Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0

Eid Albalawi
Department of Computer Science
University Of Manitoba
Winnipeg, Manitoba
Email: albalawi@cs.umanitoba.ca

Parimala Thulasiraman
Department of Computer Science
University Of Manitoba
Winnipeg, Manitoba
Email: thulasir@cs.umanitoba.ca

Ruppa Thulasiram
Department of Computer Science
University Of Manitoba
Winnipeg, Manitoba
Email: tulsi@cs.umanitoba.ca

***Abstract*—OpenMP is a standard parallel programming language 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. In irregular applications, the load changes dynamically at runtime and distribution of work among the threads can be done only at runtime. In the literature, it has been shown that OpenMP produces poor performance for irreg-ular applications. In 2008, the OpenMP 3.0 version introduced new features such as "tasks" to handle irregular computations. Not much work has gone into studying irregular algorithms in OpenMP 3.0. In this paper, we consider one graph problem, the all pair shortest path problem and its implementation in OpenMP 3.0. We show that for large number of vertices, the algorithm running on OpenMP 3.0 surpasses the one on OpenMP 2.5 by 1.6 times.**

Keywords-OpenMP 3.0; All Pair Shortest Path; Task Parallelization

I. INTRODUCTION

Homogeneous multicore architectures have been used widely in the past decade. This is due to the need to have machines with high performance that are more computationally powerful than uniprocessor machines. In a homogeneous multi-core architecture, many identical processors or cores work together to perform complex tasks. Many companies such as Intel, have moved towards increasing the processor's power by adding more cores on a single chip. Most of the commodity homogeneous architectures have many duplicated CPUs on a single chip with a shared memory. The different CPUs interact with each other through shared variables.

Shared memory machines can be categorized as either Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA) architectures. In UMA machines, the CPUs have same access time to a shared primary memory. On the other hand, each CPU in NUMA has its own memory. This memory can be accessed by the CPU that it belongs to or by other CPUs. The memory access time is, therefore, non-uniform. Modern homogeneous multicore architectures with a shared memory system are also multithreaded. The cores have the capabilities of handling several threads concurrently. These architectures exploit both instruction level parallelism and thread level parallelism. There are many parallel programming languages or APIs that support a shared memory paradigm. One such API is OpenMP [11].

OpenMP contains a set of compiler directives and libraries to execute specific instructions in parallel and to divide the work among threads. OpenMP employs a fork-join paradigm. The program starts with one thread called the master thread. Then, whenever there is a parallel region in the program, the master thread invokes a set of slave threads and distributes the work among them. This operation is called fork. After forking, the threads are allocated to the processors by the runtime environment and work concurrently to solve the problem. Once the slave threads have completed their work, they are destroyed and the master thread continues until it encounters another parallel region. This operation is called join.

OpenMP is very suitable for designing algorithms for regular applications. The data structures used in these problems are structured (such as an array). The program flow and memory access patterns are also very structured and are known apriori. An example of a regular problem is matrix-vector multiplication, where A is a dense matrix, x is a vector and b is the resultant vector. In this example, the computations or operations required producing the output and data access patterns are known beforehand. On a multiprocessor system, each processor can be assigned the same vector x with certain number of data elements (a row or a given number of rows) to compute an element(s) in b. All processors perform the same computations to produce the resultant vector but with different data sets. As a result, these problems can be optimized to run on any type of architecture relatively easily. These problems are also classified as data parallel applications.

The same is not true for irregular applications. Irregular applications rely on pointer or graph-based data structures. The algorithms used to solve irregular applications are referred to as irregular algorithms. Graph problems, list ranking and unstructured grid problems are examples of irregular computations. In these computations [15], [12], [6], [10], the data size changes dynamically at runtime, leading to non-uniform memory access and communication latencies. The load or amount of work to be distributed to the threads is not known apriori. We could consider the matrix-vector multiplication as an irregular problem, if A is a sparse matrix. Since A is instance specific, the structure of A is unknown at compile time. A matrix is not necessarily the correct data structure to use since there may be many 0's in the matrix wasting memory resources. In such problems, accesses to data often have poor spatial and temporal locality leading to ineffective use of the memory hierarchy

© 2013. The authors - Published by Atlantis Press

[15].

It is important to find efficient solutions in solving irregular problems. Irregular adaptive methods [1], [6], for example, have their applications in many science and engineering problems. With muticores becoming very popular, having a standard programming language that addresses both irregular and regular applications is very important. OpenMP is one such language. In the literature, some works [13], [3], [4] have shown that OpenMP produces reduced performance when dealing with irregular computations. The earlier versions of OpenMP were not meant to handle irregular computations [8]. In 2008, the OpenMP 3.0 version introduced a directive called "task" to help develop parallel algorithms for irregular applications. The directive "task" creates independent work units to be executed. The task in OpenMP 3.0 is nothing but a thread that can be created and destroyed as needed. It can also spawn other tasks that are not possible under the previous version of OpenMP. Spawning threads allows dynamic creation of threads incorporating fine grained parallelism and exploiting load balancing at runtime which is important for performance improvement in irregular computations.

In this paper we focus on one graph problem, all pair shortest path (APSP) problem and its implementation on OpenMP 3.0.

II. RELATED WORK

APSP can be solved using Floyd Warshall's algorithm. Venkataraman et al. [14] proposed a blocked algorithm to find APSP. Their algorithm exploits cache locality to optimize cache performance. The algorithm divides the adjacency ma-trix into blocks of B×B and each block processes individually in B iterations. They tested their blocked algorithm on two different machines, Sun Ultra Entrprise 4000/5000 and SGI O2. Their blocked algorithm delivers a speedup between 1.6 to 1.9 for graphs that are between 480 to 3200 vertices on Sun Ultra Entrprise 4000/5000 and 1.6 to 2 on SGI O2 for graphs that are between 240 to 1200 vertices. Likewise, Ma et al. [7] developed parallel Floyd Warshall's algorithm for multi-core architecture on threading building blocks (TBB). TBB is a parallel programming model for C++ code. It is a runtime based programming model that specifies tasks. The task is mapped to threads. However, unlike Venkatraman et al., Ma et al. use task and data level parallelism available in the algorithm to find all pair shortest paths. The results reveal that the parallel algorithm surpasses both serial and single threaded algorithms by 57.26% and 50.06% respectively.

Recently, Jasika et al. [5] used Dijkstra's algorithm for APSP. They used OpenMP to parallelize Dijkstra algorithm. They use the algorithm to find the single source shortest path for every vertex. They compared the OpenMP implementation to OpenCL [9] and showed that there was no gain in perfor-mance in the two implementations. This they showed is due to the inherent sequential nature of Dijkstra's algorithm problems which makes this algorithm very difficult to be efficiently parallelized.

III. IMPLEMENTATION AND RESULTS

There are two algorithms to find APSP which are Floyd-Warshall and Dijkstra algorithms. As mentioned in section 2, Dijkstra is not an efficient algorithm to be used in parallel. Therefore, in this work we consider Floyed-Warshall's algo-rithm. We use the new directive called "collapse" available in OpenMP 3.0 to handle nested loops. This directive deals efficiently with multi-dimensional loops. In other words, it combines multiple loops into single loop. Thus, by using "collapse" directive, we avoid the overhead of spawning of the nested loop in the algorithm. Also, we create a task for each vertex and process them in parallel since each vertex is independent of each other. Algorithm 1 shows our proposed parallel APSP.

Algorithm 1: Parallel APSP Algorithm

```
Input: G = (V; E)
begin
    Cost(i; j)        1
    Wight(i; j) = Cost(i; j)
    for i   0 to n do in parallel
        Collapse (2)
        for j   0 to n do
            for k   0 to n do
                Cost(j; k) =
                min(Cost(j; k); Cost(j; i) + Cost(i; k))
```

IV. RESULTS

This section shows our results for our parallel APSP algorithm. We report results on an AMD Accelerated Processing Unit (APU) 8 quad-core machine. Each core has clocks speed of 3.0 MHz and 48GB of RAM memory. We used GCC 4.4 compiler to compile and run the algorithm. We implemented our algorithm on two types of graphs:

- R-MAT graphs: These are random graphs [2] allowing high and low degree vertices.
- SSCA#2 graphs: Graphs in this category have high connected cliques. The size of the clique is distributed uniformly. Then, they generate edges of inter-clique with a chosen probability.

We used undirected graphs for our experiments. We start from 16 vertices and increase the number of vertices to 4096. We compare with OpenMP 2.5 and newer OpenMP 3.0 versions for both types of graphs.

TABLE I: The execution time on SSAC#2

Number of vertices	OpenMP 3.0	OpenMP 2.5
16	0.002	0.001
32	0.003	0.001
64	0.01	0.004
128	0.03	0.01
256	0.11	0.07
512	0.53	0.50
1024	3.06	4.06
2048	19.59	31.81
4096	158.85	257.47

TABLE II: The execution time on R-MAT

Number of vertices	OpenMP 3.0	OpenMP 2.5
16	0.002	0.001
32	0.003	0.001
64	0.01	0.004
128	0.03	0.01
256	0.11	0.07
512	0.73	0.52
1024	4.08	3.91
2048	21.56	31.02
4096	154.21	251.12

As shown in table I and table II and its subsequent figures 1 and 2 respectively, the algorithm runs a bit slower on OpenMP 3.0 for small number of vertices. However, for large number of vertices, the algorithm on OpenMP 3.0 surpasses the one on OpenMP 2.5 by 1.6 times. The new directive allows effective use of the OpenMP 3.0 threads. By collapsing the loops we make efficient use of the resources and also eliminate any sychronization issues between the two for loops.

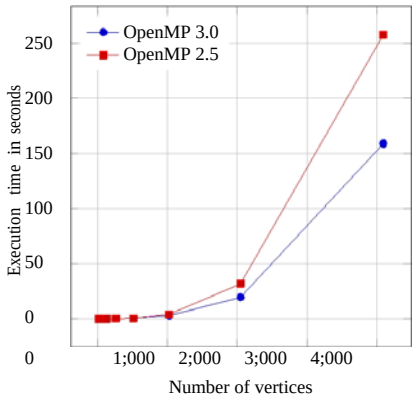

Fig. 1: Execution Time for SSAC#2

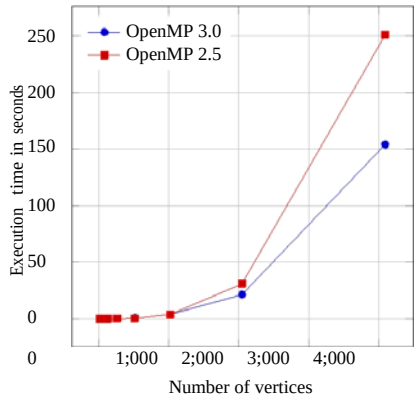

Fig. 2: Execution Time for R-MAT

V. CONCLUSION

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 for two different types of graphs.

ACKNOWLEDGMENTS

A special thanks to the Saudi Cultural Bureau in Canada that facilitated everything for us and provided the expenses of the equipments to success this project.

REFERENCES

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

[2] 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.

[3] 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 Applied to Networking & Parallel/Distributed Computing (SNPD 2000), Champagne-Ardenne, France, pages 53–60, 19–21 May 2000.

[4] 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.

[5] Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, and Novica Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In Proceedings of the 35th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2012), Opatija, Croatia, pages 1811 –1815, 21–25 May 2012.

[6] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Rama-narayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In Proceedings of the 2007 ACM SIGPLAN con-ference on Programming language design and implementation (PLDI 2007), San Diego, CA, USA, pages 211–222, 2007.

[7] Jian Ma, Ke ping Li, and Li yan Zhang. A parallel Floyd-Warshall algorithm based on TBB. In In proceesing of The 2nd IEEE International Conference on Information Management and Engineering (ICIME 2010), Bangkok, Thailand, pages 429–433, 2010.

[8] Timothy G. Mattson. How good is openmp. Scientific Programming, 11(2):81–93, 2003.

[9] Aaftab Munshi. The opencl specification. Khronos OpenCL Working Group, 1:l1–15, 2009.

[10] Jarek Nieplocha, Andres` Marquez,´ John Feo, Daniel Chavarr´ıa-Miranda, George Chin, Chad Scherrer, and Nathaniel Beagley. Evaluating the potential of multithreaded platforms for irregular scientific computa-tions. In Proceedings of the 4th international conference on Computing frontiers (CF 2007), Ischia, Italy, pages 47–58, 7–9 May 2007.

[11] OpenMP. The OpenMP API specification for parallel programming. http://openmp.org/wp/, 1998.

[12] Simone Secchi, Antonino Tumeo, and Oreste Villa. A bandwidth-optimized multi-core architecture for irregular applications. In Proceed-ings of 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (CCGrid 2012), Ottawa, ON, Canada, pages 580–587, 13–16 May 2012.

[13] Michael Sußand¨ Claudia Leopold. Implementing irregular parallel algorithms with OpenMP. In Proceedings of the 12th international conference on Parallel Processing (Euro-Par 2006), Dresden, Germany, pages 635–644, 2006.

[14] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. Journal on Experimental Algorithmics, 8, December 2003.

[15] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group

prefetch-ing. In Proceedings of the 22nd annual international symposium on computer architecture (ISCA 1995), S. Margherita Ligure, Italy, pages 188–199, 1995.