Parallelization of Graph Algorithm in Java

By Group 11. GitHub: https://github.com/tshi206/Softeng751_Project_Group_11

Introduction

The objective of this project is to choose three graph algorithms and employ parallelization techniques in order to increase their computational efficiency and performance. As result, the selected algorithms comprise the following: (1) bidirectional breadth-first search (BBFS for short); (2) iterative deepening depth-first search (i.e., IDDFS); (3) Floyd Warshall (hereinafter abbreviated as FW). The remaining sections will be presented accordingly.

Bidirectional Breadth-First Search

Overview

BBFS is a variation of Breadth-First Search (BFS) where its purpose is to find the shortest path between two nodes in a Directed Acyclic Graph (DAG) with positively weighted nodes and/or edges. The underlying procedure can be described as simple as starting from both the source node and the sink node simultaneously and meeting somewhere in the middle of the search space, and thus, the shortest path is yielded according to the calculation and aggregation of costs from both sides. This algorithm is proven to be complete. Its optimality will be true if and only if one of the following holds: (1) the graph is uniformly weighted; (2) the heuristic employed is admissible. The time complexity of this algorithm is: $O(b^{d/2})$ where b is the effective branching factor and d is the distance from the starting node to the destination[1][2]. Its concrete implementation in this project is based on the algorithm of bidirectional Dijkstra without additional heuristics (i.e., bidirectional greedy search). This allows a heuristic value of h(x) = 0 at all time. Thereby, the heuristic is admissible and the algorithm is both complete and optimal. Furthermore, the parallelization framework employed by this project is @PT together with PARCutils. Additionally, the implementation of BBFS assumes the input graph consists of only a single source and a single sink and they will be the start and the destination. Another assumption is all weights are positive.

Implementation

The sequential implementation follows the standard algorithm using Cardinality Criterion[3][4][5] as the growing scheme of the frontiers, and the stopping criterion in which the heads of both

[1] (2015, November 25). algorithm - Time Complexity - Bidirectional Dijkstra - Stack Overflow. Retrieved May 14, 2018, from https://stackoverflow.com/questions/33889143/time-complexity-bidirectional-dijkstra
[2] (n.d.). Bidirectional search - Wikipedia. Retrieved May 14, 2018, from https://en.wikipedia.org/wiki/Bidirectional_search
[3] (n.d.). BI-DIRECTIONAL AND HEURISTIC SEARCH IN PATH PROBLEMS. Retrieved May 14, 2018, from http://www.slac.stanford.edu/pubs/slacreports/reports04/slac-r-104.pdf
[4] (n.d.). A Brief History and Recent Achievements in Bidirectional Search. Retrieved May 14, 2018, from http://www.bgu.ac.il/~felner/2018/SMTBD.pdf
[5] (n.d.). Bidirectional Search That Is Guaranteed to Meet in the Middle. Retrieved May 14, 2018, from https://webdocs.cs.ualberta.ca/~holte/Publications/MM-AAAI2016.pdf

binary heaps in the frontiers are added and compared against the current shortest cost in order to detect the termination constraint[6]. The implementation of Dijkstra algorithm is based on the conventional approach by using a sorted collection of nodes as the open list (aka, the frontier)[7]. The parallel implementation applies the identical growing scheme. The stopping criterion remains the same in both implementations.

In total, there are four versions implemented in order to undertake a meaningful analysis. The first version is the sequential implementation and the rest of them are variants differentiated by their parallelization approach. The following list will briefly go through each version in terms of their implementations:

1. Sequential BBFS:
    a. Prepare the data structures, e.g., maps for costs from source and sink, lists for both frontiers.
    b. Start with the source node, run the conventional Dijkstra algorithm for the head of the heap. Pop the head out of the frontier and check if its cost has been determined by the sink side. If true then update the shortest cost so far otherwise pass the current iteration.
    c. Switch to the sink side, run the Dijkstra algorithm for the head of the heap. Pop the head out and check if its cost has been determined by the opposite side. If true then update the shortest cost so far otherwise pass.
    d. Repeat the step b and c until the stopping criterion is met.
2. Parallel BBFS with inner loop parallelization:
   The outer loop stays the same therefore the implementation is almost identical to the sequential. The parallelization is carried out in the inner loop in which branching to the child nodes is parallelized.
3. Parallel BBFS with parallelized outer loop and two worker threads:
   Two threads run simultaneously from both the source and the sink sides. The inner loop (i.e., branching) is not parallelized (identical to the sequential version).
4. Parallel BBFS with parallelized outer loop and a dynamic number of worker threads:
   Threads are executed simultaneously from both sides. The exact number of threads is determined by the number of logical processors of the underlying system at runtime. Both sides get allocated to an even number of workers (≈ the total number of workers divided by two). The source frontiers and sink frontiers are partitioned by the number of workers on each side and assigned to each of them accordingly. Cost maps are distributed as private copies among all workers.

In the following sections, each version will be referred to in terms of their index number from the list above, for example, the sequential version will be referred to as version 1 for simplicity. The same reference scheme applies to the other versions as well.

---

[6] (n.d.). EPP Shortest Path Algorithms - Princeton CS - Princeton University. Retrieved May 14, 2018, from http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf

[7] (n.d.). What is the complexity of Dijkstra's algorithm? - Quora. Retrieved May 14, 2018, from https://www.quora.com/What-is-the-complexity-of-Dijkstras-algorithm

Evaluation

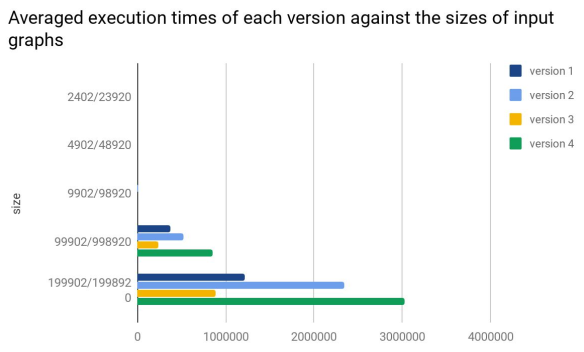
Hypothesis:

Presumably, the version 3 should be twice faster than the version 1. The version 4 will reduce the execution time of the version 1 where the execution time of the version 1 should be effectively divided by the number of available logical processors in version 4.

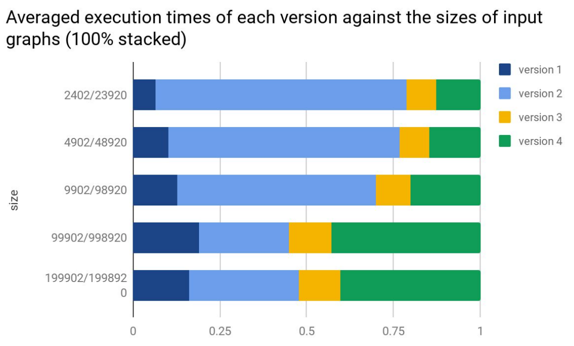
Experiments:

The execution times of each version were sampled by five trials for each of the five different sizes of input graphs and averaged accordingly. The experiment was undertaken using a home desktop PC equipped with 16 GB RAM, 4 cores, 8 logical processors, 1 memory socket (thereby, shared memory architecture), 3.40 GHz base speed, virtualized memory space, and 256 KB, 1.0 MB, 8.0 MB as the sizes of three levels of caches respectively. Moreover, the program under test was ensured to be the only user-level process in addition to kernel processes. Besides, all the input graphs are made to have a constant branching factor of 10, in order to retain the consistency throughout the experiments. Figure 1 depicts the results of the experiments.

Size of input graph (in nodes/edges)	Execution time of version 1 (in milliseconds)	Execution time of version 2 (in milliseconds)	Execution time of version 3 (in milliseconds)	Execution time of version 4 (in milliseconds)
2402/23920	174.0581	1896.354	221.7369	338.3587
4902/48920	552.6452	3548.729	461.7756	792.972
9902/98920	1734.902	7728.748	1366.931	2713.644
99902/998920	375245.9	517140.4	241970.1	847917.7
199902/1998920	1216930	2339416	889543.2	3021030



(Figure 1 - non-stacked bar chart)



(Figure 2 - 100% stacked bar chart)

Analysis

Consequently, the hypothesis turns out to be false in terms of the experimental results. As result, only the version 3 exhibits the speedup gained from the parallelization while the other two parallel versions both fail to demonstrate the benefit of parallelization. In terms of the experiments run, the speedup starts to emerge as the size of the input graph approaching 10,000 nodes and 100,000 edges.

The version 2 is a naive implementation of parallelisation of the version 1 in which the inner loop is parallelized because of the lack of inter-loop dependences. Therefore, the version 2 was considered to be a counterexample in the first place. The version 3 and 4 are inspired by this paper[8][9] which can be found in this book[10]. It is interesting to see the fourth version end up being the worst version in terms of its performance. The analysis of each parallel version will be illustrated in the following subsections:

- **The version 2**: It is a good example to demonstrate how the parallelization benefit is counteracted by the execution overheads. The reason for it being slower than the version 1 (sequential) is because: (1) the individual execution times for tasks branching on the child nodes and the associated computations are very small; (2) the time taken by generating new threads for each branching tasks and the associated management operations like context-switching is reasonably large in contrast to the time taken by the actual tasks. Hence, the overhead outweighs the execution time of subtasks. Thus, the second version is slower than the version 1.
- **The version 3**: It is the natural implementation of the parallelization for the sequential version in the sense that it utilizes the inherent parallelizability in which both sides of the graph can start at the same time and execute in parallel. Because in the sequential version, there are already two sets of data structures prepared for both sides, in the version 3, those data structures are re-used and thus there is no extra overhead induced by creating new datasets or copying old data structures. Due to the simple fact that only two threads are spawn, the threading overhead is minimised. Theoretically, the optimal speedup will be 50% of the sequential time. However, because of the overhead induced by the concurrent access to the shared data structures (e.g., maps), some forms of synchronization like locking occur behind the scene resulting in the actual speedup (roughly 25 ~ 30%) being less than the hypothetical value.
- **The version 4:** The reason for this version being worse than the sequential version by over 50% in terms of execution is because: (1) in order to prepare private datasets for each pair of source tasks and sink tasks, the overhead is imposed on copying data structures for each pair and this overhead grows as the input size increases, and this also means the scalability no longer conforms to Amdahl's law as the sequential length is not a constant anymore; (2) because the actual size of runtime datasets is the sequential dataset multiplied by the number of tasks of one side (whether it is the source side or the sink side does not matter), the version ends up dumping a huge amount of objects into the memory and thus the intensive memory access (reads, writes, and implicit data exchange among threads) becomes the bottleneck of the

[8] (2017, December 19). Parallel Bidirectional Dijkstra's Shortest Path Algorithm. - ResearchGate. Retrieved May 14, 2018, from https://www.researchgate.net/publication/221278140_Parallel_Bidirectional_Dijkstra's_Shortest_Path_Algorithm

[9] (2011, August 13). Parallel Bidirectional Dijkstra's Shortest Path Algorithm. Retrieved May 14, 2018, from https://dl.acm.org/citation.cfm?id=1940629

[10] (n.d.). Databases and Information Systems VI: Selected ... - Google Books. Retrieved May 14, 2018, from https://books.google.com/books/about/Databases_and_Information_Systems_VI.html?id=7fYS29re860C

parallelization; (3) on the algorithm's point of view, any implementation with more than two worker threads will not be benefited by multithreading or multitasking, given the input graph is not a form of trees, simply due to the fact that individual search spaces will overlap and even heavily overlap if the input graph is big enough. For all the aforesaid reasons, in this version, the overhead caused by the parallelization diminishes, counteracts, and eventually eliminates and even negated the theoretical speedup.

Conclusion

In conclusion, BBFS is parallelizable in the sense that utilizing two worker threads enables a reasonable speedup. However, further parallelization where more than two worker threads are employed will adversely impact the performance due to the inherent growth of sequential time during the data preparation phase. Furthermore, the algorithm will not benefit from multitasking in the branching phase because the computational resource spent in and the time taken by each task on child nodes is relatively small and tiny enough to be ignored in contrast to the overhead. Thereby, the version 3 is chosen to be the final version of the parallelization for the BBFS implementation.

Iterative Deepening Depth-First Search

Overview

IDDFS is used to find the shortest path between two nodes in a Directed Acyclic Graph (DAG) without weights. IDDFS is a depth-limited version of depth-first search (DFS), which run repeatedly with increasing depth limits until the goal is found. IDDFS is equivalent to breadth-first search, but uses much less memory. On each iteration, it visits the nodes in the graph in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first search. (1) when the BFS algorithm will cause memory overflow (2) when the shortest path is needed. The time complexity of this algorithm is $O(b^d)$ where b is the branching factor and d is the depth of the target node. The parallelization framework employed by this project is PARCutils. Additionally, this algorithm does not like the other two algorithms of this project using weights for edges and vertices. It assumes the weight of edges are all same. Another assumption is all weights are positive.

Implementation

IDDFS is a depth-limited version of DFS. Thus the implementation is similar with the implementation of DFS. DFS algorithm has two common implementation. One implementation use the recursive method and another implementation use stack to store the status. In this project, the implementation with stack is chosen, because for parallelization stack as a collection can be accessed by different thread, which means it will be easier to be paralyzed.

The sequential implementation of IDDFS:

1. This algorithm will visit one of the nodes of current node until the current node have no children which haven't been visited or the depth of current node is greater than the

depth limitation. Then the algorithm will go back to the parent of current node. This process will repeat until all reachable node are visited.

2. The depth limitation will increase if last iteration didn't found the target node. Then the algorithm will do the DFS with the increased depth limitation until the target node is found

The parallel implementation of IDDFS:

1. One approach to parallelise this algorithm is using multiple threads to run the DFS with different depth
2. Another approach is to parallelise the DFS algorithm. By using multiple threads to access the stack together and process the data together.

Evaluation

Hypothesis:

The first approach of parallelisation should be much faster than the sequential implementation.

Experiments:

The experiment was undertaken using a laptop with a Intel i7-3632QM CPU, with 4 Cores, 8 Threads, processors frequency of 2.2 GHZ, 6MB cache and 8GB RAM. Moreover, the program under test was ensured to be the only user-level process in addition to kernel processes. The results of the experiments are illustrated by the table below。

Graph(number of nodes)	Sequential version time taken (ms)	Parallel version time taken (ms)
80	10	22
392	61	59
902	103	99
2402	420	228
3986	1305	538
4902	1450	493
9902	6885	1535

Analysis

According to the experimental results, only when the graph is very small (less than 100 nodes), the parallel version a little slower than sequential version because giant grain task can reduce the overheads of creating new threads. With the increasing of graph size, the parallel version becomes very efficient. When the graph big enough the time taken for parallel version 4 time faster than sequential version. For this experiment environment there are 4 processor in the laptop. Therefore, this parallelisation is very successful.

Conclusion

In conclusion, IDDFS is parallelizable in the sense that utilizing multiple worker threads enables a reasonable speedup. For the first approach, because there are no dependency between each iterations with different depth, those threads can process those iterations at the same time. Such tasks are giant grain. Thus the overhead will not cost too much time. For the second approach, the algorithm will not benefit from multitasking, because each task is relatively small and overhead will cost a lot compare with the time saved by parallel. Thereby, the first approach is chosen to be the final version of the parallelization for the IDDFS implementation.

Floyd Warshall

Overview

The Floyd–Warshall algorithm is an example of dynamic algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles), and was published in its currently recognized form by Robert Floyd in 1962. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertices. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal, thus it is able to do this with O(V^3) comparisons in a graph. However, very large graphs may take a very long time to do the O(V^3) comparisons in sequential. Therefore we aimed at developing the parallel version of Floyd Warshall algorithm to improve its efficiency.

Implementation

The implementation of Floyd Warshall algorithm in this project is based on these assumptions: (1) all the graphs are oriented; (2) the edges of the graphs are non-uniformly weighted; (3) all the weights of edges are always positive for all the graphs.

The sequential implementation considers a graph G with vertices V numbered 1 through N. Further consider a function shortestPath(i,j,k) that returns the shortest possible path from i to j using vertices only from the set {1,2,...,k} as intermediate points along the way. In the algorithm, our goal is to find the shortest path from each i to each j using only vertices in {1,2,...,N}. For each of these pairs of vertices, the shortestPath(i,j,k) could be either a path that does not go through k or a path that does go through k(from i to k plus from k to j, both only using intermediate vertices in {1,...,k-1}).

The parallel version is implemented by @PT, based on PARCutils. Floyd-warshall algorithm can be parallelised in two methods: (1) for each task, go through k from the set {0,1,...,k-1} to and for each k(k is the outer loop), and for each matrix[i][j] get shortestPath(i,j,k), which is the task that can be parallelised. (2) In each task, go through i from the set {0,1,2...,i-1} and for each i, go through j from the set {0,1,2,...,j-1}, and then get shortestPath(i,j,k) for each k(k is the inner loop). Theoretically, the method 2 can parallel both loops of i and j, which is more efficient than method 1, because the latter one can only parallel the loop for intermediate points k. However, Floyd-Warshall algorithm is basically a greedy algorithm and dynamic programming, in which k should always be in the stage of dynamic programming. To ensure that the result matrix is always correct, k should be in the outer loop. According to this, method 1 is the appropriate way to do the parallelisation. The following list will briefly go through each version in terms of their implementations:

1. Sequential Floyd-Warshall algorithm
    a. Initialize the matrix of path(i,j)
    b. The inner double loop of i and j go through the whole matrix and get the value of the path from i to j
    c. The outer loop k of set{0,1…,k} go through the whole matrix to compare the value of path(i,j) and path(i,k)+path(k,j). If path(i,j) > path(i,k)+path(k,j), then path(i,j) = path(i,k+path(j,j)
2. Parallel Floyd-Warshall algorithm method 1
    a. Initialize the matrix of path(i,j)
    b. In parallelSearch() method, add @Future annotation to a list. Put list in the loop for k and the result of processSearch(k) to the list in every iteration.
    c. In processSearch(int k) method, iterate the two inner loops for i and j, then do the same as sequential search.

Evaluation

Hypothesis:

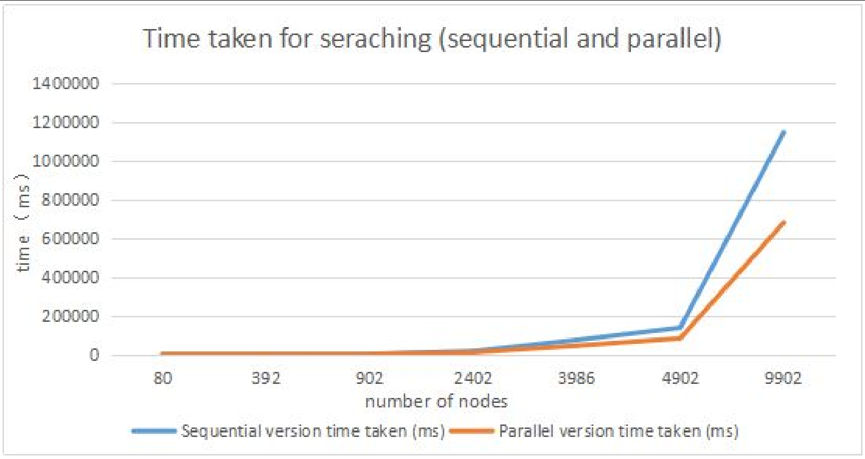
Ideally, the parallel version should be twice faster than the sequential version. However in parallel version, there are overheads for creating new threads and some other overheads. The improvement for small size of graphs may not be significant.

Experiments:

The experiment was undertaken using a laptop with a Intel i7-6700HQ CPU, with 4 Cores, 8 Threads, processors frequency of 3.5 GHZ and 6MB cache. Moreover, the program under test was ensured to be the only user-level process in addition to kernel processes. Small graphs are processed multiple times and get the average value, however because Floyd-Warshall is O(n^3) algorithm that large graphs could take an extremely long time. Graphs with large number of nodes are only processed once in this experiment. The results of the experiments are illustrated by the table below。

Graph(number of nodes)	Sequential version time taken (ms)	Parallel version time taken (ms)
80	≈0	47
392	94	110
902	906	547
2402	16175	9985

3986	73459	44215
4902	136745	81860
9902	1143389	678874



（Figure 3 - Floyd algorithm）

Analysis

According to the experimental results, it is obviously that when the graph is small (less than 400 nodes), the parallel version is even slower than sequential version because of the overheads of creating new threads. With the increasing of graph size, the parallel version becomes more and more efficient and when the graph is large enough, the time taken for parallel version is around half of that of sequential version. Therefore, our hypothesis is proved to be correct.

Conclusion

In conclusion, Floyd-Warshall algorithm is parallelizable in the sense that utilizing two worker threads enables a reasonable speedup. However, since there is temporarily not any appropriate method to reduce the overheads of creating new threads. The parallel implementation of Floyd-Warshall algorithm is only suitable for graphs with large number of nodes.

Table of Contributions

Member	I/O	BBFS seq	BBFS par	IDDFS seq	IDDFS par	FW seq	FW par
Mason Shi	✓	✓	✓				
Baiwei Chen				✓	✓		
Tao Ge						✓	✓