Dijkstra's shortest path algorithm serial and parallel execution performance analysis

Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, Novica Nosovic

Department for Computing and Informatics, Faculty of Electrical Engineering, University of Sarajevo

anaidana@gmail.com, elmaars@gmail.com, nadira.j90@gmail.com, ilvana89@hotmail.com, elma.lagumdzija@gmail.com, novica.nosovic@etf.unsa.ba

Abstract - ***This article introduces the problem of parallelization of Dijkstra's algorithm, a well known algorithm for computing single-source shortest path in a graph. Dijkstra's algorithm can be applied to graphs with a various number of vertices and edges. Dijkstra's shortest path algorithm is implemented and presented, and the performances of its parallel and serial execution are compared. The algorithm implementation was parallelized using OpenMP (Open Multi-Processing) and OpenCL (Open Computing Language) standards. Its performances were measured on 4 different configurations, based on dual core and i5 processors. The experimental results prove that the parallel execution of the algorithm has good performances in terms of speed-up ratio, when compared to its serial execution. Finally, the results show that, because of Dijkstra's algorithm in itself is sequential, and difficult to parallelize, average speed-up ratio achieved by parallelization is only 10%. This proves to be a huge disadvantage of this algorithm, because its use is widespread, and enhancing its performance would have great effects in its many uses.***

Keywords - **Dijkstra algorithm; single source shortest path; parallel algorithm; OpenMP,** ***OpenCL***

I. INTRODUCTION

With the development of computer and information technology, the research about graph theory get a wide range of attention, and a variety of graph structures and algorithms have been proposed [2].

The shortest path algorithm is always a research hotspot in graph theory and it is the most basic algorithm. For example, shortest path algorithm is used to implement traffic engineering in IP networks and to improve Intelligent and Transportation Systems. But for that kind of algorithm it is very difficult to improve its performance.

Currently the algorithms for the serial shortest path optimization have reached the time limitation. Therefore the parallel computation is an efficient way to improve the performance. By putting some constraints on the data and taking the advantage of the hardware, the performance of the algorithms can be significantly improved.

In this article, firstly you can find a definition of the Dijkstra's algorithm for computing single-source shortest path in a graph, and then there is a parallel algorithm given to solve the shortest path problem of the single source graphs, where one implementation of single source shortest path algorithm executed on dual-core and i5 processor with OpenMP, and one with OpenCL. Finally we can find through experiments that parallel algorithm used in this article is has better performance than the serial one.

II. DIJKSTRA'S ALGORITHM

Dijkstra's algorithm finds the length of an shortest path between two vertices in a graph.

The algorithm works as follows:

1. Choose the source vertex
2. Define a set S of vertices, and initialise it to emptyset. As the algorithm progresses, the set S will store those vertices to which a shortest path has been found.
3. Label the source vertex with 0, and insert it into S.
4. Consider each vertex not in S connected by an edge from the newly inserted vertex. Label the vertex not in S with the label of the newly inserted vertex + the length of the edge.

But if the vertex not in S was already labeled, its new label will be min(label of newly inserted vertex + length of edge, old label)

5. Pick a vertex not in S with the smallest label, and add it to S.
6. Repeat from step 4, until the destination vertex is in S or there are no labeled vertices not in S.

If the destination is labeled, its label is the distance from source to destination. If it is not labeled, there is no path from the source to the destination.[6]

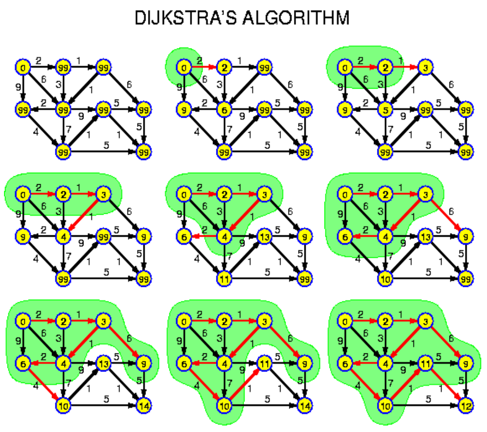


Figure 1. Example of Dijkstra's algorithm in steps

III. EXPLANATION OF SERIAL DIJKSTRA ALGORITHM

In this algorithm, two sets are maintained: D for all discovered nodes and U for undiscovered nodes. At first, D contains only the starting node s, use d(v) to denote the current estimated best distance from s to v, which is his neighbor. Initially all v except s has d(v)=infinity. Set D grows by adding the node with minimal estimated distance from U into D, and update the current distance of the neighbor nodes of the newly added node, and repeat this procedure until all nodes are in D.

IV. EXPLANATION OF PARALLEL DIJKSTRA ALGORITHM

There are two parallelization strategies - execute each of the n shortest path problems on a different processor (source partitioned), or use a parallel formulation of the shortest path problem to increase concurrency (source parallel). Using n processors, each processor Pi finds the shortest paths from vertex Vi to all other vertices by executing Dijkstra's sequential single-source shortest paths algorithm.
It requires no interprocess communication (provided that the adjacency matrix is replicated at all processes). In this article it is used formulation of Dijkstra’s algorithm using OpenMP. An OpenMP application begins with a single thread, the master thread. As the program executes, the application may encounter parallel regions in which the master thread creates thread teams (which include the master thread). At the end of a parallel region, the thread teams are stopped and the master thread continues execution. Since OpenMP is primarily a pragma or directive based standard,it can be easily combine serial & parallel code in a single source. By simply compiling with or without the /OpenMP compiler, it can turn OpenMP on or off. Code compiled without the /OpenMP simply ignores the OpenMP pragmas which allows original serial execution.

Figure 2. Part of code where OpenMP pragmas was used, wg represents weighted graph which contains weights of paths between nodes and vid.size() is size of vector with all nodes in graph

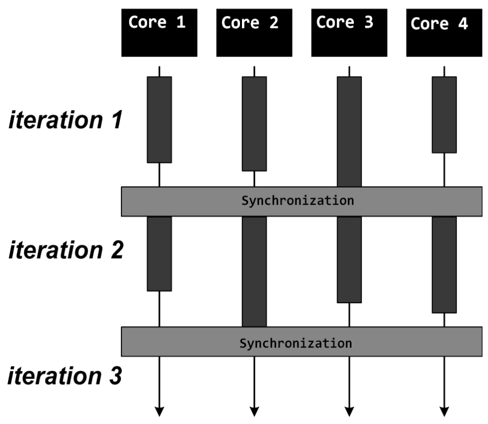


Figure 3. Example of iterations during execution of parallelized Dijkstra's algorithm on four processors using OpenMP

V. EXPLANATION OF IMPLEMENTATION OF DIJSKTRA'S ALGORITHM

The first step in running Dijkstra’s algorithm on GPU (Graphics Processing Unit) is to create a graph data structure that is efficiently accessible by the GPU. The graph is composed of vertices and edges that connect vertices together. Each edge has a weight value associated with it that typically measures some cost in traveling across that edge. In this case, the edge weights were determined by the curvature function as curvature values were minimalized in traveling across the mesh. Each execution of Dijkstra will output an array the size of the number of vertices in the graph with the total cost of the shortest distance from the source vertex to each vertex in the graph.
The first kernel (functions that execute on OpenCL devices) that is queued to OpenCL for each source vertex is simply responsible for initialization of buffers. This was done using a kernel rather than on the CPU to reduce the amount of data transferred between the CPU and GPU.
The algorithm itself is broken into two phases. This is necessary because synchronization isn’t possible outside of local work-groups in OpenCL, and this would be required to execute the kernel algorithm in a single phase. The first phase of the algorithm visits all vertices that have been marked and determines the cost to each neighbor. If the current cost plus the new edge weight is less than what is currently stored in function, then that new cost is stored for the vertex. The second phase of the algorithm checks to see if a smaller cost has been found for each vertex and, if so, marks it as needing visitation and updates the costArray.
In order to leverage multiple compute devices, the workload needs to be partitioned. The approach taken in this implementation is to partition the number of searches

across the available compute hardware. The application detects the number of GPU and CPU devices and splits the workload across the devices.
Each of the kernels is written to process one vertex at a time and it was implemented by workload portioning.
In the case CPU device is selected from the initial menu when execution starts, the OpenCL implementation will multithread the implementation across the available cores.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

The experimental environments for analyzing the performance of algorithms in this article were as follows:

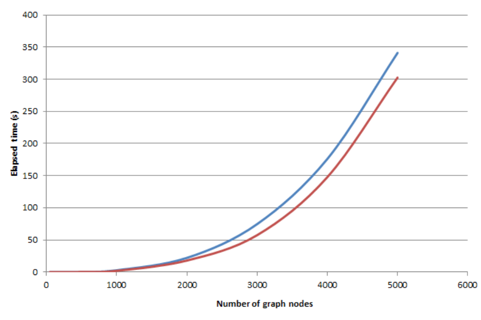
<u>Configuration 1:</u>
Processor: Intel(R) Core(TM) i5 CPUM460 @ 2.53GHz 2.53GHz
GPU: AMD Radeon HD 6500M/5600/5700 series

<u>Configuration 2:</u>
Processor: Intel(R) Pentium(R) Dual CPU T2390@ 1.86GHz 1.87GHz
GPU: Intel® GMA X3100

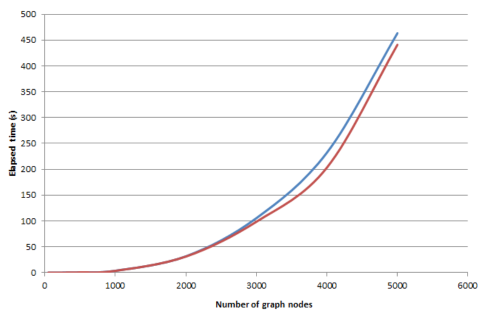
<u>Configuration 3:</u>
Processor: Intel Celeron Dual Core CPU T1500 @ 1.86GHz 1.86GHz
GPU: Intel GMA X3100

<u>Configuration 4:</u>
Processor: Intel(R) Core(TM)2 Duo CPU T5760 @ 1.80GHz
GPU: Mobile Intel(R) 965 Express Chipset Family

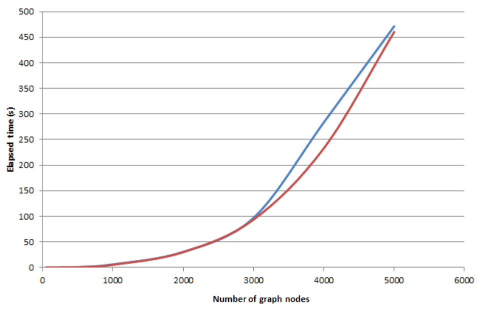
Execution speed was compared for those algorithms: sequential one, the parallel one, and the parallel one made for OpenCL.
They are all implemented and tasted on processors (dual core and i5). As it can be seen from the graphs on Figure 4., parallel version seems to perform better than the sequential one when the number of nodes becomes larger than 3000.



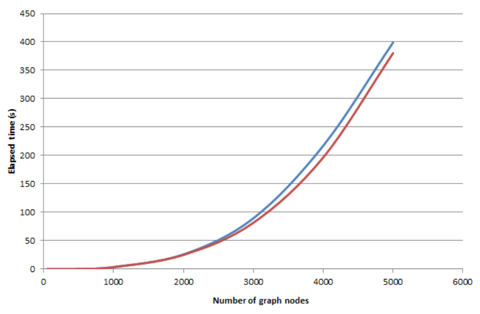
4.a



4.b



4.c



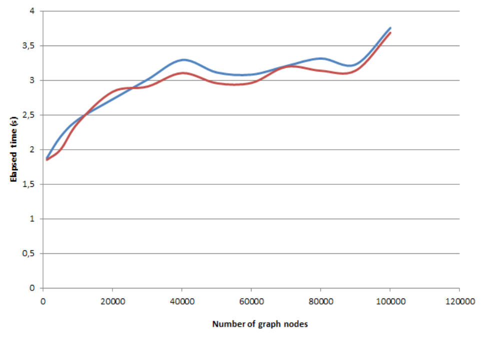
4.d

🟦 Serial execution
🟥 Parallel execution

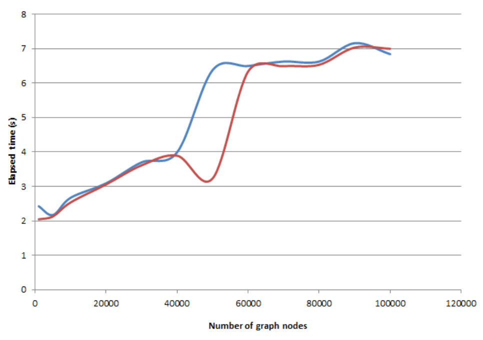
4.e

Figure 4. The comparison of execution time in algorithms (parallel and sequential) running on a different machines: 4.a: Configuration 1, 4.b: Configuration 2, 4.c: Configuration 3, 4.d: Confituration 4, 4.e: Legend

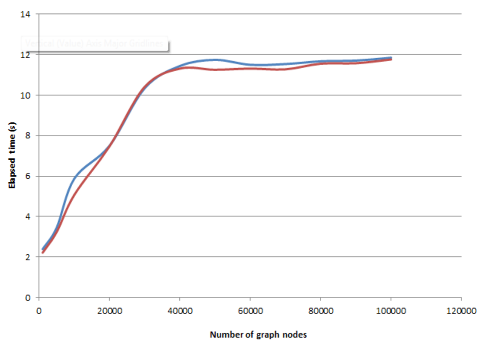
As can be seen from Figure 4, for weighted graphs, parallel algorithm has better performance than the sequential one, but these performance also depends on which machines was this tested, where the best performance were on i5 processor.



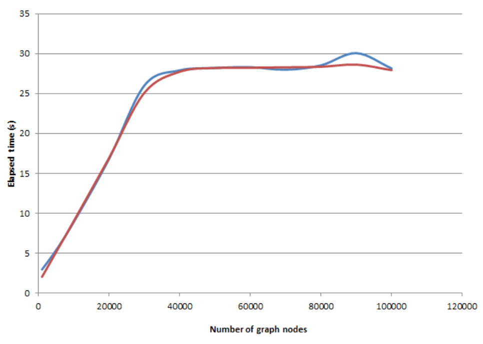
5.a



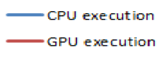
5.b



5.c



5.d



5.e

Figure 5. The comparison of execution time in algorithms (parallel one made for OpenCL and sequential one) on Configuration 1; 5.a: 5 edges per node, 5.b: 10 edges per node, 5.c: 20 edges per node, 5.d: 50 edges per node, 5.e: Legend

As can be seen from the collected performance data, the number of edges per node affect the performance. On figure 5. It can be seen that the performance on GPU are better than the one on CPU, but when the number of edges per node is bigger than 10, as shown on figure 7. and 8. the performances are almost the same. That means that for the different values of vertices the speed-up ratio of our parallel algorithm keep basically unchanged.

VII. CONCLUSION

Parallel programming is a very intricate, yet increasingly important task as we have entered the multicore era and more cores are made available to theprogrammer. Although separate applications or independent tasks within a single application can be easily mapped on multicore platforms, the sameis not true for applications that do not expose parallelism in a straightforward way. Dijkstra's algorithm is a challenging example of such an algorithmthat is difficult to accelerate when executed in a multithreaded fashion. It is very important to have in mind the two major issues inherent to Dijkstra'salgorithm: limited explicit parallelism and excessive synchronization. Dijkstra's algorithm does not readily lend itself to parallelization mainly because itrelies on priority queue. Because evaluating the vertex at the head of the priority queue may change the order of elements remaining within the priorityqueue, it is not necessarily correct or efficient to evaluate more than the head of the priority queue on each iteration.

There are several possible reasons why the more advanced parallel execution of Dijkstra's algorithm wasn't as fast as expected. One of the reasons could be that the code used may be inefficient. Another possibility is that, for a small amount of nodes, more time could be spent on parallelization and synchronization then it is spent on execution of code as sequential.

There are several areas in which this article could be extended with further work. Overcoming issues such as utilizing shared memory, avoiding branches, choosing appropriate data structures and designing algorithms correctly are the key to success with parallelization of Dijkstra's algorithm.

Some problems appear to be inherently sequential, and are difficult, if not impossible, to implement in parallel. Unfortunately, one of these problems is Dijkstra's algorithm. This is actually the main problem for the implementation of its parallelization.

VIII. REFERENCES

[1] Han Cao,Fei Wang, Xin Fang, Hong-lei Tu and Jun Shi, „OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis“, World Congress on Software Engineering, 2006.
[2] Qiang Peng, Yulian Yu and Wenhong Wei, „The Shortest Path Parallel Algorithm on Single Source Weighted Multi-level Graph“, 2009 Second International Workshop on Computer Science and Engineering, 2009.
[3] Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris, „Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm“ ,2009.
[4] Fang Zhou Lin, Nan Zhao, „Parallel Implementation of Dijkstra's Algorithm“ , pp COP5570, 1999.

[5] Kevin Kelley, Tao B. Schardl, „Parallel Single-Source Shortest Paths“, 2007.
[6] http://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/DijkstraAlgo.html, last modified on 29th October 2004.