Vol. 1, No. 1 | Jan – June 2017

A Fast Implementation of Minimum Spanning Tree Method and Applying it to Kruskal's and Prim's Algorithms

Badri Munier, Muhammad Aleem, Muhammad Arshad Islam, Muhammad Azhar Iqbal

Department of Computer Science, Capital University of Science and Technology, Islamabad

badar_lucky83@yahoo.com, aleem@cust.edu.pk, arshad.islam@cust.edu.pk, azhar@cust.edu.pk

Waqar Mehmood

Department of Computer Science, COMSATS Institute of Information Technology, Wah Cantt

drwaqar@ciitwah.edu.pk

Abstract

In last decade, application developers attained improved performances by merely employing the machines based on higher-clocked processors. However, in 2003 multi-core processors emerged and eradicated the old processor manufacturing technology based on increasing processor's clock frequencies. After emergence of new parallel processor architectures, serial applications must be re-engineered into parallel versions to exploit the computing power of the existing hardware. In this paper, we present an efficient parallel implementation of minimum spanning tree algorithm to take advantage of the computing power of multi-core machines. Computer network routing, civil infrastructure planning and cluster analysis are typically use-cases of spanning tree problem. The experimental results show that the proposed algorithm is scalable for different machine and graph sizes. The methodology is simple and can easily be implemented using different shared-memory parallel programming models.

Keywords: Graph, Minimum spanning tree, Performance analysis, OpenMP.

1. Introduction

Today, multi-core processors have emerged as a viable source of processing power. The introduction of multi-core architecture was due to the power consumption and heat dissipation problems associated with high-clocked single-core processors [1]. A multi-core processor consists of several processing units known as cores [2]. On single core-processors, most of the applications were developed using sequential execution model. Applications executed on new processor architectures suffer performance degradations due to the stall in processor's clock frequencies. To exploit the processing power of the underlying multi-cores, applications need to be re-engineered and parallelised [3]. Computer network routing, civil infrastructure planning and cluster analysis are typical use-cases of spanning tree problem. A spanning tree can be defined as a subset of Graph G, where all the vertices are covered with minimum possible number of edges. Basic properties of a spanning tree are that it is in the form of a connected graph and does not contain any cycles. A Minimum Spanning Tree (MST) [4] connects vertices of a weighted graph such that the total weight is minimum. In this paper, we discuss the two famous MST algorithms known as Kruskal's [5] and Prim's [6] [7] algorithms. Major contributions of the paper are the development and benchmarking of parallel implementations of Kruskal's and Prim's algorithms. Moreover, we have evaluated the attained performance results using low-level hardware performance counters.

Kruskal's [5] and Prim's [6] [7] algorithms are the two basic techniques to solve the minimum spanning tree problem. An MST represents a sub-graph of an undirected graph such that the sub-graph spans (includes) all graph nodes, is connected, is acyclic, and has minimum total edge-weight. Both the Prim's and Kruskal's algorithms utilize undirected weighted graphs. Prim's and Kruskal's algorithms are considered as greedy algorithms and produce optimal solutions for the MST problem. Prim's algorithm is similar to Dijkstra's algorithm [8]; however, it records previous edge-weights instead of path lengths.

In this work, we present a shared memory-based parallel implementations of Kruskal's [5] and Prim's [6] [7] algorithms. To program shared memory parallel machines, two most-often-used methods are 1) hand-coded parallel threads and 2) using serial code with compiler directives. Writing parallel program is a difficult task, especially if one is required to hand-code the parallel threads. To program shared memory parallel machines, today OpenMP [9] is the most widely used framework for parallelization of the serial code. All threading models typically involve a large overhead related to task parallelization and execution [10] [11]. OpenMP compared to threading method induces less parallelization and execution overhead [12]. Thread-pool model of OpenMP results in significantly less overhead during execution of a parallel program [13]. This paper presents an analysis and study of the parallel MST (Prim's and Kruskal's) programs executed using classical multi-threaded and OpenMP-based execution models.

The rest of the paper is organized as follows: Section II presents the work related to parallelization of Prim's and Kruskal's algorithms. Section III discusses working of the Prim's and Kruskal's MST algorithms. Section IV looks at detailed insight of parallelization of these algorithms. Section V presents the experimental results and the low-level performance analysis of the parallel executions. Section VI concludes the paper.

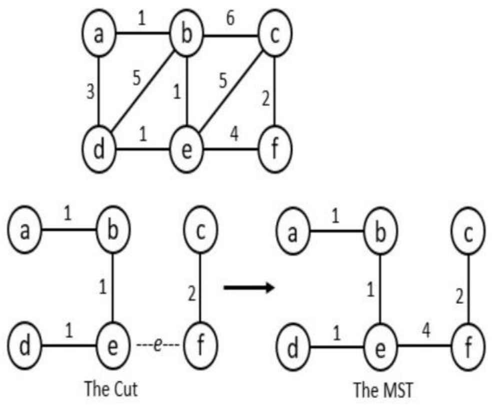


Figure 1: Cut property of MST

2. Related Work

Authors in [14] have proposed a parallel Prim's algorithm implementation on symmetric multiprocessors using cut property of an MST. The cut property can be defined as for any cut C in a graph, if the weight of an edge of cut C is strictly lesser than the weights of all others edges in the cut C, only then this edge belongs to all MSTs in the graph. Figure 1 shows the CUT property of MST. Suppose, S = (a, b, c, d, e, f) Figure 1 shows the CUT property of MST. Different parallel tasks may initiate separate sub-trees simultaneously and then after merging these sub-trees, the final an MST is formed. Suppose, S = (a, b, c, d, e, f) a parallel task starts developing its tree by selecting the smallest weighted edge (a, b) in the portion of the graph assigned to it. Edges (b, e) and (d, e) also have the smallest weights with no circular paths, and are so included in the tree. Another parallel task starts from a different vertex (f) to form its part of the tree and finds edge (c, f) as the smallest of the assigned edges to this task. Now, for both the parallel tasks the next smallest weighted edge is (e, f) (and has no cycle.) At this point, both the tasks merge their sub-trees to build the final MST.

Another parallel implementation of Prim's algorithm is proposed in [14]. The authors experimented using shred memory multi-processor machines. The presented parallel implementation arbitrarily selects a vertex and keeps it as the root-vertex. Each thread starts a distinct parallel tree. The threads have the ability to conduct asynchronous signals. In the meantime, when crash occurs, one of the threads sends signals to other executing threads using a merge tree operation. In the end, the first thread (thread-0) will have the calculated MST. Authors in [14] employed a load-balanced thread scheduling to reduce make-span of the application.

In [15], authors proposed an efficient parallel implementation of Kruskal's algorithm. The proposed implementation employed helper-thread technique. Kruskal's algorithm is known for showing characteristically sequential features, because it strictly examines all the edges whether they are part of the minimum spanning forest [16] graph or not. However, parallel characteristics of Kruskal's algorithm was exploited using helper threads which check each edge having maximum weight for the cycle. An edge is rejected if a cycle is found. The main thread only checks those edges which are not discarded by helper threads. Using the main and helper thread mechanism the proposed technique avoids any blocking or non-blocking synchronization.

In [17], authors proposed a new approach to speed-up the minimum spanning forest algorithm. The accelerated performance was achieved by employing cache optimization and reduced synchronization overhead. The reduced random memory access behaviour resulted in improved speed-up of Bor˚uvka-based implementation.
Authors in [18] proposed a fast solution to the MST problem based on Bor˚uvka algorithm.

The proposed platform-independent implementation can be executed using multicore CPUs or GPUs. The authors [18] introduced a new and effective technique to perform a contraction of the graph. The contracted graph is obtained by merging vertices into super-vertices. To optimize data-locality, the authors employed compressed sparse row format to build the contracted graph. Their implemented version achieved linear scalability up to 8 threads.

Another approach presented in [19] analyses two algorithms: Shiloach-Vishkin and Hirschberg-Chandra-Sarwate. Authors proposed a new parallel-randomized algorithm for calculating an MST. They employed a randomized greedy approach for the implementation. The employed greedy approach allows one processor to arbitrarily access another processor for work stealing if it finishes its assigned task earlier.

Compared to the helper-threading scheme mentioned in [14] and other techniques discussed in this section, our proposed parallelization technique is simpler and easily implementable. We create threads at runtime that reduces the cost of computation, (as compared to 14]).

3. Minimum Spanning Tree Methods

Prim's and Kruskal's algorithms are two well-known and most-used techniques used for the solution of a minimum spanning tree problem. Below sections present the detailed description of these algorithms.

A. Prim's Algorithm**:** Prim's is a memory-bound algorithm and its performance largely depends on the memory accesses pattern and memory organization [9]. After input is provided, Prim's algorithm starts with an arbitrary vertex (let's say vertex B in our example, see Figure 2) in the graph and marks it as visited. In the next step, the algorithm considers all the edges connected to this vertex (for example edges (B, A), (B, C) and (B, F)), finds the minimum weighted-edge (B, A) among them and adds its weight to the MST. The vertex on the other side connected to this minimum weighted-edge (vertex A) is now the visited one. Now all of the edges connected to this vertex are considered and the

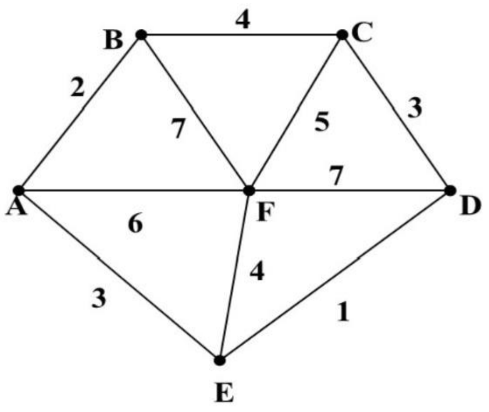


Figure 2: Example: An Undirected Weighted Graph

minimum one is selected and added to the MST if it does not form a circle. In this way, all the vertices connected by the minimum weighted acyclic edges are found, that form the MST.

B. Kruskal's Algorithm: Kruskal's algorithm is based on a greedy approach and is known for exhibiting inherently sequential characteristics. Kruskal's algorithm strictly examines the edges whether they are part of the minimum spanning forest or not [16]. Kruskal's algorithm sorts all edges in ascending order of their weights. It starts adding the minimum weighted-edges to the MST (edges not forming a cycle).

A typical implementation of the Kruskal's algorithm starts with input from the user for the number of vertices and related weights for each edge of the graph. After reading all the edges, the program starts to find a minimum cost edge in the graph (1 less than the total vertices to avoid a loop). The minimum spanning tree is provided as output to the user, showing all the edges that generate a part of the MST. Vertices that do not have edges between them are indicated with 0 values in Table I.

Table 1: An Example Adjacency Matrix (for Figure 2)

	A	B	C	D	E	F
A	0	2	0	0	3	6
B	2	0	4	0	0	7
C	0	4	0	3	0	5
D	0	0	3	0	1	7
E	3	0	3	1	0	4
F	6	7	5	7	4	0

4. Parallelization of Kruskal's and Prim's Algorithms

For parallelization of both the Kruskal's and Prim's algorithms, we employed a divide-and-conquer technique [20]. First, we divide the adjacency matrix into four equal parts considering both the rows and columns of the matrix. Figure 3 shows an example of symmetric matrix having 8 rows and 8 columns. The matrix can be divided into four parts: A, B, C, and D as shown in Figure 3. Figure 3 shows that the matrix partitions B and C are the same and symmetric. Both of these partitions contain exactly the same set of edges. Any one of these two partitions (B or C) is enough to represent the graph. Therefore, we only employ three parts of the matrix (i.e., A, B, and D) for parallelization. As we have only three parts of the adjacency matrix to compute, we use the number of threads as a multiple of three. The strategy of using the part of the matrix reduces the computational cost considerably and results in improved performance. Each of the three selected parts of the matrix can be computed using a single or multiple computing threads. Each computing thread computes and finds a minimum weighted-edge within the assigned matrix part and returns its cost to the main program. After receiving all the minimum costs (for lower cost edges), the main program selects an edge with the lowest cost. After that, the lowest-cost edge is checked for a cycle in the graph and a cycle-free edge is added to the MST. The correctness of both the serial and parallel versions are ensured by comparing the produced results.

A. Kruskal's Parallelization

The proposed parallelization of the Kruskal's algorithms is performed using the following steps:

Step-1: Partition the input N × N adjacency matrix of a graph G into 4 equal fragments as shown in Figure 2. As we assume that input is an undirected simple graph, fragments B and C are symmetric and same. Using this strategy, we have to compute only 3 parts of the matrix, which reduces the overall computational cost of the parallel version;

Step-2: Using p = 3× i threads are created to compute an MST T by employing the 3 partitions i.e., ith threads for each partition. Each thread sorts the matrix representing the edge weights. In case of more employed processors, one matrix part can be computed using multiple threads;

Step-3: Tree generation is initiated by iterating over the sorted edge-list obtained in step 2;

Step-4: An edge (u, v) is inserted into the MST; if it does not create cycles, i.e., (u, v) ∪ T ⇐⇒ T' does not contain cycles;

Step-5: Step 4 is repeated till node Count (G) ≠ node Count (T).

B. Prim's Parallelization

The proposed parallelization of the Prim's algorithms is performed using the following steps:

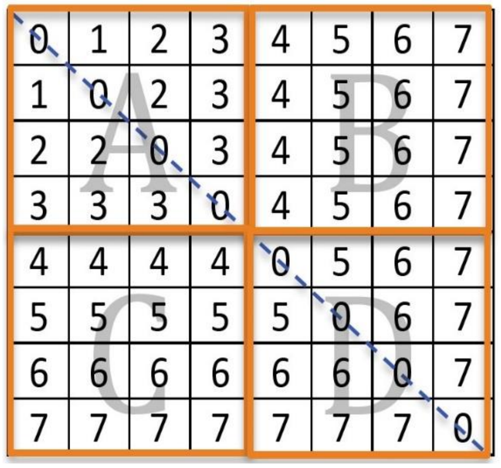


Figure 3: Division of a Symmetric Matrix

Step-1: Partitioning approach for Prim's implementation is similar to the Kruskal's implementation presented in this work. Following are the detailed steps for the parallelization of Prim's algorithm;

Step-2: p = (3 × i) +1 threads are created to compute the MST T from the 3 partitions, i.e., ith threads for each partition. In case of more employed processors, one matrix part can be computed using multiple threads. The additional thread, i.e., main thread, is employed to coordinate among the rest of the threads;

Step-3: Every thread pi where i≤n finds minimum weight edge ei within a row of the adjacency matrix;

Step-4: Every thread pi sends its unmarked minimum ei edge to the main thread. The main thread identifies the global minimum edge from the received edges emin, adds it to the MST and broadcasts it to all other threads;

Step-5: A computing thread marks corresponding vertices connected by emin as belonging to the MST and updates their assigned part of row i.e. Steps 4 and 5 are repeated till node Count(G) ≠ node Count(T).

5. Results and Discussion

A. Experimental Setup: For experimentations, we employ two nodes or multi-core machines, N1 and N2 parts of our computational cluster. The N1 multicore machine is based on quad-core Intel Q6600 2.4Ghz processor with 8MBs of L2 cache memory. The second multicore machine named N2 was based on Intel core i5-4460 quad-core processor. The processor has a clock speed of 3.2 GHz with turbo-boost option up to 3.4 GHz clock-rate. The processor has a shared L3 cache of size 6 MBs. The implementation of the serial version was done in C++ programming language. The parallel versions were implemented using a threading library of C++ and an industry standard for shared memory parallel programming called OpenMP [13], [21]. Six different graph sizes were used for the experimentation i.e., 256, 512, 1024, 2048, 4096, and 8192 node graphs (having density of 0.87). The parallel versions were evaluated up to 12 parallel tasks on the shared machine. To understand the achieved performances, we measure low-level hardware performance counters [22]. Hardware performance counters are employed for originating micro-architecture level execution profiles. Some of the employed low-level performance counters are L1 data loads, L1 cache misses, last-level cache loads, last-level cache misses, time-elapsed, etc.

B. Kruskal's Results: Figure 4 shows the experimental results of the Kruskal's algorithm. The experimental results conducted on N1 multicore node are shown in Figure 4(a). Larger (graph and machine-size) experiments were conducted on N2 multi-core machine are shown in Figures 4(b) and 4(c). As shown in Figure 4(a), we can observe that the execution time of the Kruskal's parallel implementation (based on 3 parallel tasks) has reduced significantly. The parallel version was implemented using C++ multi-threaded API (named as POSIX execution) and consumes 44.28% less execution time for graph of 256 nodes. For the graph with 512 nodes, the multithreaded implementation performs 13.63% better in terms of execution speed compared to the serial version of the kruskal's implementation. For the largest problem size (graph of 1024 nodes), the parallel implementation achieves 69.14% less execution time. The OpenMP based parallel implementation of the Kruskal's algorithm scales better than the POSIX based parallel implementation (as shown in Figure 4(a)). The parallel version implemented using OpenMP API consumes 46.78% less time for graph size 256. For the graph (with 512 nodes), the OpenMP based implementation performs 5.5× better compared to the serial implementation.

For the largest problem size (graph of 1024 nodes), the OpenMP based parallel implementation achieves 23.27 times less execution time. Figure 4(b) shows the Kruskal's parallel implementation using a C++ based multi-threaded API. The experiment was conducted using 3, 6, 9, and 12 parallel threads. For this experiment, we employ different graph sizes i.e., 256, 512, 1024, 2048, 4096, and 8192 node graphs. The multi-threaded executions based on 3−12 threads show exceptional scalability of the proposed parallel technique of the Kruskal's algorithm.

Figure 4(c) shows execution results of multi-threaded andOpenMP based parallel executions. For this experiment, 256, 512, 1024, 2048, 4096, and 8192 node graphs were employed and executed using 6 parallel tasks (using both the C++ multi-threaded and OpenMP based frameworks). As shown in Figure 4(c), our proposed parallel implementation of Kruskal's algorithm achieves commendable scalability for all graph sizes (256–8192). The implementation is more scalable for large graph sizes i.e., 4096 and 8192 node graphs (as shown in Figure 4(c)). This experiment shows that the OpenMP based implementation achieves more improved performance compared to the multi-threaded execution. The better performance by the openMP based implementation is due to the decreased threading overhead (because of a thread-pool mechanism of OpenMP).

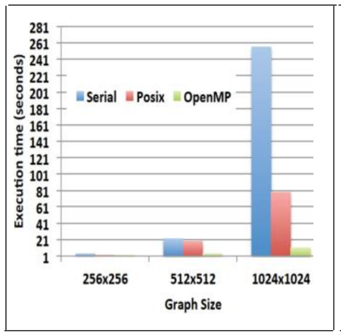
C. Prim's Results: Figure 5 shows the experimental results of the Prims' algorithm. The experimental results conducted on N1 multicore node are shown in Figure 5(a). Larger graph and machine size based experiments were conducted on N2 machine and are shown in Figures 5(b) and 5(c).

Figure 5(a) shows that the execution time of the Prim's parallel implementation is significantly low compared to the serial implementation of the algorithm. The multi-threaded parallel implementation of the algorithm consumes 40% less time for graph of 256 nodes. For the graph of 512 nodes, the multi-threaded implementation performs 23.8% better compared to the serial

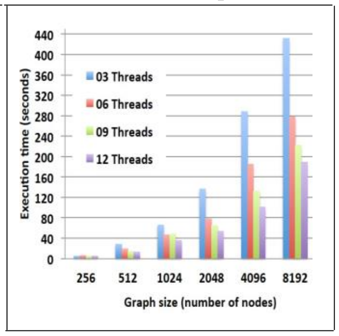
Fig. 4: Kruskal's algorithm - experimental results.

Fig. 5: Prim's algorithm - experimental results.

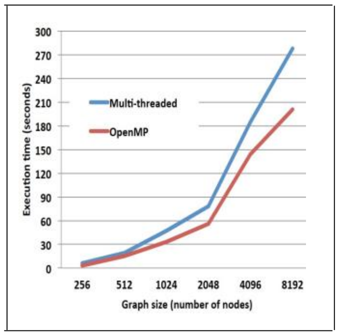
For 1024 node graph, the parallel multi-threaded implementation achieves 15.15% less execution time compared to the serial implementation. Figure 5(a) also shows the performance results of the OpenMP based implementation of the algorithm. The OpenMP based execution consumes 40% less



(a) N machine experimental results.

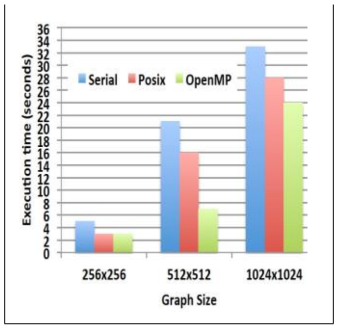


(b) N Machine Experimental results.

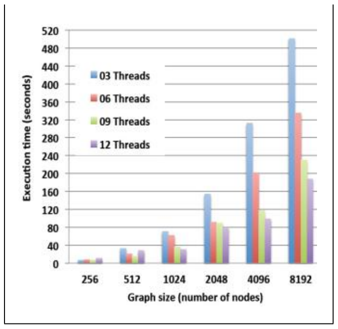


(c) N machine - multi-threaded Vs OpenMP.

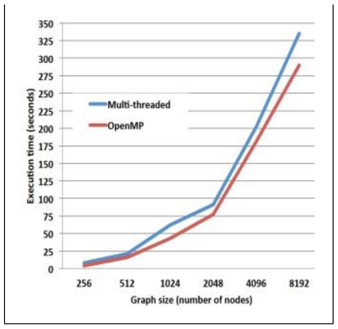
Figure 4: Kruskal's Algorithm - Experimental Results



(a) N machine experimental



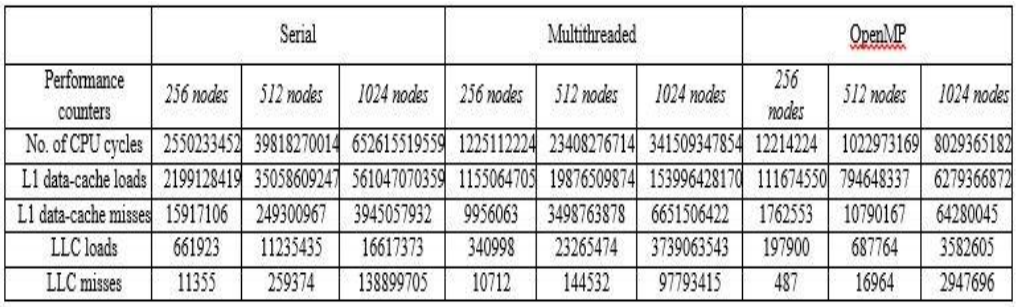
(b) N machine experimental



(c) N machine - multi-threaded Vs

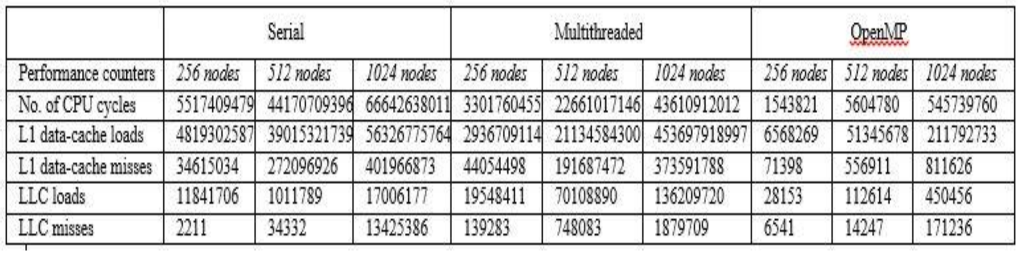
Figure 5: Prim's Algorithm - Experimental Results

Table 2: Kruskal's implementations' low-level Performance Analysis



	Serial			Multithreaded			OpenMP		
Performance counters	*256 nodes*	*512 nodes*	*1024 nodes*	*256 nodes*	*512 nodes*	*1024 nodes*	*256 nodes*	*512 nodes*	*1024 nodes*
No. of CPU cycles	2550233452	39818270014	652615519559	1225112224	23408276714	341509347854	12214224	1022973169	8029365182
L1 data-cache loads	2199128419	35058609247	561047070359	1155064705	19876509874	153996428170	111674550	794648337	6279366872
L1 data-cache misses	15917106	249300967	3945057932	9956063	3498763878	6651506422	1762553	10790167	64280045
LLC loads	661923	11235435	16617373	340998	23265474	3739063543	197900	687764	3582605
LLC misses	11355	259374	138899705	10712	144532	97793415	487	16964	2947696

Table 3: Prim's implementations' low-level performance analysis



	Serial			Multithreaded			OpenMP		
Performance counters	*256 nodes*	*512 nodes*	*1024 nodes*	*256 nodes*	*512 nodes*	*1024 nodes*	*256 nodes*	*512 nodes*	*1024 nodes*
No. of CPU cycles	5517409479	44170709396	66642638011	3301760455	22661017146	43610912012	1543821	5604780	545739760
L1 data-cache loads	4819302587	39015321739	56326775764	2936709114	21134584300	453697918997	6568269	51345678	211792733
L1 data-cache misses	34615034	272096926	401966873	44054498	191687472	373591788	71398	556911	811626
LLC loads	11841706	1011789	17006177	19548411	70108890	136209720	28153	112614	450456
LLC misses	2211	34332	13425386	139283	748083	1879709	6541	14247	171236

time for graph of 256 nodes. For 512 node graph, the OpenMP based implementation performs 3× better compared to the serial version. For the larger graph (1024 nodes), the parallel implementation achieves 1.37 times less execution time.

Figure 5(b) shows the Prims' parallel implementation using a C++ based multi-threaded API. The experiment was conducted using 3, 6, 9, and 12 parallel threads and different graph sizes i.e., 256, 512, 1024, 2048, 4096, and 8192 node graphs. The multi-threaded executions based on 3−12 threads show excellent scalability of the proposed parallel technique of the Prims' algorithm.

Figure 5(c) shows execution results of multi-threaded and OpenMP based parallel executions. For this experiment, 256, 512, 1024, 2048, 4096, and 8192 node graphs were Employed and executed using 6 parallel tasks (using both the C++ multi-threaded and OpenMP based frameworks). As shown in Figure 5(c), our proposed parallel implementation of Prims' algorithm achieves excellent scalability for all graph sizes (256−8192). The implementation is more scalable for large graph sizes i.e., 4096 and 8192 node graphs (as shown in Figure 5(c)). This experiment shows that the OpenMP based implementation achieves more improved performance compared to the multi-threaded execution. Due to the OpenMP's thread-pool mechanism, less threading-overhead results in improved performances for the OpenMP based executions.

Tables II and III show measurements of the low-level hardware performance counters. We can observe several consistent trends in both tables. In the majority cases, the OpenMP based executions observe lower number of cache loads and cache misses (approximately 10.3% less compared to the multi-threaded execution). The low number of cache misses result in on average better performance of the OpenMP based implementations.

6. Conclusions and Future Work

In this paper, we proposed and evaluated performance of parallel MST methods (Kruskal's and Prim's algorithms). The proposed parallel algorithms were evaluated for their scalability by employing different graph and machine sizes. The experiments showed that the Kruskal's achieves 23× better results in terms of execution time compared to the serial version of the algorithm. For the prim's algorithm, we attained up to 3× better performances compared to the serial version of the algorithm. Moreover, the OpenMP based implementations of both algorithms showed excellent performance and scalability compared to the simple multi-threaded based implementations. Our future work includes large-scale experiments on compute Clouds such as Amazon EC2 and Microsoft Azure platforms.

References

[1] D. Geer, "Chip makers turn to multicore processors," Computer, vol. 38, no. 5, pp. 11–13, 2005.

[2] D. E. Lenoski and W.-D. Weber, Scalable shared-memory multiprocessing. Elsevier, 2014.

[3] M. Aleem, R. Prodan, and T. Fahringer, "The javasymphony extensions for parallel gpu computing," in 2012 41st International Conference on Parallel Processing. IEEE, 2012, pp. 30–39.

[4] P. Jayawant and K. Glavin, "Minimum spanning trees," Involve, a Journal of Mathematics, vol. 2, no. 4, pp. 439–450, 2009.

[5] C. Zhong, M. Malinen, D. Miao, and P. Franti, "A fast minimum¨ spanning tree algorithm based on k-means," Information Sciences, vol. 295, pp. 1–17, 2015.

[6] S. Manen, M. Guillaumin, and L. Van Gool, "Prime object proposals with randomized prim's algorithm," in Proceedings of the IEEE International Conference on Computer Vision, 2013, pp. 2536–2543.

[7] W. Guttmann, "Relation-algebraic verification of prim's minimum spanning tree algorithm," in International Colloquium on Theoretical Aspects of Computing. Springer, 2016, pp. 51–68.

[8] M. Yan, "Dijkstra's algorithm," Massachusetts Institute of Technology. regexstr, 2014.

[9] J. Adams and E. Shoop, "Teaching shared memory parallel concepts with openmp," Journal of Computing Sciences in Colleges, vol. 30, no. 1, pp. 70–71, 2014.

[10] A. M. Castaldo and R. C. Whaley, "Minimizing startup costs for performance-critical threading," in Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009, pp. 1–8.

[11] X. Ding and J. Shan, "Diagnosing virtualization overhead for multithreaded computation on multicore platforms," in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2015, pp. 226–233.

[12] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain graphs: Openmp performance analysis made easy," in Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2016, p. 28.

[13] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? on the scalability of openmp implementations," in European Conference on Parallel Processing. Springer, 2015, pp. 451–463.

[14] R. Setia, A. Nedunchezhian, and S. Balachandran, "A new parallel algorithm for minimum spanning tree problem," in Proc. International Conference on High Performance Computing (HiPC), 2009, pp. 1–5.

[15] A. Katsigiannis, N. Anastopoulos, K. Nikas, and N. Koziris, "An approach to parallelize kruskal's algorithm using helper threads," in Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE, 2012, pp. 1601–1610.

[16] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," in ACM SIGPLAN Notices, vol. 47, no. 8. ACM, 2012, pp. 205–214.

[17] G. Cong, I. Tanase, and Y. Xia, "Accelerating minimum spanning forest computations on multicore platforms," in European Conference on Parallel Processing. Springer, 2015, pp. 541–552.

[18] C. da Silva Sousa, A. Mariano, and A. Proenc¸a, "A generic and highly efficient parallel variant of boruvka's algorithm," in 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2015, pp. 610–617.

[19] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors," in Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. IEEE, 2004, p. 38.

[20] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati, "A divide-and-conquer parallel pattern implementation for multicores," in Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems. ACM, 2016, pp. 10–19.

[21] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison," in Proceedings of the Many-core Applications Research Community (MARC) Symp. at RWTH Aachen University. RWTH Achen University, 2012, pp. 38–44.

[22] X. Xie, H. Jiang, H. Jin, W. Cao, P. Yuan, and L. T. Yang, "Metis: a profiling toolkit based on the virtualization of hardware performance counters," Human-centric Computing and Information Sciences, vol. 2, no. 1, pp. 1–15, 2012.