

Parallelizing Shortest Path Algorithms in Graph Theory and Comparing it with the Serial Algorithms

Parallel And Distributed Computing
Prof.SairaBanu .J

Review-1 Document

Presented By:

Shikhar Singh-16BCE2316

Aratrik Paul – 16BCE0519

Abstract:

A parallel algorithm can be executed simultaneously on many different processing devices and then combined together to get the correct result. Parallel algorithms are highly useful in processing huge volumes of data in quick time .The shortest path algorithms to find the shortest route in graph theory are predominantly sequential in nature. The serial implementation of the Dijkstra's algorithm has been well documented and is tried and tested . However, for a large number of nodes, the serial algorithm generally fails to maintain its efficiency as the time complexity is quite high. Through this literature Review we aim to study the implementation of the parallel algorithm to find the shortest path in the directed graph. Along with this, we aim to study and compare the parallel and serial computing of a host of shortest path algorithms using the OpenMP and SIMD(platform)

Introduction:

Shortest path algorithms are particularly important on the Internet, as they determine the most efficient paths along which packets should reach their destinations. OSPF and IS-IS routing protocols employ Dijkstra algorithm, while RIP utilizes Bellman-Ford algorithm. These algorithms find shortest paths from a single source node to all other nodes, and, they are called single source shortest path (SSSP) algorithms for this reason. Complexities of these two algorithms are $O(mn)$ and $O(m + n \log n)$, respectively, where n is the number of vertices, and m the number of edges. Networks have hierarchical structure, and routers do not get the full information about network topology. Shortest paths are calculated in domains of several hundred routers. However, nowadays large datacenter networks with flat topologies are developing. In such networks, SSSP algorithms have to become faster.

The technique used to parallelize the Dijkstra's algorithm we divide the nodes among the processors or the cores and let them handle each cluster. This greatly reduces the runtime of the algorithm (especially when number of nodes is large).

Literature Review: Grouping based on Concepts : (Group 1: Dijkstra's Algorithm)

(Group2: OpenMP and algorithm parallelization)

(Group 3: Bellman Ford and other SSSP algorithms)

List of summaries for each research paper along with their Group:

1. Performance of Shortest Path Algorithm(Group 3)

Shortest Path Algorithms are implemented serially uptill now. A few examples are Dijkstra's algorithm and the Bellman Ford Algorithm. Since the scale of these algorithms are increasing, there is a need to parallelize them. Using multicore processor the Single Source Shortest Path Algorithm has been implemented in this paper.

2. Implementing Irregular Parallel Algorithms with OpenMP(Group 2)

A simple breadth search first algorithm is used to show the stepping stone and the deficiency of the OpenMP platform. The proposal given in this paper is to give an extension of OpenMP to help cancel the threads in a parallel region.

3. Performance Analysis of Algorithm Using OpenMP(Group 2)

In this paper a description is given on how a given sequential algorithm can be parallelized making use of the multicore chip. It also deals with how a given program can be compiled using OpenMP libraries. The Dijkstra's algorithm has been parallelized as an example. The algorithm along with the steps and impact are described clearly.

4. An Improved Dijkstra's algorithm application to multi-core processors(Group 1)

Using a multi-core Environment ,the efficiency of the Dijkstra's algorithm can be increased by as much as 50% when compared to the traditional algorithm. While this has become a mainstream in embedded systems, improved algorithm is of great commercial value. It requires the use of OpenMP tools.

5. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0(Group 3)

This paper describes how OpenMP 3.0 is used to implement all the shortest path algorithms. It is shown that for a large number of vertices, the algorithm running on Open MP 3.0 surpasses the one on OpenMP 2.5 by 1.6 times. The algorithms are refined according to the parallel techniques of programming.

6. PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEM ON TIME DEPENDENT GRAPHS (Group 3)

This paper develops three different implementations by using CUDA and OpenMP. This is a Cuda based version, A OpenMP based version and a hybrid Cuda and OpenMP based version. We get up to 10-fold speedup in the OpenMP version, and a 17-fold speed up in the two versions.

7. Task Level Parallelization of Irregular Computations using OpenMP 3.0 (Group 2)

OpenMP is a standard parallel programming language used to develop parallel applications on shared memory machines. OpenMP is very suitable for designing parallel algorithms for regular applications where the amount of work is known apriori and therefore distribution of work among the threads can be done at compile time. It has been shown that OpenMP produces unsatisfactory performance for irregular applications.

8. New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA) (Group 3)

In this paper, the Single Shortest Path Algorithm i.e the Bellman Ford Algorithm. SSSP problem is a major problem in the graph theory that has various applications in real world that demand execution of these algorithms in large graphs having millions of edges and sequential implementation of these algorithms takes large amount of time. In this paper, we investigate some methods which aim at parallelizing Bellman Ford Algorithm and to implement some extended or enhanced versions of this algorithm over GPU.

9. Performance Analysis of Parallel Speedup Techniques for Shortest Path Queries in Networks of Random and Planar Types (Group 3)

This paper deals with comparison of parallelized speedup techniques with sequential version of the same and finding performance improvement achieved in parallelized speedup techniques with respect to runtime and number of vertices visited during shortest path computation.

10. Parallel Algorithm Performance Analysis using OpenMP for Multicore Machines (Group 2)

OpenMP programming model helps in creating multithreaded applications for the existing sequential programs. This paper presents the performance potential of the parallel programming model over sequential programming model using OpenMP. The experimental results show that a significant performance is achieved on multi-core system using parallel algorithm.

11. A Parallelization of Dijkstra's Algorithms (Group 2)

This paper divides Dijkstra's sequential SSSP algorithm into a number of phases, such that the operations within a phase can be done in parallel. A PRAM algorithm is given based on these criteria and analyze its performance on random digraphs with random edge weights uniformly distributed in $[0,1]$.

12. Serial and parallel implementation of Shortest path algorithm in the optimization Of public transport travel (Group 3)

This paper deals with the optimization of Public Transport Travel using the Shortest Path Algorithms Optimization using Parallel Techniques. It uses a large set of nodal points (approximately 500) where each nodal point basically represents a source or destination or it can even be a transit point between any source and destination with respect to public transport of Bangalore Metropolitan Transport Corporation.

13. Parallel Computing using OpenMP (Group 2)

In this paper, a study on a hybrid approach to programming such systems – a combination of two traditional programming models, MPI and OpenMP. The performance of standard benchmarks from the multi-zone NAS Parallel Benchmarks and two full applications using this approach on several multi-core based systems including an SGI Altix 4700, an IBM p575+ and an SGI Altix ICE 8200EX are presented. New data locality extensions to OpenMP to better match the hierarchical memory structure of multi-core architectures are also presented.

14. Parallel Dijkstra Algorithm using OpenMP (Group 1)

The Sixth SIAM Workshop on Combinatorial Scientific Computing, CSC14, was organized at the Ecole Normale Supérieure de Lyon, France on 21st to 23rd July, 2014. This two and a half day event marked the sixth in a series that started ten years ago in San Francisco, USA. The CSC14 Workshop's focus was on combinatorial mathematics and algorithms in high performance computing, broadly interpreted. The workshop featured three invited talks, 27 contributed talks and eight poster presentations. All three invited talks were focused on two interesting fields of research specifically: randomized algorithms for numerical linear algebra and network analysis. The contributed talks and the posters targeted modeling, analysis, bisection, clustering, and partitioning of graphs, applied in the context of networks, sparse matrix factorizations, iterative solvers, fast multi-pole methods, automatic differentiation, high-performance computing, and linear programming.

15. Parallel Dijkstra's Algorithm using OpenMP (Group 1)

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%.

16. Concurrent selection of the shortest paths and distances in directed graphs using vertical processing systems (Group 3)

In this paper a new implementation of Dijkstra's shortest path algorithm is proposed on a model of associative parallel processors with the vertical data processing (the STAR-machine) to obtain for every vertex of a directed graph the distance along with the shortest path from the source vertex. The correctness of this method is proven along with the computation of the time complexity. The SIMD type processors are used to implement this.

17. PHAST: Hardware-accelerated shortest path trees (Group 3)

A novel algorithm to solve the non-negative single-source shortest path problem on road networks and graphs with low highway dimension is presented in this paper. After a quick preprocessing phase, all distances from a given source in the graph with essentially a linear sweep over all vertices can be computed. Because this sweep is independent of the source, the vertices are able to be reordered in advance to exploit locality. Moreover, this algorithm takes advantage of features of modern CPU architectures, such as SSE and multiple cores.

18. UNICOR: a species connectivity and corridor network simulator (Group 1)

An introduction to UNiversal CORridor network simulator (UNICOR), a species connectivity and corridor identification tool is given. UNICOR applies Dijkstra's shortest path algorithm to individual-based simulations. Outputs can be used to designate movement corridors, identify isolated populations, and prioritize conservation plans to promote species persistence. The key features include a driver-module framework, connectivity mapping with thresholding and buffering, and calculation of graph theory metrics.

19. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths (Group 3)

Three parallel friendly and work-efficient methods to solve this Single-Source Shortest Paths (SSSP) problem: Workfront Sweep, Near-Far and Bucketing are explained. These methods choose different approaches to balance the tradeoff between saving work and organizational overhead. All of these methods do much less work than traditional Bellman-Ford methods, while adding only a modest amount of extra work over serial methods. These methods are designed to have a sufficient parallel workload to fill modern massively-parallel machines, and select reorganizational schemes that map well to these architectures.

20. Hierarchical Hub Labelings for Shortest Paths (Group 1)

A study on the hierarchical hub labelings for computing shortest paths is discussed. The new theoretical insights into the structure of hierarchical labels lead to faster pre-processing algorithms, making the labeling approach practical for a wider class of graphs. Also, finding smaller labels for road networks, improving the query speed is another goal for this paper. This is done by improvising on the linear-time Dijkstra's Algorithm.

21.Parallelization of the ant colony optimization for the shortest path problem using OpenMP and CUDA (Group 3)

Finding efficient vehicle routes is an important logistics problem which has been studied for several decades. Metaheuristic algorithms offer some solutions for that problem. This paper deals with GPU implementation of the ant colony optimization algorithm (ACO), which can be used to find the best vehicle route between designated points. The algorithm is applied on finding the shortest path in several oriented graphs. It is embarrassingly parallel, since each ant constructs a possible problem solution independently. Results of sequential and parallelized implementation of the algorithm are presented.

22.Performance comparison of OpenMP, MPI, and mapreduce in practical problems (Group 2)

This paper briefly reviews the parallel computing models and describes three widely recognized parallel programming frameworks: OpenMP, MPI, and MapReduce. OpenMP is the de facto standard for parallel programming on shared memory systems. MPI is the de facto industry standard for distributed memory systems. MapReduce framework has become the de facto standard for large scale data-intensive applications. Qualitative pros and cons of each framework are known, but quantitative performance indexes help get a good picture of which framework to use for the applications. As benchmark problems to compare those frameworks, two problems are chosen: all-pairs-shortest-path problem and data join problem. This paper presents the parallel programs for the problems implemented on the three frameworks, respectively.

23.Survey of Speedup Techniques for Shortest Path Algorithms (Group 3)

This paper reviews some latest representative results classified according to the underlying speedup techniques, including basic speedup techniques like priority queues, goal-directed techniques, and hierarchical approaches. Moreover, the paper introduces the most recent achievement of the authors on network hierarchy construction and hierarchical algorithm design.

24.Parallel iterative search methods for vehicle routing problems (Group 3)

This paper presents two partition methods that speed up iterative search methods applied to vehicle routing problems including a large number of vehicles. Indeed, using a simple implementation of taboo search as an iterative search method, every best-known solution to classical problems was found. The first partition method (based on a partition into polar regions) is appropriate for Euclidean problems whose cities are regularly distributed around a central depot. The second partition method is suitable for any problem and is based on the arborescence built from the shortest paths from any city to the depot.

25.All-pairs shortest-paths for large graphs on the GPU (Group 3)

The all-pairs shortest-path problem is an intricate part in numerous practical applications. Described is a shared memory cache efficient GPU implementation to solve transitive closure and the all-pairs

shortest-path problem on directed graphs for large datasets. The proposed algorithmic design utilizes the resources available on the NVIDIA G80 GPU architecture using the CUDA API.

Tabular Column:

Title	Author	Journal Name and Date	Key Concepts Concentrated	Advantages and Disadvantages	Future Enhancement
Performance of Shortest Path Algorithm Based on Parallel Vertex Traversal	Mihailo Vesović , Aleksandra Smiljanić , Dušan Kostić	SERBIAN JOURNAL OF ELECTRICAL ENGINEERING , Vol. 13, No. 1, February 2016, 31-43	Parallelization, Shortest Path Algorithms, Single Source Shortest Path, Bellman-Ford, Dijkstra	We conjecture that the Dijkstra algorithm is more efficient for the topologies with larger diameter.	It can be enhanced to be parallelised more under Multithreading.
Implementing Irregular Parallel Algorithms with OpenMP	Michael Süß and Claudia Leopold.	Wilhelmshöher Allee 73, D-34121 Kassel, Germany	forceful cancellation,Deferred cancellation,POSIX Threads,cooperative cancellation	Backwards Source Compatibility,Nested Parallelism,No Resource Leaks	In the future, we plan to explore more applications with OpenMP, trying to find ways to improve the specification in the process
Performance Analysis of Algorithm Using OpenMP	Sumit Pathare , Pranav Kulkarni , Rahul Kardel	IJETT ISSN: 2350 – 0808 September 2014 Volume 1 Issue 1 152	Parallel Thread. Multicore, Multiprocessor, OpenMP	It is only capable of detecting unsynchronized , shared accesses by multiple threads.Parallelize small parts of application.	As data set increases performance of sequential execution falls down where parallel execution is used for large data set then it gives best results than sequential execution
An Improved Dijkstra's algorithm application	Qiong Wu , Guihe Qin, Hongliang Li	Metallurgical and Mining Industry No. 9 — 2015	DIJKSTRA'S ALGORITHM; PARALLELIZATION; MULTI-	improved of parallelize algorithm in speed has	In this paper, the improved of parallelize Dijkstra's

to multi-core processors			CORE; THE SHORTEST PATH	the obvious improvement, speed increased about by 50%	algorithm on multi-cores can greatly increase calculation efficiency
Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0	Eid Albalaw i Parimala Thulasiraman Rупpa Thulasiram	2nd International Conference on Advances in Computer Science and Engineering (CSE 2013)	OpenMP 3.0; All Pair Shortest Path; Task Parallelization	In this paper we implemented one graph problem, all pair shortest path problem in OpenMP 3.0. We showed that the algorithm run 1.6 times faster than the OpenMP 2.5 version	It can be enhanced to be parallelised more using subsequent Multithreading.
Parallel algorithms for shortest path problem On time dependent graphs	Mehmet Akif Ersoy	Graduate Program in Computer Engineering Boğaziçi University 2015	Parallel Thread. Multicore, Multiprocessor, OpenMP	For multi-core system OpenMP provides a lot of performance increase and parallelization.	The parallel running speed can be improved with the increase of the number of cores.
Task Level Parallelization of Irregular Computations using OpenMP 3.0	Eid Albalawi	Department of Computer Science The University of Manitoba Winnipeg, Manitoba, Canada December 2013	multicore, openMP, parallel programming, speedup.	We can improve system performance by lots of parallelization. it is very good parallel platform to achieve high performance.	The parallel running speed can be improved with the increase of the number of cores.
New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design	Pankhari Agarwal, Maitreyee Dutta	International Journal of Computer Applications (0975 – 8887)	SSSP (Single Source Shortest Path) Problem, Graphics Processing Units (GPUs), CUDA	As number of edges is greater than number of nodes and threads are	Achieved results show a considerable speed-up over corresponding serial implementation

Architecture (CUDA)		Volume 110 – No. 13, January 2015	(Compute Unified Device Architecture).	mapped to edges rather than nodes, greater degree of parallelism can be achieved.	n for various graph instances having varying degrees
Performance Analysis of Parallel Speedup Techniques for Shortest Path Queries in Networks of Random and Planar Types	R.Kalpana P.Thambidurai	International Journal of Computer Applications (0975 – 8887) Volume 47– No.24, June 2012	Dijkstra’s algorithm, Shortest path computation, Speedup Techniques, Parallel Speedup.	In general planar graphs give a gain in percentage for arc flags in runtime and in the rest it shows poor results for performance gain	For these types of networks it is suggested to have the sequential version of the speedup technique than parallelized version.
Parallel Algorithm Performance Analysis using OpenMP for Multicore Machines	Mustafa B Waseem Ahmed	, International Journal of Advanced Computer Technology (IJACT) ISSN:2319-7900	multicore, openMP, parallel programming, speedup.	Performance is increased by parallelizing serial algorithm using OpenMP	For multi-core system OpenMP provides a lot of performance increase and parallelization.
A Parallelization of Dijkstra’s Algorithms	A Crauser, Melhorn, Meyer	KIm Stadwadt,6612 3 Saarbucken , Germny	Parallel Thread. Multicore, Multiprocessor, OpenMP	We can improve system performance by lots of parallelization. it is very good parallel platform to achieve high performance.	For multi-core system OpenMP provides a lot of performance increase and parallelization.
Serial and parallel implementation of Shortest path algorithm in the optimization	Guruprasad Nagraj & Dr. Y S Kumaraswamy	International Journal of Computer Science Engineering and Information	Shortest path, Multi core, Public transport, Congestion problem, Transit point.	The parallel running speed can be improved with the increase of the number of cores.	The tabulations shows that the time cost of multithreaded parallel algorithms on

Of public transport travel		Technology Research Vol.1, Issue 1 (2012) 72-87 © TJPRC Pvt. Ltd.			dual-core system are much faster than the serial algorithms.
Parallel Computing using OpenMP	Ms. Ashwini M. Bhugul	IJCSMC, Vol. 6, Issue. 2, February 2017, pg.90 – 94	OpenMP, Parallel Computing	Shared memory parallelism is easier to learn. It uses both fine grain and coarse grained parallelism. It is widely available and portable. serial code statements usually don't need modification	We can improve system performance by lots of parallelization. it is very good parallel platform to achieve high performance.
Parallel Dijkstra Algorithm using OpenMP	Rajesh Parekh Soham Kumar Chowdhary	Volume 47– No.24, June 2012	Dijkstra"s algorithm, Shortest path computation, Speedup	In general planar graphs give a gain in percentage for arc flags in runtime and in the rest it shows poor results for performance gain	Achieved results show a considerable speed-up over corresponding serial implementation for various graph instances having varying degrees
Parallel Quicksort Algorithm using OpenMP	Sinan Sameer Mahmood Al-Dabbagh Nawaf Hazim Barnouti	IJCSMC, Vol. 5, Issue. 6, June 2016, pg.372 – 382	Quicksort, Sorting Algorithms, Parallelism, OpenMP, Parallel Sorting Algorithm, Parallel Computing, Multi-Core, Speedup,	The ratio of the efficiency using 4 threads with 4 cores in the parallel method is close by the optimal result of utilizing such number of cores	We planned in the future to implement the Quicksort algorithm using Message Passing Interface (MPI) and compare its results with

			Parallel Programing		OpenMP method.
Concurrent selection of the shortest paths and distances in directed graphs using vertical processing systems	A.S. Nepomniaschaya	Bul. I. Nov. Comp. Center, Comp. Science, 19 (2013), 61-71 © 2003 NCC Publisher	OpenMP, Parallel Computing	In this paper, we have proposed a new efficient implementation of Dijkstra's shortest path algorithm on the STAR-machine which allows one for every vertex of a directed graph to simultaneously obtain the distance and the shortest path from the source vertex.	correctness of the procedure DistPath and evaluate its time complexity. In the same manner, one can modify the implementation of the BellmanFord shortest path algorithm on the STAR-machine
PHAST: Hardware-accelerated shortest path trees	Daniel Delling, Andrew Goldberg, Renato F. Werneck	Journal of Parallel and Distributed Computing Volume 73, Issue 7, July 2013, Pages 940-952	Novel single-source shortest path algorithm. Takes advantage of features of modern CPU architectures (SSE and multiple cores). Additional speedup when implemented on a GPU.	Moreover, our algorithm takes advantage of features of modern CPU architectures, such as SSE and multiple cores.	We gain additional speedup when implementing our algorithm on a GPU, where it is up to three orders of magnitude faster than Dijkstra's algorithm on a high-end CPU
Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths	Andrew Davidson, Sean Michael Garland, John D. Owens	2014-04-01 Powered by the California Digital Library University of California	GPU computing, graph traversal, single-source shortest paths, sparse graphs	Our work-saving methods always outperform a traditional GPU Bellman-Ford implementation, achieving	We also see significant speedups (20-60x) when compared against a serial implementation on graphs

				rates up to 14x higher on low-degree graphs and 340x higher on scalefree graphs.	with adequately high degree.
Hierarchical Hub Labelings for Shortest Paths	Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck	<u>European Symposium on Algorithms</u> ESA 2012: Algorithms – ESA 2012_pp 24-35	Short Path Road Network Total Order Priority Queue Query time	Our new theoretical insights into the structure of hierarchical labels lead to faster preprocessing algorithms, making the labeling approach practical for a wider class of graphs	We also find smaller labels for road networks, improving the query speed.
Performance comparison of OpenMP, MPI, and mapreduce in practical problems	Sol Ji Kang, Sang Yeon Lee, Keon Myung Lee	Journal Advances in Multimedia - Special advanced issue on Topic Detection, Tracking, and Trend Analysis for Social Multimedia archive__Article No. 7	Shortest path, Multi core, Public transport, Congestion problem, Transit point.	This paper presents the parallel programs for the problems implemented on the three frameworks, respectively. It shows the experiment results on a cluster of computers.	Qualitative pros and cons of each framework are known, but quantitative performance indexes help get a good picture of which framework to use for the applications.
Survey of Speedup Techniques for Shortest Path Algorithms	SONG Qing and WANG Xiao-fan	This paper reviews some latest representative results classified according to the underlying speedup	Dijkstra's algorithm, Shortest path computation, Speedup	Moreover, our algorithm takes advantage of features of modern CPU architectures, such as SSE and multiple cores.	Achieved results show a considerable speed-up over corresponding serial implementation for various graph

		techniques,including basic speedup techniques like priority queues,goal-directed techniques,and hierarchical approaches.			instances having varying degrees
--	--	--	--	--	----------------------------------

Conclusion:

Sequential shortest path algorithms such as Bellman-Ford and Dijkstra dominate on the Internet. As the Internet is growing, faster and more scalable algorithms are desired. Parallel algorithm proposed in this paper outperforms Bellman-Ford for most of the tested topologies. Performance of parallel SSSP algorithm is in all cases comparable to the performance of Dijkstra and it outperforms Dijkstra in large networks. We have shown that our algorithm works well for the common datacenter topologies, especially for the topologies in which the diameter is small and the average vertex degree is large. Another advantage of the proposed algorithm is its potential, as the higher number of cores is likely to give increasing speedups.

References:

1. OpenMP Architecture Review Board: OpenMP specifications.
<http://www.openmp.org/specs> (2005)
2. Mattson, T.G.: How good is OpenMP. *Scientific Programming* 11 (2003) 81–93
3. Hisley, D., Agrawal, G., Satyanarayana, P., Pollock, L.: Porting and performance evaluation of irregular codes using OpenMP. *Concurrency: Practice and Experience* (2000)
4. Dedu, E., Vialle, S., Timsit, C.: Comparison of OpenMP and classical multi-threading parallelization for regular and irregular algorithms. In Fouchal, H., Lee, R.Y., eds.: *Software Engineering Applied to Networking Parallel/Distributed Computing (SNPD)*, Association for Computer and Information Science (2000) 53–60
5. Nikolopoulos, D.S., Polychronopoulos, C.D., Ayguade, E.: Scaling irregular parallel codes with minimal programming effort. In: *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, ACM Press (2001)
6. Dimakopoulos, V.V., Georgopoulos, A., Leontiadis, E., Tzoumas, G.: OMPI compiler homepage. <http://www.cs.uoi.gr/~ompi/> (2003)
7. Süß, M., Leopold, C.: Common mistakes in OpenMP and how to avoid them. In: *Proceedings of the International Workshop on OpenMP - IWOMP'06*. (2006)
8. Sun Microsystems: Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated. <http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html> (1999)
9. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. *SIGARCH Comput. Archit. News* 29(5) (2001) 41–48
10. Süß, M.: University of Kassel OpenMP – UKOMP homepage.
<http://www.plm.eecs.uni-kassel.de/plm/index.php?id=ukomp> (2005)
11. Ashok Jagannathan. *Applications of Shortest Path Algorithms to VLSI Chip Layout Problems*. Thesis Report. University of Illinois. Chicago. 2000.

12. Karla Vittori, Alexandre C.B. Delbem and Sérgio L. Pereira. Ant-Based Phylogenetic Reconstruction (ABPR): A new distance algorithm for phylogenetic estimation based on ant colony optimization. *Genetics and Molecular Biology*, 31(4), 2008.
13. Jiyi Zhang, Wen Luo, Linwang Yuan, Weichang Mei. Shortest path algorithm in GIS network analysis based on Clifford algebra. *Transactions of the IRE Professional Group*. 18(11, 12).
14. A. Bustamam, G. Ardaneswari, D. Lestari, "Implementation of CUDA GPU-based parallel computing on Smith-Waterman algorithm to sequencedatabase searches", *International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, IEEE, pp. 137-142, 2013.
15. Meyer U., Sanders P. 2003. -stepping: a parallelizable shortest path algorithm. *J. of Algorithms* 49, pp. 114–152.
16. Bader D.A., Madduri K. 2006. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. *ICPP*, pp. 523–530.
17. Daniel Lorenz, Peter Philippen, Dirk Schmidl and Felix Wolf. *International Conference on Parallel Processing Workshops "Profiling of OpenMP Tasks with Score-P"* German Research School for Simulation Sciences, 52062 Aachen, Germany.[2012]
18. D. Dheeraj, B. Nitish, Shruti Ramesh, *International Conference on Cyber Enabled Distributed Computing and Knowledge Discover "Optimization of Automatic Conversion of Serial C to Parallel OpenMP"*, PES Institute of Technology Bangalore, India.[2012]
19. Suneeta H. Angadi, G. T. Raju Abhishek B, *International Journal of Computer Science and Informatics "Software Performance Analysis with Parallel Programming Approaches"*, ISSN (PRINT): 2231 -5292, Vol-1, Iss-4, 2012.
20. Sanjay Kumar Sharma, Dr. Kusum Gupta, *International Journal of Computer Science, Engineering and Information Technology (IJCEIT)*, "Performance Analysis of Parallel Algorithms on Multi-core System using OpenMP Programming Approaches", Vol.2, No.5, October 2012.
21. Han Cao^{1a}, Fei Wang^b, Xin Fang, Hong-lei Tu, Jun Shi *World Congress on Software Engineering*, "OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis", DOI 10.1109 WCSE.2009.[6]Javier Diaz,
22. Camelia Munoz-Caro, and Alfonso Nino, *IEEE TRANSACTIONS ON*

PARALLEL AND DISTRIBUTED SYSTEMS, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era" VOL. 23, NO. 8, AUGUST 2012.

23.Songmin Jia, Xiaolin Yin, and Xiuzhi Li, IEEE, International Conference on Robotics and Biomimetics "Mobile Robot Parallel PF-SLAM Based on OpenMP", December 2012.

24.Paul Graham,Edinburgh Parallel Computing Centre "A Parallel Programming Model forShared Memory Architectures" The University of Edinburgh March 2011.

25.R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. Applied Numerical Mathematics, 13:437–452, 1994.

26.Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In Proceedings of the Fourth SIAM International Conference on Data Mining (2004), Lake Buean Vista, FL, USA, 22–24 April 2004.

27.Eugen Dedu, Stephane´ Vialle, and Claude Timsit. Comparison of OpenMP and classical multi-threading parallelization for regular and irregular algorithms. In In proceesing of Software Engineering Ap-plied to Networking & Parallel/Distributed Computing (SNPD 2000), Champagne-Ardenne, France, pages 53–60, 19–21 May 2000.

28.Dixie Hisley, Gagan Agrawal, Punyam Satya-narayana, and Lori Pol-lock. Porting and performance evaluation of irregular codes using OpenMP. In In proceesing of First European Workshop on OpenMP (EWOMP 1999), Lund, Sweden, pages 47–59, 1999.

Parallel and Distributed Computing Review –2

Aratrik Paul-16BCE0519
Shikhar Singh – 16BCE2316

Topic: Comparison of Parallel and Serial Run Time of Shortest Path Algorithms

1) Bellman-Ford Algorithm

The **Bellman–Ford algorithm** is an [algorithm](#) that computes [shortest paths](#) from a single source [vertex](#) to all of the other vertices in a [weighted digraph](#).^[1] It is slower than [Dijkstra's algorithm](#) for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm was first proposed by Alfonso Shimbel in 1955, but is instead named after [Richard Bellman](#) and [Lester Ford, Jr.](#), who published it in 1958 and 1956, respectively. [Edward F. Moore](#) also published the same algorithm in 1957, and for this reason it is also sometimes called the **Bellman–Ford–Moore algorithm**.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.^[3] If a graph contains a "negative cycle" (i.e. a [cycle](#) whose edges sum to a negative value) that is reachable from the source, then there is no *cheapest* path: any path that has a point on the negative cycle can be made cheaper by one more [walk](#) around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence.

Areas of Parallelism
Screenshots:

```
Activities Text Editor Sat 1:20 PM
openmp_bellman_ford.cpp
Save
```

```
FinalAPSP.cpp * openmp_bellman_ford.cpp *

#include <string>
#include <cassert>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <omp.h>
#include <string>
#include <sys/time.h>

#include "omp.h"

using std::string;
using std::cout;
using std::endl;

#define INF 1000000

/**
 * utils is a namespace for utility functions
 * including I/O (read input file and print results) and matrix dimension convert(2D->1D) function
 */
namespace utils {
    int N; //number of vertices
    int *mat; // the adjacency matrix

    void abort_with_error_message(string msg) {
        std::cerr << msg << endl;
        abort();
    }

    //translate 2-dimension coordinate to 1-dimension
    int convert_dimension_2D_1D(int x, int y, int n) {
        return x * n + y;
    }

    int read_file(string filename) {
```

```
C++ Tab Width: 8 Ln 128, Col 47 INS
Activities Text Editor Sat 1:22 PM
openmp_bellman_ford.cpp
Save
```

```
FinalAPSP.cpp * openmp_bellman_ford.cpp *

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            inputf >> mat[convert_dimension_2D_1D(i, j, N)];
        }
    return 0;
}

int print_result(bool has_negative_cycle, int *dist) {
    std::ofstream outputf("output.txt", std::ofstream::out);
    if (!has_negative_cycle) {
        for (int i = 0; i < N; i++) {
            if (dist[i] > INF)
                dist[i] = INF;
            outputf << dist[i] << '\n';
        }
        outputf.flush();
    } else {
        outputf << "FOUND NEGATIVE CYCLE!" << endl;
    }
    outputf.close();
    return 0;
}
} //namespace utils

/**
 * Bellman-Ford algorithm. Find the shortest path from vertex 0 to other vertices.
 * @param p number of processes
 * @param n input size
 * @param *mat input adjacency matrix
 * @param *dist distance array
 * @param *has_negative_cycle a bool variable to recode if there are negative cycles
 */
void bellman_ford(int p, int n, int *mat, int *dist, bool *has_negative_cycle) {
```

```
C++ Tab Width: 8 Ln 128, Col 47 INS
```

Description of the Parallel Platform of Interest.

OpenMP 4.5

OpenMP (Open Multi-Processing) is an [application programming interface](#) (API) that supports multi-platform [shared memory multiprocessing](#) programming in [C](#), [C++](#), and [Fortran](#),^[3] on most platforms, [instruction set architectures](#) and [operating systems](#), including [Solaris](#), [AIX](#), [HP-UX](#), [Linux](#), [macOS](#), and [Windows](#). It consists of a set of [compiler directives](#), [library routines](#), and [environment variables](#) that influence run-time behavior.

OpenMP is managed by the [nonprofit technology consortium](#) *OpenMP Architecture Review Board* (or *OpenMP ARB*), jointly defined by a group of major computer hardware and software vendors, including [AMD](#), [IBM](#), [Intel](#), [Cray](#), [HP](#), [Fujitsu](#), [Nvidia](#), [NEC](#), [Red Hat](#), [Texas Instruments](#), [Oracle Corporation](#), and more.

OpenMP uses a [portable](#), scalable model that gives [programmers](#) a simple and flexible interface for developing parallel applications for platforms ranging from the standard [desktop computer](#) to the [supercomputer](#).

OpenMP+SIMD: `#pragma omp simd` is a command in the new OpenMP4.0 which allows us to follow Throughwith the Single Instruction and Muliple data processes as it is futile if we don't use this command and have multiple data on which we have to go through the similar Instruction.

Sample Program and Steps to download:

```

STATIC_INLINE DATA_TYPE norm(DATA_TYPE *a, int a_size)
{
    DATA_TYPE sum = 0;
#pragma omp simd reduction(+:sum)
    for (int i = 0; i < a_size; i++)
    {
        sum = sum + a[i] * a[i];
    }
    return sqrt(sum);
}

```

The norm of matrix with SIMD vectorization:

SIMD FUNCTION VECTORIZATION:

```

float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}

```

OpenMP4.0 + SIMD Code for alignment:

```

float * a;
posix_memalign(&a, ...);

```

```
float * b;
posix_memalign(&b, ...);
float s;
...
__assume_aligned(a, 32); // Intel
__assume_aligned(b, 32); // Intel
#pragma omp simd
for (int i = 0; i < N; i++)
{
a[i] += s * b[i];
}
```

SIMD programming :
WE ARE USING OPENMP 4.5

STEP 1: INSTALL OPENMP & GCC (G++ COMPILER)

OPENMP

```
sudo apt install libomp-dev
```

```
apt show libomp-dev
```

OR

GO TO <https://www.openmp.org/resources/openmp-compilers-tools/>
AND select OPENMP FOR YOUR COMPILER VERSION

GCC COMPILER

```
sudo apt-get update
```

```
sudo apt-get install g++
```

STEP 2:(OPTIONAL)

INSTALL BUILD TOOLS

```
sudo apt-get install build-essential
```

```
sudo apt-get install -y pkg-config
```

```
sudo apt-get install gtk2.0
```

```
sudo apt install intltool
```

COMPARISON BETWEEN PARALLEL AND SERIAL ALGORITHMS USING OpenMP+SIMD

A PROJECT REPORT FOR J COMPONENT

PARALLEL AND DISRIBUTED COMPUTING (CSE 4001)

PROJECT SUPERVISOR- **PROF. SAIRA BANU J**

Submitted by

SHIKHAR SINGH -16BCE2316

ARATRIK PAUL-16BCE0519

School of Computer Science Engineering



Nov. 2017

School of Computer Science and Engineering

CERTIFICATE

This is to certify that the project entitled, "Comparision between Parallel and serial algorithms using OpenMP +SIMD" submitted by the students Shikhar Singh=16BCE2316 and Aratrik Paul =16BCE0519 for the course "Parallel and Distibuted Computing(CSE-4001)" at "VIT university", is an authentic work carried out by the members of the team under my supervision and guidance. To the best of my knowledge, the matter embodied in the project has not been submitted to any other University / Institute for any purpose and is unique.

Date:

Place:

Signature

ACKNOWLEDGEMENTS

We would like to extend our thanks to the management of VIT University, Vellore and our respected chancellor G. Viswanathan for giving us the opportunity to carry out our project in VIT.

We would also like to extend our gratitude to our faculty, Prof. Saira Banu J for her constant and ceaseless guidance and support through the entire course of this project. We would also want to thank ma'am for guiding us all along the way and encouraging us to think out of the box in an attempt to maximize the project's validity for the society.

Shikhar Singh-16BCE2316

Aratrik Paul – 16BCE0519

ABSTRACT

We the students of VIT, Vellore have made a project on “ Comparison of Parallel and serial Algorithms using OpenMP +SIMD ” as our Parallel and Distributed Computing project for the fifth semester. Our goal was to give an overview of the Analysis and speedup of Different Algorithms such as Dijkstra’s Algorithm and find out how effective it would be if it were parallelized.

Table of Contents

- 1. INTRODUCTION**
- 2. LITERATURE SURVEY**
- 3. IMPLEMENTATION DETAILS**
- 4. RESULTS AND DISCUSSIONS**
- 5. PARAMETERS TO BE CONSIDERED**
- 6. CONCLUSION**
- 7. REFERENCES**

1) INTRODUCTION

OpenMP (Open Multi-Processing) is an application programming interface (API) that helps us use a multiple platform shared memory multi - processing programming in C, C++ and/or Fortran, on most platforms, the instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of directives and library routines that influence run-time behavior.

OpenMP is managed by the non for profit technology Company OpenMP Architecture Review Board or OpenMP ARB, jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and many more.

OpenMP uses a portable and scalable model that gives programmers a simple interface for developing parallel applications for platforms may it be on a computer, Desktop or a Supercomputer.

An app built with the hybrid model of parallel programming can run on a Desktop cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a multiple core node while MPI is used for parallelism amongst nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

OpenMP is the way to implement multithreading, a method of parallelizing where a master thread forks a specified number of slave threads (as it uses the Master Slave Model) and the system divides a task among them. The threads then run at the same time, with the Main Environment allocating threads to different processors.

The Part of the code that is meant to run in the parallel area is marked accordingly, with a compiler directive that forms threads before the section is executed. Each thread has an id attached to it which can be obtained using a function `omp_get_thread_num()`. The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads report back to the master Program and hence we have a completely compiled parallel program.

By default, each thread executes the parallelized section of code independently. Work-sharing can be used to divide a task among the threads so that each thread executes its allocated part of the code. We can achieve both the parallelism methods of task and data parallelism by using OpenMP this way.

The environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on

environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `<omp.h>` in C/C++.

SIMD

Single instruction, multiple data (SIMD) comes under the class of parallel computers in Flynn's taxonomy. It talks about computers with multiple processing elements that perform the same operation on multiple data points at the same point of time. These machines take advantage of data level parallelism, but not concurrency: there are parallel computations, but only a single process at a given moment. SIMD is particularly applicable to normal problems such as adjusting the volume of digital audio. Most modern CPU designs use SIMD instructions to improve the performance of multimedia use.

SIMD was the basis for vector supercomputers of the early 1970s such as the CDC Star-100 and the Texas Instruments ASC, which could operate on a "vector" of data with a single instruction. Vector processing was especially popularized by Cray in the 1970s and 1980s[1]. Vector-processing architectures are now considered separate from SIMD computers, based on the fact that vector computers processed the vectors one word at a time through pipelined processors (though still based on a single instruction), whereas modern SIMD computers process all elements of the vector simultaneously.

All of these developments have been made toward support for real-time graphics, and are therefore brought forward toward processing in two, three, or four dimensions, usually with vector lengths of between two and sixteen words, depending on its data type and architecture. When we want to make the difference between the SIMD architectures of old and the newer architectures, the newer are considered "short-vector" architectures, as earlier SIMD and vector supercomputers had vector lengths from 64 to 64,000. A modern supercomputer is almost always a cluster of MIMD computers, each of which implements (short-vector) SIMD instructions. A modern desktop computer is often a multiprocessor MIMD computer where each processor can execute short-vector SIMD instructions.

The Algorithms we have chosen to do the project are Floyd Warshall's Algorithm and Bellman Ford's Algorithm.

A) Bellman Ford's Algorithm

The Bellman Ford algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights with no negative cycles. A single execution of the algorithm will find the lengths (summed weights) of 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. Versions of the algorithm can also be used for finding the transitive closure of a relation R , or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

2) Literature Review

1) Performance of Shortest Path Algorithm Based on Parallel Vertex Traversal

Written by- Mihailo Vesović, Aleksandra Smiljanić, Dušan Kostić

Date Published: February 2016

Abstract: Shortest path algorithms for different applications, such as Internet routing, VLSI design and so on are used. Floyd Warshall and Bellman-Ford are commonly used shortest path algorithms which are typically implemented in networks with hundreds of nodes. However, scale of shortest path problems is increasing, and more efficient algorithms are needed. With the development of multicore processors, one natural way to speedup shortest path algorithms is through parallelization. In this paper, we propose a novel shortest path algorithm with parallel vertex transversal, and compare its speed with standard solutions in datacenter topologies.

2) Performance Anaysis of Parallel Algorithms on Multi-Core Systems

Written by: Sanjay Kumar Sharma and Dr Kusum Bhansali

Date Published: October 2012

Abstarct: The current multi-core architectures have become popular due to performance, and efficient processing of multiple tasks simultaneously. Today's the parallel algorithms are focusing on multi-core systems. The design of parallel algorithm and performance measurement is the major issue on multi-core environment. If one wishes to execute a

single application faster, then the application must be divided into subtask or threads to deliver desired result. Numerical problems, especially the solution of linear system of equation have many applications in science and engineering. This paper describes and analyzes the parallel algorithms for computing the solution of dense system of linear equations, and to approximately compute the value of using OpenMP interface. The performances (speedup) of parallel algorithms on multi-core system have been presented. The experimental results on a multi-core processor show that the proposed parallel algorithms achieves good performance (speedup) compared to the sequential.

3) Parallel Computing using OpenMP

Written By: Ms. Ashwini M. Bhugul

Date Published: February ,2017

Abstract: Parallel Computing is most widely used paradigm today to improve the system performance. System performance can be increase by executing application on multiple processors. Today's Computers are complex they are capable of executing application program on multiple processors. OpenMP is API for running application parallelly to improve the speedup. This paper reviews about OpenMP API and focuses on improving the system performance.

4) OpenMP: an industry standard API for shared-memory programming

Written by : L Dagum, R Menon

Date Published : Jan,2010

Abstract: At its most elemental level, OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++ to express shared memory parallelism. It leaves the base language unspecified, and vendors can implement OpenMP in any Fortran compiler. Naturally, to support pointers and allocatables, Fortran 90 and Fortran 95 require the OpenMP implementation to include additional semantics over Fortran 77. OpenMP leverages many of the shared mamory concepts while extending them to support coarse grain parallelism. The standard also includes a callable runtime library with accompanying environment variables.

5) High performance computing using MPI and OpenMP on multi-core parallel systems

Written by: Haoqiang Jin, Piyush Mehrotra

Date Published:March,2012

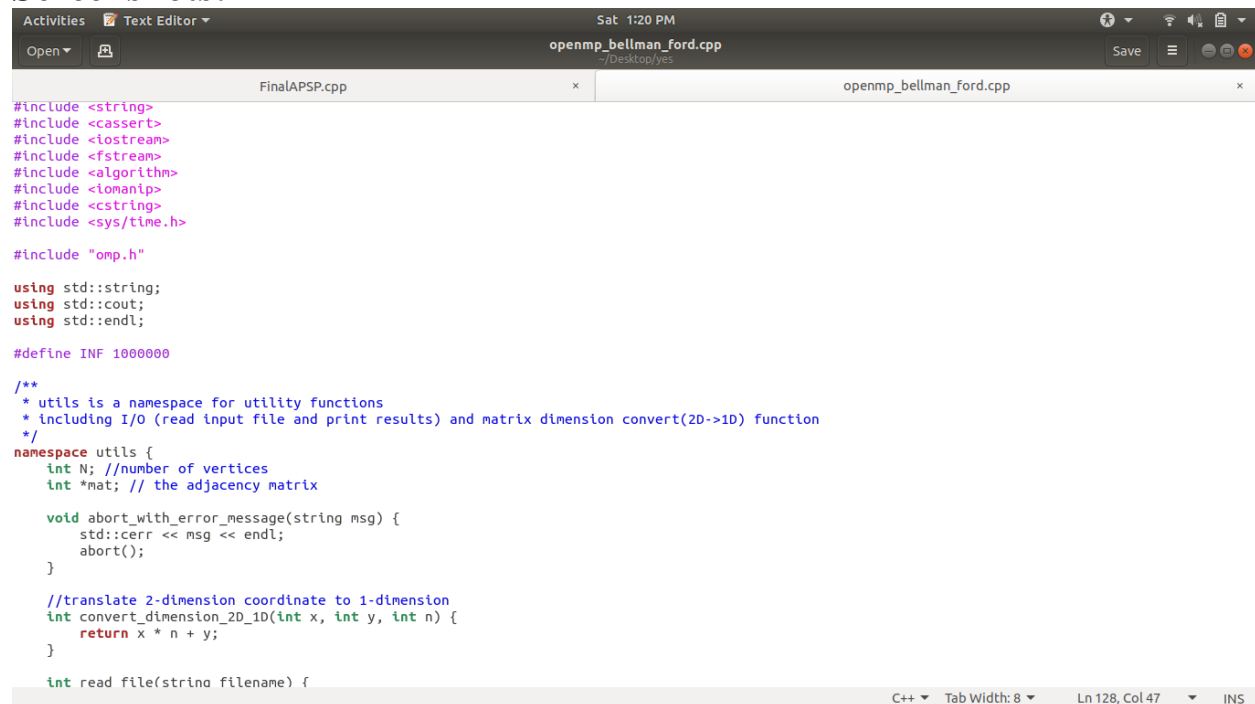
Abstract: The rapidly increasing number of cores in modern microprocessors is pushing the current high performance computing (HPC) systems into the petascale and exascale era. The hybrid nature of these systems – distributed memory across nodes and shared

memory with non-uniform memory access within each node – poses a challenge to application developers. In this paper, we study a hybrid approach to programming such systems – a combination of two traditional programming models, MPI and OpenMP

3) Implementation

We have gone ahead to implement this project on the Ubuntu linux 18.04 LTS system. Where we have used c++ codes and intrinsically installed the gcc compiler and omp libraries used to run an OpenMP Program. The two codes are for Floyd Warshall's and Bellman Ford's Parallel codes using OpenMP + SIMD.

Screenshots:



```
#include <string>
#include <cassert>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iomanip>
#include <cstring>
#include <sys/time.h>

#include "omp.h"

using std::string;
using std::cout;
using std::endl;

#define INF 1000000

/**
 * utils is a namespace for utility functions
 * including I/O (read input file and print results) and matrix dimension convert(2D->1D) function
 */
namespace utils {
    int N; //number of vertices
    int *mat; // the adjacency matrix

    void abort_with_error_message(string msg) {
        std::cerr << msg << endl;
        abort();
    }

    //translate 2-dimension coordinate to 1-dimension
    int convert_dimension_2D_1D(int x, int y, int n) {
        return x * n + y;
    }

    int read_file(string filename) {
```



```
Activities Text Editor Sat 1:22 PM
openmp_bellman_ford.cpp
~/Desktop/yes
Save
```

```
FinalAPSP.cpp x openmp_bellman_ford.cpp x

for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
        inputf >> mat[convert_dimension_2D_1D(i, j, N)];
    }
return 0;
}

int print_result(bool has_negative_cycle, int *dist) {
    std::ofstream outputf("output.txt", std::ofstream::out);
    if (!has_negative_cycle) {
        for (int i = 0; i < N; i++) {
            if (dist[i] > INF)
                dist[i] = INF;
            outputf << dist[i] << '\n';
        }
        outputf.flush();
    } else {
        outputf << "FOUND NEGATIVE CYCLE!" << endl;
    }
    outputf.close();
    return 0;
}
} // namespace utils

/**
 * Bellman-Ford algorithm. Find the shortest path from vertex 0 to other vertices.
 * @param p number of processes
 * @param n input size
 * @param *mat input adjacency matrix
 * @param *dist distance array
 * @param *has_negative_cycle a bool variable to recode if there are negative cycles
 */
void bellman_ford(int p, int n, int *mat, int *dist, bool *has_negative_cycle) {
```

```
C++ Tab Width: 8 Ln 128, Col 47 INS
Activities Text Editor Sat 1:24 PM
openmp_bellman_ford.cpp
~/Desktop/yes
Save
```

```
*has_negative_cycle = false;

//step 1: set openmp thread number
omp_set_num_threads(p);

//step 2: find local task range
int ave = n / p;
#pragma omp parallel for
for (int i = 0; i < p; i++) {
    local_start[i] = ave * i;
    local_end[i] = ave * (i + 1);
    if (i == p - 1) {
        local_end[i] = n;
    }
}

//step 3: bellman-ford algorithm
//initialize distances
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    dist[i] = INF;
}
//root vertex always has distance 0
dist[0] = 0;

int iter_num = 0;
bool has_change;
bool local_has_change[p];
#pragma omp parallel
{
    int my_rank = omp_get_thread_num();
    //bellman-ford algorithm
    for (int iter = 0; iter < n - 1; iter++) {
        local_has_change[my_rank] = false;
        for (int u = 0; u < n; u++) {
            for (int v = local_start[my_rank]; v < local_end[my_rank]; v++) {

```

```
Activities Text Editor Sat 1:24 PM
openmp_bellman_ford.cpp
~/Desktop/yes
FinalAPSP.cpp openmp_bellman_ford.cpp x
has_change = false;
for (int rank = 0; rank < p; rank++) {
    has_change |= local_has_change[rank];
}
}
if (!has_change) {
    break;
}
}
}
#pragma omp parallel reduction(|:has_change)
//do one more iteration to check negative cycles
if (iter_num == n - 1) {
    has_change = false;
    for (int u = 0; u < n; u++) {
#pragma omp parallel for simd reduction(|:has_change) // #pragma omp for simd
        for (int v = 0; v < n; v++) {
            int weight = mat[u * n + v];
            if (weight < INF) {
                if (dist[u] + weight < dist[v]) { // if we can relax one more step, then we find a negative cycle
                    has_change = true;
                }
            }
        }
    }
    *has_negative_cycle = has_change;
}

//step 4: free memory (if any)
}

int main(int argc, char **argv) {
    if (argc <= 1) {
        utils::abort_with_error_message("INPUT FILE WAS NOT FOUND!");
    }
}
```

```
C++ Tab Width: 8 Ln 128, Col 47 INS
Activities Text Editor Sat 1:26 PM
FinalAPSP.cpp
~/Desktop/yes
Save
#include <iostream>
#include <fstream>
#include <climits>
#include <string>
#include <sstream>
#include <omp.h>
using namespace std;
void updateSubmatrix(int a_row, int a_col, int b_row, int b_col, int c_row, int c_col, int **A, int b)
{
    //This will update the tile which is having dimension bxb
    int i, j, k;
    for (k = 0; k < b; k++)
    {
        for (i = 0; i < b; i++)
            for (j = 0; j < b; j++)
                if (A[b_row+i][b_col+k] < INT_MAX && A[c_row+k][c_col+j] < INT_MAX)
                    A[a_row+i][a_col+j] = min(A[a_row+i][a_col+j], (A[b_row+i][b_col+k] + A[c_row+k][c_col+j]));
    }
}

void tiledFloydWarshall(int **A, int n, int b, int t)
{
    //ntrow is for number of times kth-block will iterate
    //nbrow is actually the dimension of each tile nbrowXnbrow
    int ntrow = n/b;
    int nbrow = n/ntrow;
    int k, i, j;
    omp_set_num_threads(t);

    for (k = 0; k < ntrow; k++) //Iterator to position our main tile.
    {
        //Phase 1: updates the k-th diagonal-tile(main tile)
        updateSubmatrix(k*nbrow, k*nbrow, k*nbrow, k*nbrow, k*nbrow, k*nbrow, A, b);

        //Phase 2: updates the tiles which are remained in the main tile row.
        #pragma omp parallel for
        for (j = 0; j < ntrow; j++)
            for (i = 0; i < ntrow; i++)
                updateSubmatrix(k*nbrow, i*nbrow, k*nbrow, i*nbrow, i*nbrow, j*nbrow, A, b);
    }
}
```

Activities Text Editor Sat 1:24 PM

openmp_bellman_ford.cpp
~/Desktop/yes

Save

FinalAPSP.cpp x openmp_bellman_ford.cpp x

```
if (argc <= 2) {
    utils::abort_with_error_message("NUMBER OF THREADS WAS NOT FOUND!");
}
string filename = argv[1];
int p = atoi(argv[2]);

int *dist;
bool has_negative_cycle = false;

assert(utils::read_file(filename) == 0);
dist = (int *) malloc(sizeof(int) * utils::N);

//time counter
timeval start_wall_time_t, end_wall_time_t;
float ms_wall;

//start timer
gettimeofday(&start_wall_time_t, nullptr);

//bellman-ford algorithm
bellman_ford(p, utils::N, utils::mat, dist, &has_negative_cycle);

//end timer
gettimeofday(&end_wall_time_t, nullptr);
ms_wall = ((end_wall_time_t.tv_sec - start_wall_time_t.tv_sec) * 1000 * 1000
           + end_wall_time_t.tv_usec - start_wall_time_t.tv_usec) / 1000.0;

std::cerr.setf(std::ios::fixed);
std::cerr << std::setprecision(6) << "Time(s): " << (ms_wall / 1000.0) << endl;
utils::print_result(has_negative_cycle, dist);
free(dist);
free(utils::mat);

return 0;
}
```

C++ Tab Width: 8 Ln 128, Col 47 INS

```
Activities Text Editor Sat 1:27 PM
FinalAPSP.cpp
~/Desktop/yes

//Phase 3:updates the tiles which are remained in the main tile column.
#pragma omp parallel for
for(i=0;i<ntrow;i++)
{
    if(i==k){}
    else
    {
        updateSubmatrix(i*nbrow, k*nbrow,i*nbrow, k*nbrow,k*nbrow, k*nbrow,A,b);
    }
}

//Phase 4:finally it updates the remaining tiles except that main tile row and column of the matrix
#pragma omp parallel for collapse(2)
for(i=0;i<ntrow;i++)
{
    for(j=0;j<ntrow;j++)
    {
        if(i==k||j==k){}
        else
        {
            updateSubmatrix(i*nbrow, j*nbrow,i*nbrow, k*nbrow,k*nbrow, j*nbrow,A,b);
        }
    }
}

}
}
struct Graph {
    int **A;
    int VertexCount;
    int EdgeCount;
};
int main()
{
    int i,j;
```

```
C++ Tab Width: 8 Ln 119, Col 51 INS
Activities Text Editor Sat 1:27 PM
FinalAPSP.cpp
~/Desktop/yes

int t;
cout<<"Enter Number of Threads :";
cin>>t;
omp_set_num_threads(t);

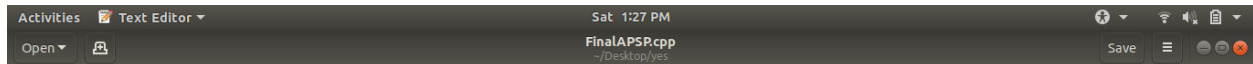
int n,dadd=0,dr;

Graph G;
int **D;
bool declared = false;
string GraphType, token;

cout << endl << "Taking Graph from sample3.txt and "<<INT_MAX<<" means Infinity :\n" << endl;
ifstream fl("./sample3.txt");
for(string line; getline(fl, line);)
{
    if (line[0] == 'c')
        continue;
    else if (line[0] == 'p')
    {
        istringstream iss(line);
        iss >> token >> GraphType >> G.VertexCount >> G.EdgeCount;
        n=G.VertexCount;
        dr=n%b;
        if(dr!=0)
            dadd=b-dr;

        if (G.VertexCount > 0)
        {
            G.A=new int*[G.VertexCount+dadd];
            D= new int*[G.VertexCount+dadd];
            for(int i=0;i<G.VertexCount+dadd;i++)
            {
                G.A[i]=new int[G.VertexCount+dadd];
                D[i]=new int[G.VertexCount+dadd];
            }

            #pragma omp parallel for collapse(2)
```

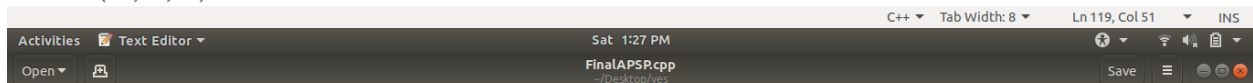


```
    }
    else if(declared && line[0] == 'a')
    {
        int w;
        istringstream iss(line);
        iss >> token >> i >> j >> w;
        G.A[i-1][j-1] = w;
    }
}

//Copying Graph 'G.A' to 'D' to apply standard floyd-warshall algorithm on 'D'.
#pragma omp parallel for collapse(2)
for(i=0;i<n+dadd;i++)
{
    for(j=0;j<n+dadd;j++)
    {
        D[i][j]=G.A[i][j];
    }
}

/* //'D' weight graph, Before Applying Standard Floyd-warshall Algorithm.
cout<<"Parallelized Standard Floyd-Warshall Algorithm :\n\n";
cout<<"Input Graph :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        cout<<D[i][j]<<" ";
    cout<<"\n";
}
*/

//Applying Parallelization to Standard Floyd-Warshall Algorithm and Calculating the Runtime for that Algorithm.
double tm0 = omp_get_wtime();
for(int k=0;k<n;k++)
#pragma omp parallel for collapse(2)
for(i=0;i<n;i++)
```



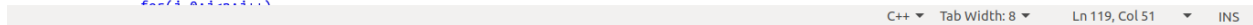
```
    }
}
tm0= omp_get_wtime() - tm0;

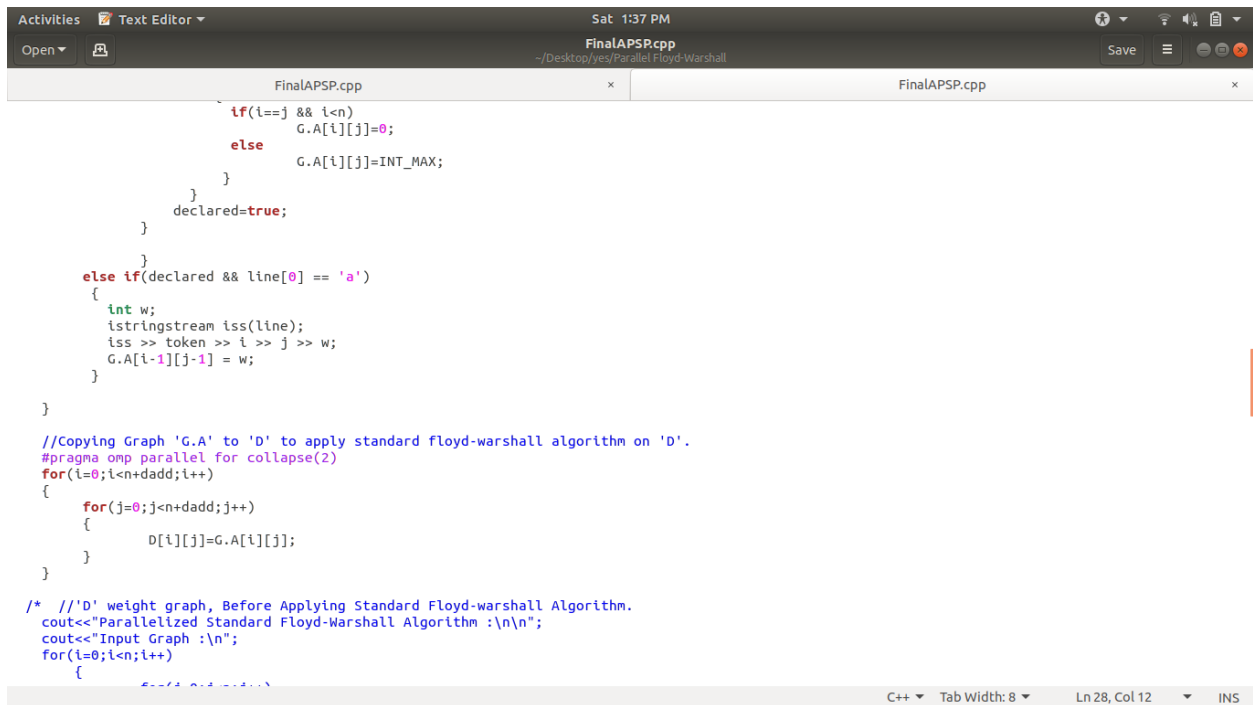
/* //'D' weight graph, After Applying Standard Floyd-warshall Algorithm i.e. Shortest path matrix.
cout<<"All Pairs Shortest Path Matrix :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        cout<<D[i][j]<<" ";
    cout<<"\n";
}
*/
cout << "Parallelized Standard FloydWarshall Algorithm Runtime is: " << tm0 << " seconds.\n\n";

/* //'G.A' weight graph, Before Applying Tiled Floyd-warshall Algorithm.
cout<<"Parallelized Tiled Floyd-Warshall Algorithm :\n\n";
cout<<"Input Graph :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        cout<<G.A[i][j]<<" ";
    cout<<"\n";
}
*/

//Applying Parallelization to Tiled Floyd-Warshall Algorithm and Calculating the Runtime for that Algorithm.
double tm = omp_get_wtime();
tiledFloydWarshall(G.A,n+dadd,b,t);
tm = omp_get_wtime() - tm;

/* //'G.A' weight graph, After Applying Tiled Floyd-warshall Algorithm i.e. Shortest path matrix.
cout<<"All Pairs Shortest Path Matrix :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
```





```
FinalAPSP.cpp
~ /Desktop/yes/Parallel Floyd-Warshall
Save
FinalAPSP.cpp
FinalAPSP.cpp

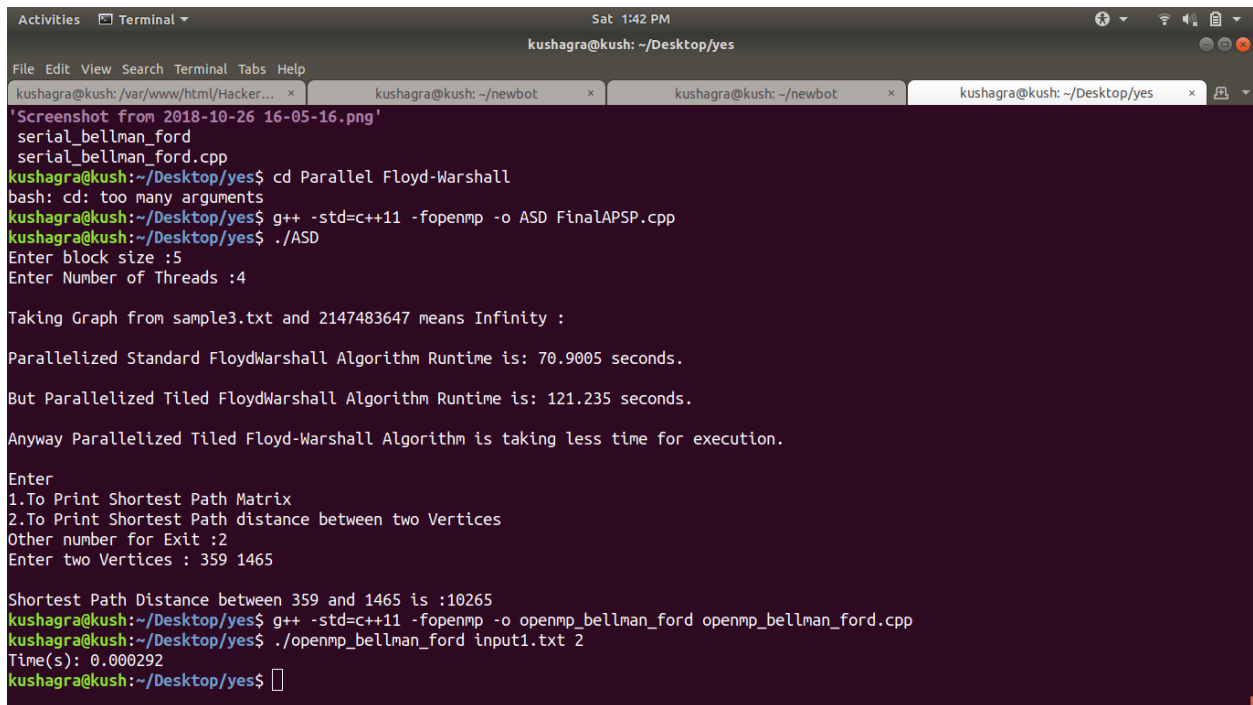
if(i==j && i<n)
    G.A[i][j]=0;
else
    G.A[i][j]=INT_MAX;
}
}
declared=true;
}

else if(declared && line[0] == 'a')
{
    int w;
    istream iss(line);
    iss >> token >> i >> j >> w;
    G.A[i-1][j-1] = w;
}
}

//Copying Graph 'G.A' to 'D' to apply standard floyd-warshall algorithm on 'D'.
#pragma omp parallel for collapse(2)
for(i=0;i<n+dadd;i++)
{
    for(j=0;j<n+dadd;j++)
    {
        D[i][j]=G.A[i][j];
    }
}

/* //'D' weight graph, Before Applying Standard Floyd-warshall Algorithm.
cout<<"Parallelized Standard Floyd-Warshall Algorithm : \n\n";
cout<<"Input Graph : \n\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        cout<<D[i][j]<<" ";
    }
    cout<<\n;
}
```

4)Results and Discussions



```
Activities Terminal
Sat 1:42 PM
kushagra@kush: ~/Desktop/yes
File Edit View Search Terminal Tabs Help
kushagra@kush: /var/www/html/Hacker... x kushagra@kush: ~/newbot x kushagra@kush: ~/newbot x kushagra@kush: ~/Desktop/yes x
'Screenshot from 2018-10-26 16-05-16.png'
serial_bellman_ford
serial_bellman_ford.cpp
kushagra@kush:~/Desktop/yes$ cd Parallel Floyd-Warshall
bash: cd: too many arguments
kushagra@kush:~/Desktop/yes$ g++ -std=c++11 -fopenmp -o ASD FinalAPSP.cpp
kushagra@kush:~/Desktop/yes$ ./ASD
Enter block size :5
Enter Number of Threads :4

Taking Graph from sample3.txt and 2147483647 means Infinity :

Parallelized Standard FloydWarshall Algorithm Runtime is: 70.9005 seconds.

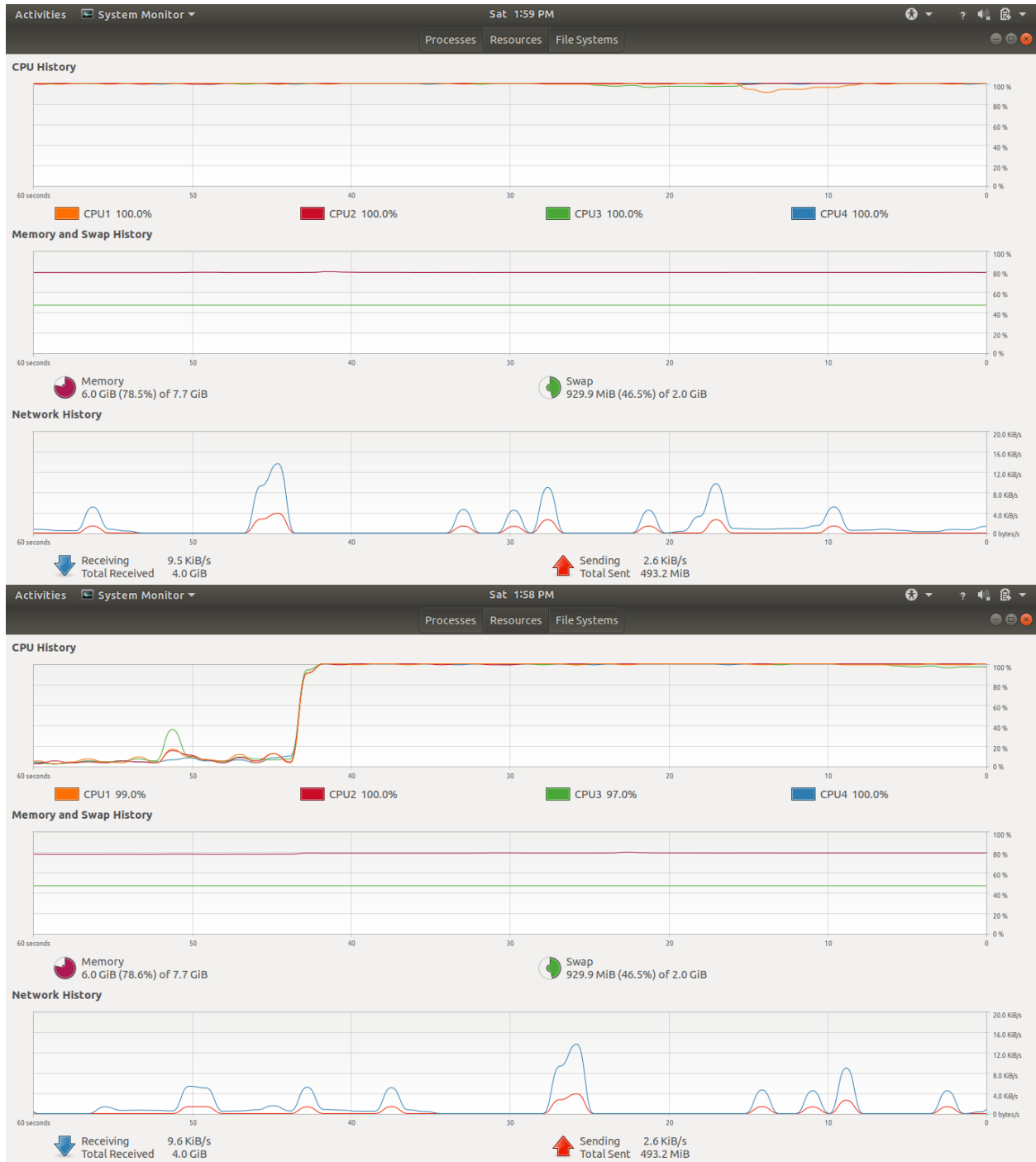
But Parallelized Tiled FloydWarshall Algorithm Runtime is: 121.235 seconds.

Anyway Parallelized Tiled Floyd-Warshall Algorithm is taking less time for execution.

Enter
1.To Print Shortest Path Matrix
2.To Print Shortest Path distance between two Vertices
Other number for Exit :2
Enter two Vertices : 359 1465

Shortest Path Distance between 359 and 1465 is :10265
kushagra@kush:~/Desktop/yes$ g++ -std=c++11 -fopenmp -o openmp_bellman_ford openmp_bellman_ford.cpp
kushagra@kush:~/Desktop/yes$ ./openmp_bellman_ford input1.txt 2
Time(s): 0.000292
kushagra@kush:~/Desktop/yes$
```

5) Parameters to be Considered





6) Conclusions

There is certainly a decrease in computation of Bellman Ford's Algorithm and Floyd Warshall's Algorithm when we use OpenMP +SIMD in the Algorithm as it uses all the 4 CPU's of the Computer as thought of. After parallelising the code was seen to be smoother and the output remained to result to us faster.

7) References

- T.H. Cormen, E.C. Leiserson, R.L. Rivest: Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 1990.
- A. Smiljanić, N. Maksić: Improving Utilization of Data Center Networks, IEEE Communication Magazine, Vol. 51, No. 11, Nov. 2013, pp. 32 – 38.
- A. Smiljanić, J. Chao, C. Minkenberg, E. Oki, M. Hamdi: Switching and Routing for Scalable and Energy-Efficient Networking, Vol. 32, No. 1, Jan. 2014, pp. 1 – 3.
- E.F. Moore: The Shortest Path through a Maze, International Symposium on the Theory of Switching, Cambridge, MA, USA, 02-05 April 1957
- E.W. Dijkstra: A Note on Two Problems in Connection with Graphs, Numerische Mathematik, Vol. 1, No. 1, Dec. 1959, pp. 269 – 271. [

- M.L. Fredman R.E. Tarjan: Fibonacci Heaps and their uses in Improved Network Optimization Algorithms, Journal of the ACM, Vol. 34, No. 3, July 1987, pp. 596 – 615. [
- D. Dundjerski, M. Tomasević: Graphical Processing Unit-based Parallelization of the Open Shortest Path First and Border Gateway Protocol Routing Protocols, Concurrency and Computation: Practice and Experience, Vol. 27, No. 1, Jan. 2015, pp. 237 – 251.
- J. Kim, W.J. Dally, S. Scott, D. Abts: Technology-driven, Highly-scalable Dragonfly Topology, International Symposium on Computer Architecture, Beijing, China, 21-25 June 2008, pp. 77 – 88.
- M. Al-Fares, A. Loukissas, A. Vahdat: A Scalable, Commodity Data Center Network Architecture, Conference SIGCOMM 2008, Seattle, WA, USA, 17-22 Aug. 2008, pp. 63 – 74.
- M. Besta, T. Hoefer: Slim Fly: A Cost Effective Low-diameter Network Topology, International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16-21 Nov. 2014, pp. 348 – 359.
- S. Mozes and C. Sommer. Exact distance oracles for planar graphs. SODA, 2012.
- G. Nannicini, P. Baptiste, G. Barbier, D. Krob, and L. Liberti. Fast paths in large-scale dynamic road networks. Computational Optimization and Applications, 2010.