magic number for the number of nodes seems to be ~180. Which means that if the number of nodes is less than 180 (<180), you are better off using the normal_bfs version as it will be faster than the parallel version. Beyond 180, the parallel version seems to be faster :)

All in all, it was a “fun”par experience :)

THE END